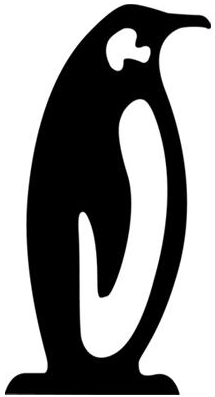


東京 2025 LINUX PLUMBERS CONFERENCE

TOKYO, JAPAN / DECEMBER 11-13, 2025

Module Error Injection with eBPF

Daniel Gomez

東京 2025
LINUX PLUMBERS CONFERENCE
TOKYO, JAPAN / DEC. 11-13, 2025

SAMSUNG

Fault Injection

- Fault Injection framework (2006)
  - should_fail()
  - debugfs
  - Examples: /sys/kernel/debug/nvme0/fault_inject/
    /sys/kernel/debug/failslab/

Fault Injection and BPF Error Injection

- Fault Injection framework (2006)
  - should_fail()
  - **debugfs**
  - **Examples:** /sys/kernel/debug/nvme0/fault_inject/
    /sys/kernel/debug/failslab/
- BPF Error Injection (2017)
  - **Introduced by Josef Bacik in 2017**
  - BPF_ALLOW_ERROR_INJECTION()
  - bpf_override_return() BPF helper
  - Support for kprobes
  - First user: **btrfs**

Error Injection

- ~~BPF~~ Error Injection (2018)
  - Name: Fault Injection
  - Generalized from kprobe in Jan 2018 —> Error Injection Framework by Masami Hiramatsu
  - BPF_ALLOW_ERROR_INJECTION() —> ALLOW_ERROR_INJECTION()
  - CONFIG_FUNCTION_ERROR_INJECTION
  - **Debugfs support**: /sys/kernel/debug/fail_function

Adding Support

cfcbfb1382db

mm/filemap.c: enable error injection at add_to_page_cache()

```
diff --git a/mm/filemap.c b/mm/filemap.c
index 4157f858a9c6..0e929b4da48b 100644
--- a/mm/filemap.c
+++ b/mm/filemap.c
@@ -24,6 +24,7 @@
 #include <linux/pagemap.h>
 #include <linux/file.h>
 #include <linux/uio.h>
+#include <linux/error-injection.h>
 #include <linux/hash.h>
 #include <linux/writeback.h>
 #include <linux/backing-dev.h>
@@ -882,6 +883,7 @@ static int __add_to_page_cache_locked(struct page *page,
        put_page(page);
        return xas_error(&xas);
 }
+ALLOW_ERROR_INJECTION(__add_to_page_cache_locked, ERRNO);

 /**
  * add_to_page_cache_locked - add a locked page to the pagecache
```

Requirements

Documentation/fault-injection/fault-injection.rst

- **The function returns an error code if it fails, and the callers must check it correctly** (need to recover from it).

- **The function does not execute any code which can change any state before the first error return**. The state includes global or local, or input variable. For example, clear output address storage (e.g. `*ret = NULL`), increments/decrements counter, set a flag, preempt/irq disable or get a lock (if those are recovered before returning error, that will be OK.)

Why?

- Testing is hard
- Test error paths is harder
  - They are difficult to trigger
  - May require specific conditions
  - May require production or simulated environments
- Error paths probably under-tested
- Deterministic

Fault and Error Injection Frameworks

- Fault Injection Framework
  - Specific:
    - should_fail()
    - **Example:** /sys/kernel/debug/failslab/
  - Generic:
    - fail_function()
      - ALLOW_ERROR_INJECTION() + should_fail()
    - **Example:** /sys/kernel/debug/fail_function/

Fault and Error Injection Frameworks

- Fault Injection (via debugfs)
  - Requires in-kernel code
  - Pollutes sources
  - Arbitrary points

Fault and Error Injection Frameworks

- Fault Injection (via debugfs)
  - Requires in-kernel code
  - Pollutes sources
  - Arbitrary points

- This may lead to maintainer rejection
  - Example:
    - https://lore.kernel.org/all/20210512064629.13899-1-mcgrof@kernel.org/

Fault and Error Injection Frameworks

- Fault Injection (via debugfs)
  - Requires in-kernel code
  - Pollutes sources
  - Arbitrary points
  - **Example:** mm/failslab.c; drivers/nvme/host/fault_inject.c
  - **Error injection points are configured via** /sys/kernel/debug/fail*
  - failcmd.sh **utility**
  - Feature reach
    - Specific: failslab, fail_page_alloc, fail_sunrpc...
      - failslab: ignore-gpf-wait
      - fail_page_alloc: ignore-gfp-wait, min-order...

