Linux Fault-injection Documentation

The kernel development community

Jan 15, 2023

CONTENTS

CHAPTER

ONE

FAULT INJECTION CAPABILITIES INFRASTRUCTURE

See also drivers/md/md-faulty.c and “every_nth” module option for scsi_debug.

1.1 Available fault injection capabilities

- failslab

  injects slab allocation failures. (kmalloc(), kmem_cache_alloc(), …)

- fail_page_alloc

  injects page allocation failures. (alloc_pages(), get_free_pages(), …)

- fail_usercopy

  injects failures in user memory access functions. (copy_from_user(), get_user(), …)

- fail_futex

  injects futex deadlock and uaddr fault errors.

- fail_sunrpc

  injects kernel RPC client and server failures.

- fail_make_request

  injects disk IO errors on devices permitted by setting /sys/block/<device>/make-it-fail or /sys/block/<device>/<partition>/make-it-fail. (submit_bio_noacct())

- fail_mmc_request

  injects MMC data errors on devices permitted by setting debugfs entries under /sys/kernel/debug/mmc0/fail_mmc_request

- fail_function

  injects error return on specific functions, which are marked by ALLOW_ERROR_INJECTION() macro, by setting debugfs entries under /sys/kernel/debug/fail_function. No boot option supported.

- NVMe fault injection

  inject NVMe status code and retry flag on devices permitted by setting debugfs entries under /sys/kernel/debug/nvme*/fault_inject. The default status code is NVME_SC_INVALID_OPCODE with no retry. The status code and retry flag can be set via the debugfs.

1.2 Configure fault-injection capabilities behavior

1.2.1 debugfs entries

fault-inject-debugfs kernel module provides some debugfs entries for runtime configuration of fault-injection capabilities.

- /sys/kernel/debug/fail*/probability:

  likelihood of failure injection, in percent.

  Format: <percent>

  Note that one-failure-per-hundred is a very high error rate for some testcases. Consider setting probability=100 and configure /sys/kernel/debug/fail*/interval for such testcases.

- /sys/kernel/debug/fail*/interval:

  specifies the interval between failures, for calls to should_fail() that pass all the other tests.

  Note that if you enable this, by setting interval>1, you will probably want to set probability=100.

- /sys/kernel/debug/fail*/times:

  specifies how many times failures may happen at most. A value of -1 means "no limit". Note, though, that this file only accepts unsigned values. So, if you want to specify -1, you better use 'printf' instead of 'echo', e.g.: $ printf %#x -1 > times

- /sys/kernel/debug/fail*/space:

  specifies an initial resource "budget", decremented by "size" on each call to should_fail(,size). Failure injection is suppressed until "space" reaches zero.

- /sys/kernel/debug/fail*/verbose

  Format: { 0 | 1 | 2 }

  specifies the verbosity of the messages when failure is injected. '0' means no messages; '1' will print only a single log line per failure; '2' will print a call trace too – useful to debug the problems revealed by fault injection.

- /sys/kernel/debug/fail*/task-filter:

  Format: { 'Y' | 'N' }

  A value of 'N' disables filtering by process (default). Any positive value limits failures to only processes indicated by /proc/<pid>/make-it-fail==1.

- /sys/kernel/debug/fail*/require-start, /sys/kernel/debug/fail*/require-end, /sys/kernel/debug/fail*/reject-start, /sys/kernel/debug/fail*/reject-end:

  specifies the range of virtual addresses tested during stacktrace walking. Failure is injected only if some caller in the walked stacktrace lies within the required range, and none lies within the rejected range. Default required range is [0,ULONG_MAX) (whole of virtual address space). Default rejected range is [0,0).

- /sys/kernel/debug/fail*/stacktrace-depth:

specifies the maximum stacktrace depth walked during search for a caller within [require-start,require-end) OR [reject-start,reject-end).

- /sys/kernel/debug/fail_page_alloc/ignore-gfp-highmem:

  Format: { 'Y' | 'N' }

  default is 'Y', setting it to 'N' will also inject failures into highmem/user allocations (__GFP_HIGHMEM allocations).

