kcov: code coverage for fuzzing

kcov exposes kernel code coverage information in a form suitable for coverage- guided fuzzing (randomized testing). Coverage data of a running kernel is exported via the "kcov" debugfs file. Coverage collection is enabled on a task basis, and thus it can capture precise coverage of a single system call.

Note that kcov does not aim to collect as much coverage as possible. It aims to collect more or less stable coverage that is function of syscall inputs. To achieve this goal it does not collect coverage in soft/hard interrupts and instrumentation of some inherently non-deterministic parts of kernel is disabled (e.g. scheduler, locking).

kcov is also able to collect comparison operands from the instrumented code (this feature currently requires that the kernel is compiled with clang).

Prerequisites

Configure the kernel with:

```
CONFIG_KCOV=y
```

CONFIG_KCOV requires gcc 6.1.0 or later.

If the comparison operands need to be collected, set:

```
CONFIG_KCOV_ENABLE_COMPARISONS=y
```

Profiling data will only become accessible once debugfs has been mounted:

```
mount -t debugfs none /sys/kernel/debug
```

Coverage collection

The following program demonstrates coverage collection from within a test program using kcov:

```
#include <stdio.h>
#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/types.h>

#define KCOV_INIT_TRACE                     _IOR('c', 1, unsigned long)
#define KCOV_ENABLE                 _IO('c', 100)
#define KCOV_DISABLE                        _IO('c', 101)
#define COVER_SIZE                  (64<<10)

#define KCOV_TRACE_PC  0
#define KCOV_TRACE_CMP 1

int main(int argc, char **argv)
{
    int fd;
    unsigned long *cover, n, i;

    /* A single fd descriptor allows coverage collection on a single
     * thread.
     */
    fd = open("/sys/kernel/debug/kcov", O_RDWR);
    if (fd == -1)
            perror("open"), exit(1);
    /* Setup trace mode and trace size. */
    if (ioctl(fd, KCOV_INIT_TRACE, COVER_SIZE))
            perror("ioctl"), exit(1);
    /* Mmap buffer shared between kernel- and user-space. */
    cover = (unsigned long*)mmap(NULL, COVER_SIZE * sizeof(unsigned long),
                                 PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if ((void*)cover == MAP_FAILED)
            perror("mmap"), exit(1);
    /* Enable coverage collection on the current thread. */
    if (ioctl(fd, KCOV_ENABLE, KCOV_TRACE_PC))
            perror("ioctl"), exit(1);
    /* Reset coverage from the tail of the ioctl() call. */
    __atomic_store_n(&cover[0], 0, __ATOMIC_RELAXED);
    /* That's the target syscal call. */
    read(-1, NULL, 0);
    /* Read number of PCs collected. */
    n = __atomic_load_n(&cover[0], __ATOMIC_RELAXED);
    for (i = 0; i < n; i++)
            printf("0x%lx\n", cover[i + 1]);
    /* Disable coverage collection for the current thread. After this call
     * coverage can be enabled for a different thread.
     */
    if (ioctl(fd, KCOV_DISABLE, 0))
            perror("ioctl"), exit(1);
    /* Free resources. */
    if (munmap(cover, COVER_SIZE * sizeof(unsigned long)))
            perror("munmap"), exit(1);
    if (close(fd))
            perror("close"), exit(1);
    return 0;
}
```

After piping through addr2line output of the program looks as follows:

```
SyS_read
fs/read_write.c:562
__fdget_pos
fs/file.c:774
__fget_light
fs/file.c:746
__fget_light
fs/file.c:750
__fget_light
fs/file.c:760
__fdget_pos
fs/file.c:784
SyS_read
fs/read_write.c:562
```

If a program needs to collect coverage from several threads (independently), it needs to open /sys/kernel/debug/kcov in each thread separately.

The interface is fine-grained to allow efficient forking of test processes. That is, a parent process opens /sys/kernel/debug/kcov, enables trace mode, mmaps coverage buffer and then forks child processes in a loop. Child processes only need to enable coverage (disable happens automatically on thread end).