Fault and Error Injection Frameworks

- Fault Injection (via debugfs)
  - Requires in-kernel code
  - Pollutes sources
  - Arbitrary points
  - Example: mm/failslab.c; drivers/nvme/host/fault_inject.c
  - Error injection points are configured via /sys/kernel/debug/fail_func
  - failcmd.sh utility
  - Feature reach
    - Generic: fail_func: Probability, interval, times, space, task-filter, stacktrace failslab: ignore-gpf-wait
    - Only requires annotated functions with ALLOW_ERROR_INJECTION() tag

Fault and Error Injection Frameworks

- Fault Injection (via debugfs)
- BPF
  - Only requires annotated functions with ALLOW_ERROR_INJECTION() tag
  - Code lives in a standalone tool

Fault and Error Injection Frameworks

- Fault Injection (via debugfs)
- BPF
- XFS errortag (Custom XFS Error Injection framework via debugfs)

Tooling

- Fault Injection (via debugfs)
  - Specific: MM (slab, page_alloc), NVMe, Block Layer, Net, MTD...
    - ?
  - Generic: xe, btrfs, Microsoft Surface Agreggator, filemap...
    - IGT testsuite, xfstests, ...?
- BPF
  - xe, btrfs, Microsoft Surface Agreggator, filemap...
- Custom XFS Error Injection framework (debugfs)
  - xfs_io

Users

- 40 + 62 + 47 error injection points

Users

- ~~40~~ + 62 + ~~47~~ error injection points
  - btrfs
  - Intel Xe (37)
  - MTD
  - Microsoft Surface System Aggregator
  - Block layer
  - MM
    - page_alloc, slab (kmalloc), filemap
  - Net
    - skb, page_pool_netmems

Introducing moderr

Example: Inject error -22 to module_enable_rodata_ro_after_init() for brd module:

```
sudo moderr --modname=brd \
        —modfunc=module_enable_rodata_ro_after_init \
        —error=-22 —trace
  Monitoring module error injection... Hit Ctrl-C to end.
  MODULE    ERROR FUNCTION
  brd       -22   module_enable_rodata_after_init()

Kernel messages:
  [  89.463690] brd: module loaded
  [  89.463855] brd: module_enable_rodata_ro_after_init() returned -22,
  ro_after_init data might still be writable
```

Initial Error Injection Points

ALLOW_ERROR_INJECTION(module_enable_rodata_ro_after_init, ERRNO);

When an error occurs while setting memory to RO after module initialization. This is when module initialization reaches MODULE_STATE_LIVE stage.

ALLOW_ERROR_INJECTION(complete_formation, ERRNO);

When module initialization switches from MODULE_STATE_UNFORMED to MODULE_STATE_COMING stage.

ALLOW_ERROR_INJECTION(do_init_module, ERRNO);

When an error occurs during module initialization and before we switch from MODULE_STATE_COMING to MODULE_STATE_LIVE stage.

Additional Error Injection Points

ALLOW_ERROR_INJECTION(module_memory_alloc, ERRNO);

When allocating memory types for module. Helped to reproduce a memory leak problem.

Moderr Use Cases

1. To test error paths and validate correctness:
Enable rodata after init

2. To test and validate memory leak
- kmemleak couldn't detect it (marked with kmemleak_not_leak())
- Only found through systematic error injection
- 1.2GB memory leak in move_module() validated with moderr

Upstream moderr 1/2

Feedback:

- Valuable tool
- BPF side
  - Not valuable in-tree (No tools in the kernel repo)
  - Not clear value on the initial error injection points
  - Error injection points need to be test paths that are difficult to test
- Selftest/modules side
  - We do add random tools
  - Suggestion to go under: tools/testing/selftests/module/

Upstream moderr 2/2

Feedback:

- Why needs to be module-specific?
  - Suggestion to make it generic
  - Xe maintainer interesting in using it
    - Reminder: +30 error injection points
  - Suggestion to integrate the tool under kmod-project
  - Suggestion to have it in-tree for simpler maintenance-wise

Alternatives

IOVisor/BCC: inject.py

- Currently maintained (last commit 2024)
- Feature reach
  - eBPF tool generator
  - Detect call chain and optional predicates
- Uses the old and deprecated Python eBPF/BCC infrastructure
- No evidence of libbpf replacement exists
- What do currently users of ALLOW_ERROR_INJECTION() use?