- /sys/kernel/debug/failslab/ignore-gfp-wait:
- /sys/kernel/debug/fail_page_alloc/ignore-gfp-wait:

  Format: { 'Y' | 'N' }

  default is 'Y', setting it to 'N' will also inject failures into allocations that can sleep (__GFP_DIRECT_RECLAIM allocations).

- /sys/kernel/debug/fail_page_alloc/min-order:

  specifies the minimum page allocation order to be injected failures.

- /sys/kernel/debug/fail_futex/ignore-private:

  Format: { 'Y' | 'N' }

  default is 'N', setting it to 'Y' will disable failure injections when dealing with private (address space) futexes.

- /sys/kernel/debug/fail_sunrpc/ignore-client-disconnect:

  Format: { 'Y' | 'N' }

  default is 'N', setting it to 'Y' will disable disconnect injection on the RPC client.

- /sys/kernel/debug/fail_sunrpc/ignore-server-disconnect:

  Format: { 'Y' | 'N' }

  default is 'N', setting it to 'Y' will disable disconnect injection on the RPC server.

- /sys/kernel/debug/fail_function/inject:

  Format: { 'function-name' | '!function-name' | '' }

  specifies the target function of error injection by name. If the function name leads '!' prefix, given function is removed from injection list. If nothing specified ('') injection list is cleared.

- /sys/kernel/debug/fail_function/injectable:

  (read only) shows error injectable functions and what type of error values can be specified. The error type will be one of below; - NULL: retval must be 0. - ERRNO: retval must be -1 to -MAX_ERRNO (-4096). - ERR_NULL: retval must be 0 or -1 to -MAX_ERRNO (-4096).

- /sys/kernel/debug/fail_function/<function-name>/retval:

  specifies the "error" return value to inject to the given function. This will be created when the user specifies a new injection entry. Note that this file only accepts unsigned values. So, if you want to use a negative errno, you better use 'printf' instead of 'echo', e.g.: $ printf %#x -12 > retval

1.2.2 Boot option

In order to inject faults while debugfs is not available (early boot time), use the boot option:

```
failslab=
fail_page_alloc=
fail_usercopy=
fail_make_request=
fail_futex=
mmc_core.fail_request=<interval>,<probability>,<space>,<times>
```

1.2.3 proc entries

- /proc/<pid>/fail-nth, /proc/self/task/<tid>/fail-nth:

  Write to this file of integer N makes N-th call in the task fail. Read from this file returns a integer value. A value of '0' indicates that the fault setup with a previous write to this file was injected. A positive integer N indicates that the fault wasn't yet injected. Note that this file enables all types of faults (slab, futex, etc). This setting takes precedence over all other generic debugfs settings like probability, interval, times, etc. But per-capability settings (e.g. fail_futex/ignore-private) take precedence over it.

  This feature is intended for systematic testing of faults in a single system call. See an example below.

1.3 How to add new fault injection capability

- #include <linux/fault-inject.h>
- define the fault attributes

  DECLARE_FAULT_ATTR(name);

  Please see the definition of struct fault_attr in fault-inject.h for details.
- provide a way to configure fault attributes
- boot option

  If you need to enable the fault injection capability from boot time, you can provide boot option to configure it. There is a helper function for it:

  setup_fault_attr(attr, str);
- debugfs entries

  failslab, fail_page_alloc, fail_usercopy, and fail_make_request use this way. Helper functions:

  fault_create_debugfs_attr(name, parent, attr);
- module parameters

  If the scope of the fault injection capability is limited to a single kernel module, it is better to provide module parameters to configure the fault attributes.