Comparison operands collection

Comparison operands collection is similar to coverage collection:

```
/* Same includes and defines as above. */

/* Number of 64-bit words per record. */
#define KCOV_WORDS_PER_CMP 4

/*
 * The format for the types of collected comparisons.
 *
 * Bit 0 shows whether one of the arguments is a compile-time constant.
 * Bits 1 & 2 contain log2 of the argument size, up to 8 bytes.
 */

#define KCOV_CMP_CONST          (1 << 0)
#define KCOV_CMP_SIZE(n)        ((n) << 1)
#define KCOV_CMP_MASK           KCOV_CMP_SIZE(3)

int main(int argc, char **argv)
{
    int fd;
    uint64_t *cover, type, arg1, arg2, is_const, size;
    unsigned long n, i;

    fd = open("/sys/kernel/debug/kcov", O_RDWR);
    if (fd == -1)
            perror("open"), exit(1);
    if (ioctl(fd, KCOV_INIT_TRACE, COVER_SIZE))
            perror("ioctl"), exit(1);
    /*
    * Note that the buffer pointer is of type uint64_t*, because all
    * the comparison operands are promoted to uint64_t.
    */
    cover = (uint64_t *)mmap(NULL, COVER_SIZE * sizeof(unsigned long),
                                  PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if ((void*)cover == MAP_FAILED)
            perror("mmap"), exit(1);
    /* Note KCOV_TRACE_CMP instead of KCOV_TRACE_PC. */
    if (ioctl(fd, KCOV_ENABLE, KCOV_TRACE_CMP))
            perror("ioctl"), exit(1);
    __atomic_store_n(&cover[0], 0, __ATOMIC_RELAXED);
    read(-1, NULL, 0);
    /* Read number of comparisons collected. */
    n = __atomic_load_n(&cover[0], __ATOMIC_RELAXED);
    for (i = 0; i < n; i++) {
            uint64_t ip;

            type = cover[i * KCOV_WORDS_PER_CMP + 1];
            /* arg1 and arg2 - operands of the comparison. */
            arg1 = cover[i * KCOV_WORDS_PER_CMP + 2];
            arg2 = cover[i * KCOV_WORDS_PER_CMP + 3];
            /* ip - caller address. */
            ip = cover[i * KCOV_WORDS_PER_CMP + 4];
            /* size of the operands. */
            size = 1 << ((type & KCOV_CMP_MASK) >> 1);
            /* is_const - true if either operand is a compile-time constant.*/
            is_const = type & KCOV_CMP_CONST;
            printf("ip: 0x%lx type: 0x%lx, arg1: 0x%lx, arg2: 0x%lx, "
                    "size: %lu, %s\n",
                    ip, type, arg1, arg2, size,
            is_const ? "const" : "non-const");
    }
    if (ioctl(fd, KCOV_DISABLE, 0))
            perror("ioctl"), exit(1);
    /* Free resources. */
    if (munmap(cover, COVER_SIZE * sizeof(unsigned long)))
            perror("munmap"), exit(1);
    if (close(fd))
            perror("close"), exit(1);
    return 0;
}
```

Note that the kcov modes (coverage collection or comparison operands) are mutually exclusive.

Remote coverage collection

With KCOV_ENABLE coverage is collected only for syscalls that are issued from the current process. With KCOV_REMOTE_ENABLE it's possible to collect coverage for arbitrary parts of the kernel code, provided that those parts are annotated with kcov_remote_start()/kcov_remote_stop().

This allows to collect coverage from two types of kernel background threads: the global ones, that are spawned during kernel boot in a limited number of instances (e.g. one USB hub_event() worker thread is spawned per USB HCD); and the local ones, that are spawned when a user interacts with some kernel interface (e.g. vhost workers); as well as from soft interrupts.

To enable collecting coverage from a global background thread or from a softirq, a unique global handle must be assigned and passed to the corresponding kcov_remote_start() call. Then a userspace process can pass a list of such handles to the KCOV_REMOTE_ENABLE ioctl in the handles array field of the kcov_remote_arg struct. This will attach the used kcov device to the code sections, that are referenced by those handles.