Proposal: Promote moderr to generic

- “It is already generic”
  - Command line supports hook point + error
  - Command line also supports module specific filters
    - Module name, memory type,
  - Only supports module hooks (hardcoded)

How do we scale?

- Function discovery
- Error code injection via bpf_override_return()
- Probability and count limits?
- Call chain filtering (using bpf_get_stack(), blazesym...)
- Real-time tracing output

How do we scale?

moderr uses an enum to detect kprobe function to override the return value

1. moderr.h - **Enum**

```
enum modfunc {
    COMPLETE_FORMATION = 1,
    DO_INIT_MODULE,
    MODULE_ENABLE_RODATA_AFTER_INIT,
    // Add new entry for EACH function
};
```

How do we scale?

moderr uses an enum to detect kprobe function to override the return value

1. moderr.h - **Enum**
2. moderr.c - **String parsing**

```
static enum modfunc string_to_modfunc(char *arg) {
   if (strncmp(arg, "complete_formation", ...) == 0)
      return COMPLETE_FORMATION;
   // Add new comparison for EACH function
}
```

How do we scale?

moderr uses an enum to detect kprobe function to override the return value

1. moderr.h - **Enum**
2. moderr.c - **String parsing**
3. moderr.bpf.c - **Kprobe handler**

```
SEC("kprobe/complete_formation")
int BPF_KPROBE(complete_formation, struct module *mod, ...) {
    return module_error_injection(ctx, mod, COMPLETE_FORMATION);
}
```

Solutions to Problem 1

- bpf_get_stack()/bpf_get_stackid()
- bpf_get_func_ip()

- **Old approach: hardcoded per-function handler**

```
SEC("kprobe/complete_formation")
int BPF_KPROBE(complete_formation, struct module *mod, ...) { }
```

New Problems

- bpf_get_stack()/bpf_get_stackid()
- bpf_get_func_ip()

- Old approach: hardcoded per-function handler
- New approach: fentry with BTF

```
SEC("fentry")
int BPF_PROG(generic_entry, void *arg0, void *arg1, ...) {
    // Get function IP
  u64 func_ip = bpf_get_func_ip(ctx);
  struct module *mod = (struct module *)arg0;
  bpf_override_return(ctx, target_error);
```

New Problems

- bpf_get_stack()/bpf_get_stackid()
- bpf_get_func_ip()

- Old approach: hardcoded per-function handler
- New approach: fentry with BTF

```
SEC("fentry")
int BPF_PROG(generic_entry, void *arg0, void *arg1, ...) {
    // Get function IP
  u64 func_ip = bpf_get_func_ip(ctx);
  struct module *mod = (struct module *)arg0;
  bpf_override_return(ctx, target_error);
```

- Optional and subsytem specific
  - Allows additional filters:
    - Module name
    - Block layer device

Home

- kernel sources
- IOVisor/BCC
  - blkalgn: add support
  - https://github.com/iovisor/bcc/pull/5128
- https://git.kernel.org/errinj/errinj.git

Possible Roadmap

- Support dynamic function discovery
  - Remove enums
  - Replace kprobe with fentry
- Allow subsystem specific filtering
  - Access to arg0, arg1, etc (context)
  - Example: module name filter
  - Can we generalize this with predicates?
- Add inject.py features
  - Stack trace filter with bpf_get_stack(), (or blazesym)

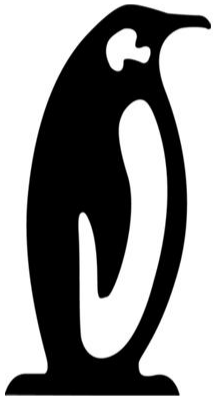
Example

Generic:

errinject —func=<function_ip> —error=-<error_code>

Subsystem specific (e.g. modules)

errinject —func=<function_ip> —error=-<error_code> —modname=brd



東京 2025
LINUX PLUMBERS CONFERENCE

TOKYO, JAPAN / DECEMBER 11-13, 2025