- add a hook to insert failures

  Upon should_fail() returning true, client code should inject a failure:

  > should_fail(attr, size);

1.4 Application Examples

- Inject slab allocation failures into module init/exit code:

```
#!/bin/bash

FAILTYPE=failslab
echo Y > /sys/kernel/debug/$FAILTYPE/task-filter
echo 10 > /sys/kernel/debug/$FAILTYPE/probability
echo 100 > /sys/kernel/debug/$FAILTYPE/interval
printf %#x -1 > /sys/kernel/debug/$FAILTYPE/times
echo 0 > /sys/kernel/debug/$FAILTYPE/space
echo 2 > /sys/kernel/debug/$FAILTYPE/verbose
echo Y > /sys/kernel/debug/$FAILTYPE/ignore-gfp-wait

faulty_system()
{
    bash -c "echo 1 > /proc/self/make-it-fail && exec $*"
}

if [ $# -eq 0 ]
then
    echo "Usage: $0 modulename [ modulename ... ]"
    exit 1
fi

for m in $*
do
    echo inserting $m...
    faulty_system modprobe $m

    echo removing $m...
    faulty_system modprobe -r $m
done
```

- Inject page allocation failures only for a specific module:

```
#!/bin/bash

FAILTYPE=fail_page_alloc
module=$1

if [ -z $module ]
then
```

```
    echo "Usage: $0 <modulename>"
    exit 1
fi

modprobe $module

if [ ! -d /sys/module/$module/sections ]
then
    echo Module $module is not loaded
    exit 1
fi

cat /sys/module/$module/sections/.text > /sys/kernel/debug/$FAILTYPE/
↪require-start
cat /sys/module/$module/sections/.data > /sys/kernel/debug/$FAILTYPE/
↪require-end

echo N > /sys/kernel/debug/$FAILTYPE/task-filter
echo 10 > /sys/kernel/debug/$FAILTYPE/probability
echo 100 > /sys/kernel/debug/$FAILTYPE/interval
printf %#x -1 > /sys/kernel/debug/$FAILTYPE/times
echo 0 > /sys/kernel/debug/$FAILTYPE/space
echo 2 > /sys/kernel/debug/$FAILTYPE/verbose
echo Y > /sys/kernel/debug/$FAILTYPE/ignore-gfp-wait
echo Y > /sys/kernel/debug/$FAILTYPE/ignore-gfp-highmem
echo 10 > /sys/kernel/debug/$FAILTYPE/stacktrace-depth

trap "echo 0 > /sys/kernel/debug/$FAILTYPE/probability" SIGINT SIGTERM EXIT

echo "Injecting errors into the module $module... (interrupt to stop)"
sleep 1000000
```

- Inject open_ctree error while btrfs mount:

```
#!/bin/bash

rm -f testfile.img
dd if=/dev/zero of=testfile.img bs=1M seek=1000 count=1
DEVICE=$(losetup --show -f testfile.img)
mkfs.btrfs -f $DEVICE
mkdir -p tmpmnt

FAILTYPE=fail_function
FAILFUNC=open_ctree
echo $FAILFUNC > /sys/kernel/debug/$FAILTYPE/inject
printf %#x -12 > /sys/kernel/debug/$FAILTYPE/$FAILFUNC/retval
echo N > /sys/kernel/debug/$FAILTYPE/task-filter
echo 100 > /sys/kernel/debug/$FAILTYPE/probability
echo 0 > /sys/kernel/debug/$FAILTYPE/interval
printf %#x -1 > /sys/kernel/debug/$FAILTYPE/times
```

```
echo 0 > /sys/kernel/debug/$FAILTYPE/space
echo 1 > /sys/kernel/debug/$FAILTYPE/verbose

mount -t btrfs $DEVICE tmpmnt
if [ $? -ne 0 ]
then
    echo "SUCCESS!"
else
    echo "FAILED!"
    umount tmpmnt
fi

echo > /sys/kernel/debug/$FAILTYPE/inject

rmdir tmpmnt
losetup -d $DEVICE
rm testfile.img
```

1.5 Tool to run command with failslab or fail_page_alloc

In order to make it easier to accomplish the tasks mentioned above, we can use tools/testing/fault-injection/failcmd.sh. Please run a command "./tools/testing/fault-injection/failcmd.sh –help" for more information and see the following examples.

