Race detection tool

This is a data race detection result from the paper: Efficient, context-sensitive and guarded data race detection.

Experiment results.

It consists of two parts: tool comparison results and race detection results for real-world programs.

Tool comparison.

Checking results are packed at directory "comparison".

Reported races.

Due to the double-blind requirement, we only show the screenshot of bug reports.

Axel (link, confirmed). Its report is at Reports/axel.bc-bugs.txt

> Our bug scanner has reported a data race issue at text.c#L237,
>
> in function `main`, `search_sortlist` sorts threads created by search_getspeeds,

```
j = search_getspeeds( search, i );
search_sortlist( search, i );
```

> `search_getspeeds` returns 0 if `pthread_create` failed,
> would it create a part of speedtest threads, then hit one `pthread_create` failure and returns to main function? Could it result a situation that the created threads still running and `search_sortlist` writes to variables used by threads in the same time?

```
int search_getspeeds( search_t *results, int count )
{
        .......

        while( done < count )
        {
                for( i = 0; i < count; i ++ )
                {
                        if( running < results->conf->search_threads && !results[i].speed )
                        {
                                results[i].speed = SPEED_ACTIVE;
                                results[i].speed_start_time = gettime();
                                if( pthread_create( results[i].speed_thread,
                                        NULL, search_speedtest, &results[i] ) == 0 )
                                {
                                        running ++;
                                        break;
                                }
                                else
                                {
                                        return( 0 );        // hit one pthread_create failu
                                }
                        }
```

Lunchbox (link, confirmed). Its report is at Reports/Lunchbox-threadPool.bc-bug.txt

> Our bug scanner has reported a data race issue at thread.cpp#L131 and thread.cpp#L202.
>
> Followings are code snippets.

```
void Thread::_runChild()
{
    setName(boost::lexical_cast<std::string>(_impl->index));
    _impl->id._impl->pthread = pthread_self();
```

> and

```
bool Thread::start()
{
    ...
    // avoid memleak, we don't use pthread_join
    pthread_detach(_impl->id._impl->pthread);
```

> The race occurs when *_impl->id._impl->pthread = pthread_self();* and *pthread_detach(_impl->id._impl->pthread);* are executed simultaneously.

rsyslog (link, confirmed). Its report is at Reports/rsyslog-imudp.so.bc-bug.txt

Hi all,

Our bug scanner has reported a data race issue from imudp.c#L429 to imudp.c#L430

Followings are code snippets.

```
static rsRetVal
processPacket(...)
{
    ...
    if(bDoACLCheck) {
        socklen = sizeof(struct sockaddr_storage);
        if(net.CmpHost(frominet, frominetPrev, socklen) != 0) {
            ...
            if(*pbIsPermitted == 0) {
                DBGPRINTF("msg is not from an allowed sender\n");
                if(glbl.GetOption_DisallowWarning) {
                    time_t tt;
                    datetime.GetTime(&tt);
                    if(tt > ttLastDiscard + 60) {
                        ttLastDiscard = tt;
                        errmsg.LogError(0, NO_ERRCODE,
                            "UDP message from disallowed sender discarded");
                    }
                }
            }
        }
    }
    ...
}
```

`ttLastDiscard` is a static variable and the function `processPacket` could be executed parallelly. Therefore, when two threads reach the `if(tt > ttLastDiscard + 60)` simultaneously, they may be all true and then begin to assign the variable `ttLastDiscard`. However, they could corrupt the `ttLastDiscard`, leading to confused information.

A possible call trace could be as follows:

`BEGINrunInput` creates threads that execute function `wrkr`. Then `wrkr` calls `rcvMainLoop(pWrkr)`. At the last loop of `rcvMainLoop`, it calls `processSocket(pWrkr, currEvt[i].data.ptr, &frominetPrev, &bIsPermitted);`.

s3fs-fuse (link, confirmed). Its report is at Reports/s3fs.bc-bug.txt

Hi all,

Our bug scanner has reported a race condition issue from curl.cpp#L1486 to curl.cpp#L1495

Followings are code snippets.

```
bool S3fsCurl::AddUserAgent(CURL* hCurl)
{
    ...
    static bool   init = false;
    if(!init){
      strua =  "s3fs/";
      strua += VERSION;
      strua += " (commit hash ";
      strua += COMMIT_HASH_VAL;
      strua += "; ";
      strua += s3fs_crypt_lib_name();
      strua += ")";
      init = true;
    }
    ...
}
```

Note that `init` is a static variable and the function `AddUserAgent` could be executed parallelly. Therefore, when two threads reach the `if(!init)` simultaneously, they are all true and then begin to construct the string `strua`. However, they could corrupt the `strua`, leading to confused information.