Since there might be many local background threads spawned from different userspace processes, we can't use a single global handle per annotation. Instead, the userspace process passes a non-zero handle through the common_handle field of the kcov_remote_arg struct. This common handle gets saved to the kcov_handle field in the current task_struct and needs to be passed to the newly spawned threads via custom annotations. Those threads should in turn be annotated with kcov_remote_start()/kcov_remote_stop().

Internally kcov stores handles as u64 integers. The top byte of a handle is used to denote the id of a subsystem that this handle belongs to, and the lower 4 bytes are used to denote the id of a thread instance within that subsystem. A reserved value 0 is used as a subsystem id for common handles as they don't belong to a particular subsystem. The bytes 4-7 are currently reserved and must be zero. In the future the number of bytes used for the subsystem or handle ids might be increased.

When a particular userspace process collects coverage via a common handle, kcov will collect coverage for each code section that is annotated to use the common handle obtained as kcov_handle from the current task_struct. However non common handles allow to collect coverage selectively from different subsystems.

```
/* Same includes and defines as above. */

struct kcov_remote_arg {
    __u32           trace_mode;
    __u32           area_size;
    __u32           num_handles;
    __aligned_u64   common_handle;
    __aligned_u64   handles[0];
};
```

```c
#define KCOV_INIT_TRACE                     _IOR('c', 1, unsigned long)
#define KCOV_DISABLE                        _IO('c', 101)
#define KCOV_REMOTE_ENABLE          _IOW('c', 102, struct kcov_remote_arg)

#define COVER_SIZE  (64 << 10)

#define KCOV_TRACE_PC       0

#define KCOV_SUBSYSTEM_COMMON       (0x00ull << 56)
#define KCOV_SUBSYSTEM_USB  (0x01ull << 56)

#define KCOV_SUBSYSTEM_MASK (0xffull << 56)
#define KCOV_INSTANCE_MASK  (0xffffffffull)

static inline __u64 kcov_remote_handle(__u64 subsys, __u64 inst)
{
    if (subsys & ~KCOV_SUBSYSTEM_MASK || inst & ~KCOV_INSTANCE_MASK)
            return 0;
    return subsys | inst;
}

#define KCOV_COMMON_ID      0x42
#define KCOV_USB_BUS_NUM    1

int main(int argc, char **argv)
{
    int fd;
    unsigned long *cover, n, i;
    struct kcov_remote_arg *arg;

    fd = open("/sys/kernel/debug/kcov", O_RDWR);
    if (fd == -1)
            perror("open"), exit(1);
    if (ioctl(fd, KCOV_INIT_TRACE, COVER_SIZE))
            perror("ioctl"), exit(1);
    cover = (unsigned long*)mmap(NULL, COVER_SIZE * sizeof(unsigned long),
                             PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if ((void*)cover == MAP_FAILED)
            perror("mmap"), exit(1);

    /* Enable coverage collection via common handle and from USB bus #1. */
    arg = calloc(1, sizeof(*arg) + sizeof(uint64_t));
    if (!arg)
            perror("calloc"), exit(1);
    arg->trace_mode = KCOV_TRACE_PC;
    arg->area_size = COVER_SIZE;
    arg->num_handles = 1;
    arg->common_handle = kcov_remote_handle(KCOV_SUBSYSTEM_COMMON,
                                                    KCOV_COMMON_ID);
    arg->handles[0] = kcov_remote_handle(KCOV_SUBSYSTEM_USB,
                                            KCOV_USB_BUS_NUM);
    if (ioctl(fd, KCOV_REMOTE_ENABLE, arg))
            perror("ioctl"), free(arg), exit(1);
    free(arg);

    /*
     * Here the user needs to trigger execution of a kernel code section
     * that is either annotated with the common handle, or to trigger some
     * activity on USB bus #1.
     */
    sleep(2);

    n = __atomic_load_n(&cover[0], __ATOMIC_RELAXED);
    for (i = 0; i < n; i++)
            printf("0x%lx\n", cover[i + 1]);
    if (ioctl(fd, KCOV_DISABLE, 0))
            perror("ioctl"), exit(1);
    if (munmap(cover, COVER_SIZE * sizeof(unsigned long)))
            perror("munmap"), exit(1);
    if (close(fd))
            perror("close"), exit(1);
    return 0;
}
```