Examples:

Run a command "make -C tools/testing/selftests/ run_tests" with injecting slab allocation failure:

```
# ./tools/testing/fault-injection/failcmd.sh \
        -- make -C tools/testing/selftests/ run_tests
```

Same as above except to specify 100 times failures at most instead of one time at most by default:

```
# ./tools/testing/fault-injection/failcmd.sh --times=100 \
        -- make -C tools/testing/selftests/ run_tests
```

Same as above except to inject page allocation failure instead of slab allocation failure:

```
# env FAILCMD_TYPE=fail_page_alloc \
        ./tools/testing/fault-injection/failcmd.sh --times=100 \
        -- make -C tools/testing/selftests/ run_tests
```

1.6 Systematic faults using fail-nth

The following code systematically faults 0-th, 1-st, 2-nd and so on capabilities in the socketpair() system call:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/syscall.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int main()
{
	int i, err, res, fail_nth, fds[2];
	char buf[128];

	system("echo N > /sys/kernel/debug/failslab/ignore-gfp-wait");
	sprintf(buf, "/proc/self/task/%ld/fail-nth", syscall(SYS_gettid));
	fail_nth = open(buf, O_RDWR);
	for (i = 1;; i++) {
		sprintf(buf, "%d", i);
		write(fail_nth, buf, strlen(buf));
		res = socketpair(AF_LOCAL, SOCK_STREAM, 0, fds);
		err = errno;
		pread(fail_nth, buf, sizeof(buf), 0);
		if (res == 0) {
			close(fds[0]);
			close(fds[1]);
		}
		printf("%d-th fault %c: res=%d/%d\n", i, atoi(buf) ? 'N' : 'Y',
			res, err);
		if (atoi(buf))
			break;
	}
	return 0;
}
```

An example output:

```
1-th fault Y: res=-1/23
2-th fault Y: res=-1/23
3-th fault Y: res=-1/12
4-th fault Y: res=-1/12
5-th fault Y: res=-1/23
6-th fault Y: res=-1/23
7-th fault Y: res=-1/23
```

```
8-th fault Y: res=-1/12
9-th fault Y: res=-1/12
10-th fault Y: res=-1/12
11-th fault Y: res=-1/12
12-th fault Y: res=-1/12
13-th fault Y: res=-1/12
14-th fault Y: res=-1/12
15-th fault Y: res=-1/12
16-th fault N: res=0/12
```

CHAPTER

TWO

NOTIFIER ERROR INJECTION

Notifier error injection provides the ability to inject artificial errors to specified notifier chain callbacks. It is useful to test the error handling of notifier call chain failures which is rarely executed. There are kernel modules that can be used to test the following notifiers.

- PM notifier
- Memory hotplug notifier
- powerpc pSeries reconfig notifier
- Netdevice notifier

2.1 PM notifier error injection module

This feature is controlled through debugfs interface

/sys/kernel/debug/notifier-error-inject/pm/actions/<notifier event>/error

Possible PM notifier events to be failed are:

- PM_HIBERNATION_PREPARE
- PM_SUSPEND_PREPARE
- PM_RESTORE_PREPARE

Example: Inject PM suspend error (-12 = -ENOMEM):

```
# cd /sys/kernel/debug/notifier-error-inject/pm/
# echo -12 > actions/PM_SUSPEND_PREPARE/error
# echo mem > /sys/power/state
bash: echo: write error: Cannot allocate memory
```

2.2 Memory hotplug notifier error injection module

This feature is controlled through debugfs interface

/sys/kernel/debug/notifier-error-inject/memory/actions/<notifier event>/error

Possible memory notifier events to be failed are:

- MEM_GOING_ONLINE
- MEM_GOING_OFFLINE

Example: Inject memory hotplug offline error (-12 == -ENOMEM):

```
# cd /sys/kernel/debug/notifier-error-inject/memory
# echo -12 > actions/MEM_GOING_OFFLINE/error
# echo offline > /sys/devices/system/memory/memoryXXX/state
bash: echo: write error: Cannot allocate memory
```

2.3 powerpc pSeries reconfig notifier error injection module

This feature is controlled through debugfs interface

/sys/kernel/debug/notifier-error-inject/pSeries-reconfig/actions/<notifier event>/error

Possible pSeries reconfig notifier events to be failed are:

- PSERIES_RECONFIG_ADD
- PSERIES_RECONFIG_REMOVE
- PSERIES_DRCONF_MEM_ADD
- PSERIES_DRCONF_MEM_REMOVE

2.4 Netdevice notifier error injection module

This feature is controlled through debugfs interface

/sys/kernel/debug/notifier-error-inject/netdev/actions/<notifier event>/error

Netdevice notifier events which can be failed are:

- NETDEV_REGISTER
- NETDEV_CHANGEMTU
- NETDEV_CHANGENAME
- NETDEV_PRE_UP
- NETDEV_PRE_TYPE_CHANGE
- NETDEV_POST_INIT
- NETDEV_PRECHANGEMTU

- NETDEV_PRECHANGEUPPER
- NETDEV_CHANGEUPPER

Example: Inject netdevice mtu change error (-22 == -EINVAL):

