Report 3: Kyua test program for mkdir(2)

Explicit System Call Testing

The test application would trigger all Syscalls one by one, evaluating that the audit record contains all the expected parameters, e.g the arguments, valid argument types, return values etc. The testing will be done for various success and failure modes, with cross checking for appropriate error codes in case of failure mode.

Repository

AuditTestSuite

The Problem

The previous two reports outlined the general effort required to build the Audit Test Suite. Although the results were cool and the automation was successful to some extent, there was a major issue with the structure - * It would be difficult to test the `audit(4)` system along with the entire operating system for any regressions using Kyua. * The automation tool I developed didn't have much scope to inculcate the tests for arguments and errno codes. The tests would have broken eventually.

After a thorough discussion with my mentor, we came to a conclusion that developing independent tests using atf(1) libraries would be a much better option than creating everything from scratch.

Following points were to be kept in mind :- * The tests must not rely on each other for execution. It should be possible to run them in parallel unless specified otherwise. * The tests must not depend on the state of the auditd(8) at the time of execution. If some changes are unavoidable, everything must be restored to it's initial state. * The tests must not depend on the state of the `/etc/rc.conf`. It may be possible that audit daemon was not even enabled in the OS. This should not lead to failure of the tests.

Helper Functions

Each test program would have a set of functions to start/stop the audit daemon. Apart from that, the main function will be responsible for * Setting the appropriate IOCTL request. * Opening the auditpipe(4) device and polling onto it to listen to any waiting event. * If the audit daemon was already running, there is no need to check for it's audit. However, if it was started by us, we need to make sure that the daemon was properly running before we test the audit of mkdir(2). * Use praudit(1) to extract the audit tokens from the binary data. This can be stored in another temporary file for the final confirmation. * Check if the tests pass. Compile the results and do the cleanup.

Auditpipe IOCTLs used

1) AUDITPIPE_PRESELECT_MODE_LOCAL (For local mode testing) 2) AUDITPIPE_SET_PRESELECT_MODE (Set the mode as local) 3) AUDITPIPE_SET_PRESELECT_FLAGS (Set the appropriate flag) 4) AUDITPIPE_FLUSH (Flush any outsanding data)

```
    /*
     * The next three steps ensure that the auditpipe(4) does not depend
     * on the universal audit configuration at /etc/security/audit_control
     * by setting the flag mask as the corresponding class of the event
     * to be audited, mkdir(2) in this case.
     */

    /* Set local preselection mode for auditing */
    if(ioctl(fds[0].fd, AUDITPIPE_SET_PRESELECT_MODE, &fmode) < 0){
        atf_tc_fail("Preselection mode: %s", strerror(errno));
    }

    /* Set local preselection flag as (fc) for mkdir(2) */
    if(ioctl(fds[0].fd, AUDITPIPE_SET_PRESELECT_FLAGS, &fmask) < 0){
        atf_tc_fail("Preselection flag: %s", strerror(errno));
    }

    /* This removes any outstanding record on audit pipe */
    if(ioctl(fds[0].fd, AUDITPIPE_FLUSH) < 0){
        atf_tc_fail("Auditpipe flush: %s", strerror(errno));
    }
```

ppoll(2)ing at the device descriptor

Attempt here is to wait for an output at the device descriptor. Check if any event recorded is mkdir(2). If not, continue the process, else the test case was successful. Meanwhile, a clock ensure that the process does not go into an infinite loop. The system call `clock_gettime(2)` is used with the id `CLOCK_MONOTONIC` to keep track of `ppoll(2)`.

```
/*
     * Loop until the auditpipe returns something, check if it is what
     * we want else repeat the procedure until poll(2) times out.
     */
    while(true) {
        /* Update the current time left for auditpipe to return any event */
        ATF_REQUIRE_EQ(0, clock_gettime(CLOCK_MONOTONIC, &curptr));
        curptr.tv_sec = endptr.tv_sec - curptr.tv_sec;

        switch(ppoll(fds, 1, &curptr, NULL)) {
            /* ppoll(2) returns an event, check if it's the event we want */
            case 1:
                if (fds[0].revents & POLLIN) {
                    if (getrecords(path, pipefd)) {
                    /* We have confirmed mkdir(2)' audit */
                        atf_tc_pass();
                    }
                } else {
                    atf_tc_fail("Auditpipe returned an unknown event "
                                "%#x", fds[0].revents);
                } break;

            /* poll(2) timed out */
            case 0:
                atf_tc_fail("Auditpipe did not return anything within "
                            "the time limit"); break;
            /* poll(2) standard error */
            case ERROR:
                atf_tc_fail("Poll: %s", strerror(errno)); break;

            default:
                atf_tc_fail("Poll returned an unknown event");
        }
    }
```

The function `getrecords()` does all the hard work, extracting the audit records from auditpipe and then printing it to memory for further checks.

```
static bool
getrecords(char *path, FILE *pipestream)
{
    u_char *buff;
    tokenstr_t token;
    ssize_t size = BUFFLEN;
    char *del = ",", membuff[size];
    int reclen, bytesread = 0;

    /*
     * Open a stream on 'membuff' (address to memory buffer) for storing
     * the audit records in the default mode.'reclen' reads the available
     * records from auditpipe and the let's the functions au_fetch_tok(3)
     * and au_print_flags_tok(3) do their respective jobs.
     */
    FILE *memstream = fmemopen(membuff, size, "w");
    ATF_REQUIRE(reclen = au_read_rec(pipestream, &buff) != ERROR);

    /*
     * Iterate through each BSM token, extracting the bits that are
     * required to starting processing sequences.
     */
    while (bytesread < reclen) {
        if (au_fetch_tok(&token, buff + bytesread, \
             reclen - bytesread) == ERROR) {
            atf_tc_fail("Incomplete audit record");
        };

    /* Print the tokens as they are obtained, in their default form */
        au_print_flags_tok(memstream, &token, del, AU_OFLAG_NONE);
        bytesread += token.len;
    }

    free(buff); fclose(memstream);
    return atf_utils_grep_string("%s", membuff, path);
}
```

Cleanup tasks

Kyua has an amazing feature which let's us include the cleanup code for certain test cases. We can use it to our advantage to close the audit daemon if it was started by us.

For this example, The cleanup was as simple as this:-

```
ATF_TC_CLEANUP(mkdir_success, tc)
{
    system("[ -f started_auditd ] && service auditd onestop > /dev/null 2>&1");
}
```

However, a point to be noted here: `auditd(8)` is only stopped if it was running already. That is confirmed by the presence of the file "started_auditd".

Results

Final test program passed in both cases whether the auditd(8) was running already or not. The state of the machine is preserved.
Audit daemon not running

```
□ kyua test
mkdir:mkdir_success  ->  passed  [0.045s]

Results file id is usr_src_sbin_audit_tests.20180320-052603-091854
Results saved to
/root/.kyua/store/results.usr_src_sbin_audit_tests.20180320-052603-091854.db

1/1 passed (0 failed)
```

Audit daemon already running

```
□ kyua test
mkdir:mkdir_success  ->  passed  [0.025s]

Results file id is usr_src_sbin_audit_tests.20180320-052851-676424
Results saved to
/root/.kyua/store/results.usr_src_sbin_audit_tests.20180320-052851-676424.db

1/1 passed (0 failed)
```