```
# cd /sys/kernel/debug/notifier-error-inject/netdev
# echo -22 > actions/NETDEV_CHANGEMTU/error
# ip link set eth0 mtu 1024
RTNETLINK answers: Invalid argument
```

2.5 For more usage examples

There are tools/testing/selftests using the notifier error injection features for CPU and memory notifiers.

- tools/testing/selftests/cpu-hotplug/on-off-test.sh
- tools/testing/selftests/memory-hotplug/on-off-test.sh

These scripts first do simple online and offline tests and then do fault injection tests if notifier error injection module is available.

CHAPTER

THREE

NVME FAULT INJECTION

Linux's fault injection framework provides a systematic way to support error injection via debugfs in the /sys/kernel/debug directory. When enabled, the default NVME_SC_INVALID_OPCODE with no retry will be injected into the nvme_try_complete_req. Users can change the default status code and no retry flag via the debugfs. The list of Generic Command Status can be found in include/linux/nvme.h

Following examples show how to inject an error into the nvme.

First, enable CONFIG_FAULT_INJECTION_DEBUG_FS kernel config, recompile the kernel. After booting up the kernel, do the following.

3.1 Example 1: Inject default status code with no retry

```
mount /dev/nvme0n1 /mnt
echo 1 > /sys/kernel/debug/nvme0n1/fault_inject/times
echo 100 > /sys/kernel/debug/nvme0n1/fault_inject/probability
cp a.file /mnt
```

Expected Result:

```
cp: cannot stat ‘/mnt/a.file’: Input/output error
```

Message from dmesg:

```
FAULT_INJECTION: forcing a failure.
name fault_inject, interval 1, probability 100, space 0, times 1
CPU: 0 PID: 0 Comm: swapper/0 Not tainted 4.15.0-rc8+ #2
Hardware name: innotek GmbH VirtualBox/VirtualBox,
BIOS VirtualBox 12/01/2006
Call Trace:
  <IRQ>
  dump_stack+0x5c/0x7d
  should_fail+0x148/0x170
  nvme_should_fail+0x2f/0x50 [nvme_core]
  nvme_process_cq+0xe7/0x1d0 [nvme]
  nvme_irq+0x1e/0x40 [nvme]
  __handle_irq_event_percpu+0x3a/0x190
  handle_irq_event_percpu+0x30/0x70
  handle_irq_event+0x36/0x60
```

```
  handle_fasteoi_irq+0x78/0x120
  handle_irq+0xa7/0x130
  ? tick_irq_enter+0xa8/0xc0
  do_IRQ+0x43/0xc0
  common_interrupt+0xa2/0xa2
  </IRQ>
RIP: 0010:native_safe_halt+0x2/0x10
RSP: 0018:ffffffff82003e90 EFLAGS: 00000246 ORIG_RAX: ffffffffffffffdd
RAX: ffffffff817a10c0 RBX: ffffffff82012480 RCX: 0000000000000000
RDX: 0000000000000000 RSI: 0000000000000000 RDI: 0000000000000000
RBP: 0000000000000000 R08: 000000008e38ce64 R09: 0000000000000000
R10: 0000000000000000 R11: 0000000000000000 R12: ffffffff82012480
R13: ffffffff82012480 R14: 0000000000000000 R15: 0000000000000000
  ? __sched_text_end+0x4/0x4
  default_idle+0x18/0xf0
  do_idle+0x150/0x1d0
  cpu_startup_entry+0x6f/0x80
  start_kernel+0x4c4/0x4e4
  ? set_init_arg+0x55/0x55
  secondary_startup_64+0xa5/0xb0
  print_req_error: I/O error, dev nvme0n1, sector 9240
EXT4-fs error (device nvme0n1): ext4_find_entry:1436:
inode #2: comm cp: reading directory lblock 0
```

3.2 Example 2: Inject default status code with retry

```
mount /dev/nvme0n1 /mnt
echo 1 > /sys/kernel/debug/nvme0n1/fault_inject/times
echo 100 > /sys/kernel/debug/nvme0n1/fault_inject/probability
echo 1 > /sys/kernel/debug/nvme0n1/fault_inject/status
echo 0 > /sys/kernel/debug/nvme0n1/fault_inject/dont_retry

cp a.file /mnt
```

Expected Result:

```
command success without error
```

Message from dmesg:

```
FAULT_INJECTION: forcing a failure.
name fault_inject, interval 1, probability 100, space 0, times 1
CPU: 1 PID: 0 Comm: swapper/1 Not tainted 4.15.0-rc8+ #4
Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
Call Trace:
  <IRQ>
  dump_stack+0x5c/0x7d
  should_fail+0x148/0x170
  nvme_should_fail+0x30/0x60 [nvme_core]
```

```
  nvme_loop_queue_response+0x84/0x110 [nvme_loop]
  nvmet_req_complete+0x11/0x40 [nvmet]
  nvmet_bio_done+0x28/0x40 [nvmet]
  blk_update_request+0xb0/0x310
  blk_mq_end_request+0x18/0x60
  flush_smp_call_function_queue+0x3d/0xf0
  smp_call_function_single_interrupt+0x2c/0xc0
  call_function_single_interrupt+0xa2/0xb0
  </IRQ>
RIP: 0010:native_safe_halt+0x2/0x10
RSP: 0018:ffffc9000068bec0 EFLAGS: 00000246 ORIG_RAX: ffffffffffffff04
RAX: ffffffff817a10c0 RBX: ffff88011a3c9680 RCX: 0000000000000000
RDX: 0000000000000000 RSI: 0000000000000000 RDI: 0000000000000000
RBP: 0000000000000001 R08: 000000008e38c131 R09: 0000000000000000
R10: 0000000000000000 R11: 0000000000000000 R12: ffff88011a3c9680
R13: ffff88011a3c9680 R14: 0000000000000000 R15: 0000000000000000
  ? __sched_text_end+0x4/0x4
  default_idle+0x18/0xf0
  do_idle+0x150/0x1d0
  cpu_startup_entry+0x6f/0x80
  start_secondary+0x187/0x1e0
  secondary_startup_64+0xa5/0xb0
```

3.3 Example 3: Inject an error into the 10th admin command

```
echo 100 > /sys/kernel/debug/nvme0/fault_inject/probability
echo 10 > /sys/kernel/debug/nvme0/fault_inject/space
echo 1 > /sys/kernel/debug/nvme0/fault_inject/times
nvme reset /dev/nvme0
```

Expected Result:

```
After NVMe controller reset, the reinitialization may or may not succeed.
It depends on which admin command is actually forced to fail.
```

Message from dmesg:

```
nvme nvme0: resetting controller
FAULT_INJECTION: forcing a failure.
name fault_inject, interval 1, probability 100, space 1, times 1
CPU: 0 PID: 0 Comm: swapper/0 Not tainted 5.2.0-rc2+ #2
Hardware name: MSI MS-7A45/B150M MORTAR ARCTIC (MS-7A45), BIOS 1.50 04/25/2017
Call Trace:
 <IRQ>
 dump_stack+0x63/0x85
 should_fail+0x14a/0x170
 nvme_should_fail+0x38/0x80 [nvme_core]
 nvme_irq+0x129/0x280 [nvme]
 ? blk_mq_end_request+0xb3/0x120
```

```
 __handle_irq_event_percpu+0x84/0x1a0
 handle_irq_event_percpu+0x32/0x80
 handle_irq_event+0x3b/0x60
 handle_edge_irq+0x7f/0x1a0
 handle_irq+0x20/0x30
 do_IRQ+0x4e/0xe0
 common_interrupt+0xf/0xf
 </IRQ>
RIP: 0010:cpuidle_enter_state+0xc5/0x460
Code: ff e8 8f 5f 86 ff 80 7d c7 00 74 17 9c 58 0f 1f 44 00 00 f6 c4 02 0f 85␣
↪69 03 00 00 31 ff e8 62 aa 8c ff fb 66 0f 1f 44 00 00 <45> 85 ed 0f 88 37 03␣
↪00 00 4c 8b 45 d0 4c 2b 45 b8 48 ba cf f7 53
RSP: 0018:ffffffff88c03dd0 EFLAGS: 00000246 ORIG_RAX: ffffffffffffffdc
RAX: ffff9dac25a2ac80 RBX: ffffffff88d53760 RCX: 000000000000001f
RDX: 0000000000000000 RSI: 000000002d958403 RDI: 0000000000000000
RBP: ffffffff88c03e18 R08: fffffff75e35ffb7 R09: 00000a49a56c0b48
R10: ffffffff88c03da0 R11: 0000000000001b0c R12: ffff9dac25a34d00
R13: 0000000000000006 R14: 0000000000000006 R15: ffffffff88d53760
 cpuidle_enter+0x2e/0x40
 call_cpuidle+0x23/0x40
 do_idle+0x201/0x280
 cpu_startup_entry+0x1d/0x20
 rest_init+0xaa/0xb0
 arch_call_rest_init+0xe/0x1b
 start_kernel+0x51c/0x53b
 x86_64_start_reservations+0x24/0x26
 x86_64_start_kernel+0x74/0x77
 secondary_startup_64+0xa4/0xb0
nvme nvme0: Could not set queue count (16385)
nvme nvme0: IO queues not created
```

CHAPTER

FOUR

PROVOKING CRASHES WITH LINUX KERNEL DUMP TEST MODULE (LKDTM)

The lkdtm module provides an interface to disrupt (and usually crash) the kernel at predefined code locations to evaluate the reliability of the kernel's exception handling and to test crash dumps obtained using different dumping solutions. The module uses KPROBEs to instrument the trigger location, but can also trigger the kernel directly without KPROBE support via debugfs.

You can select the location of the trigger ("crash point name") and the type of action ("crash point type") either through module arguments when inserting the module, or through the debugfs interface.

Usage:

```
insmod lkdtm.ko [recur_count={>0}] cpoint_name=<> cpoint_type=<>
                [cpoint_count={>0}]
```

recur_count Recursion level for the stack overflow test. By default this is dynamically calculated based on kernel configuration, with the goal of being just large enough to exhaust the kernel stack. The value can be seen at */sys/module/lkdtm/parameters/recur_count*.

cpoint_name Where in the kernel to trigger the action. It can be one of INT_HARDWARE_ENTRY, INT_HW_IRQ_EN, INT_TASKLET_ENTRY, FS_DEVRW, MEM_SWAPOUT, TIMERADD, SCSI_QUEUE_RQ, or DIRECT.

cpoint_type Indicates the action to be taken on hitting the crash point. These are numerous, and best queried directly from debugfs. Some of the common ones are PANIC, BUG, EXCEPTION, LOOP, and OVERFLOW. See the contents of */sys/kernel/debug/provoke-crash/DIRECT* for a complete list.

cpoint_count Indicates the number of times the crash point is to be hit before triggering the action. The default is 10 (except for DIRECT, which always fires immediately).

You can also induce failures by mounting debugfs and writing the type to <debugfs>/provoke-crash/<crashpoint>. E.g.:

```
mount -t debugfs debugfs /sys/kernel/debug
echo EXCEPTION > /sys/kernel/debug/provoke-crash/INT_HARDWARE_ENTRY
```

The special file *DIRECT* will induce the action directly without KPROBE instrumentation. This mode is the only one available when the module is built for a kernel without KPROBEs support:

```
# Instead of having a BUG kill your shell, have it kill "cat":
cat <(echo WRITE_RO) >/sys/kernel/debug/provoke-crash/DIRECT
```