A possible call trace could be as follows:

`S3fsMultiCurl::MultiPerform` creates two or more threads that execute `S3fsMultiCurl::RequestPerformWrapper`. Then `S3fsMultiCurl::RequestPerformWrapper` calls `S3fsCurl::RequestPerform`. At the end of `S3fsCurl::RequestPerform`, it calls `S3fsCurl::AddUserAgent(hCurl)`.

sofa-pbrpc (link, confirmed). Its report is at Reports/sofa-pbrpc-client.bc-bug.txt

```
static void* thread_run(void* param)
{
    ThreadParam* thread_param = reinterpret_cast<ThreadParam*>(param);
    // init
    if (thread_param->init_func && !thread_param->init_func->Run())
    {
        ...
        thread_param->init_fail = true;
    }
    thread_param->init_done = true;
```

and

```
bool start()
{
    ...
    for (int i = 0; i < _thread_num; ++i)
    {
        ....
        int ret = pthread_create(&_threads[i], NULL, &ThreadGroupImpl::thread_run, &_th
        ....
                if (_thread_params[i].init_done)
                {
                    if (_thread_params[i].init_fail)
                    {
                        init_fail = true;
                        break;
                    }
                ...
```

Finedb (link, confirmed). Its report is at Reports/FineDB-liblmdb.so.bc-bug.txt

Our bug scanner has reported a data race issue at mutex.c#L71 and global.c#L241.

Followings are code snippets.

```
void nn_mutex_lock (struct nn_mutex *self)
{
    int rc;

    rc = pthread_mutex_lock (&self->mutex);
    errnum_assert (rc == 0, rc);
}
```

and

```
static void nn_global_term (void)
{
#if defined NN_HAVE_WINDOWS
    int rc;
#endif
    struct nn_list_item *it;
    struct nn_transport *tp;

    /*  If there are no sockets remaining, uninitialise the global context. */
    nn_assert (self.socks);
```

The race occurs when the two asserts failed, causing two threads accessing *stderr* simultaneously. here is one possible call trace:
global.c:321->global.c:229->pool.c:30->worker_posix.inc:109->thread_posix.c:55->thread_posix.c:44->worker_posix.inc:194->mutex.c:71; global.c:325->global.c:241

transmission (link, confirmed). Its report is at Reports/transmission-2.92-bugs.txt

In platform.c#L127

```
 pthread_create (&t->thread, NULL, (void* (*)(void*))ThreadFunc, t);
 pthread_detach (t->thread);
```

and the implementation of ThreadFunc

```
ThreadFunc (void * _t)
{
  tr_thread * t = _t;

  t->func (t->arg);

  tr_free (t);
  .....
}
```

Could `t` be freed in `ThreadFunc` before `t` passed into `pthread_detach` ? (don't sure it is possible in reality)

leveldb (link, confirmed). Its report is at Reports/leveldb-db_test.bc-bug.txt

Our bug scanner has reported a data race issue at env_posix.cc#L551

Followings are code snippets.

```
void PosixEnv::Schedule(void (*function)(void*), void* arg) {
  ...
  if (!started_bgthread_) {
    started_bgthread_ = true;
    PthreadCall(
        "create thread",
        pthread_create(&bgthread_, NULL,  &PosixEnv::BGThreadWrapper, this));
  }
  ...
  PthreadCall("unlock", pthread_mutex_unlock(&mu_));
}

void BGThread();
  static void* BGThreadWrapper(void* arg) {
    reinterpret_cast<PosixEnv*>(arg)->BGThread();
    return NULL;
  }

void PosixEnv::BGThread() {
  while (true) {
    ...
    PthreadCall("unlock", pthread_mutex_unlock(&mu_));
    ...
  }
}

void PthreadCall(const char* label, int result) {
    if (result != 0) {
      fprintf(stderr, "pthread %s: %s\n", label, strerror(result));
      ...
    }
  }
```

The call trace is as follows:

`PosixEnv::Schedule->PthreadCall`

and

`PosixEnv::Schedule->BGThreadWrapper->PosixEnv::BGThread->PthreadCall`

The race may be rarely triggered, but it would corrupt the log when occured.

zfs (this, no response). Its report is at Reports/zfs-zfs.bc-bug.txt

Hi all,

Our bug scanner has reported a data race issue at libzfs_diff.c#L461 (or libzfs_diff.c#L446) and libzfs_diff.c#L806.

Followings are code snippets.

```
static void *
differ(void *arg)
{
    ...
    if (rv < 0 || (rv == 0 && len != sizeof (dr))) {
        di->zerr = EPIPE;
        break;
    }
    ...
    default:
        di->zerr = EPIPE;
        break;
```

and

```
int
zfs_show_diffs(zfs_handle_t *zhp, int outfd, const char *fromsnap,
    const char *tosnap, int flags)
{
    ...
        iocerr = ioctl(zhp->zfs_hdl->libzfs_fd, ZFS_IOC_DIFF, &zc);
        if (iocerr != 0) {
            ...
            if (errno == EPERM) {
                ...
            } else if (errno == EXDEV) {
                ...
            } else if (errno != EPIPE || di.zerr == 0) {
                zfs_error_aux(zhp->zfs_hdl, strerror(errno));
            }
    ...
```

The call trace is as follows:

zfs_show_diffs->pthread_create(&tid, NULL, differ, &di)->di->zerr = EPIPE;

and

zfs_show_diffs->errno != EPIPE || di.zerr == 0;

lwan (this, no response). Its report is at Reports/lwan.bc-bug.txt

Our bug scanner has reported several data race issues. One is at lwan-status.c#L52 and lwan-status.c#L220

Followings are code snippets.

```
    void                                                \
    lwan_status_##fn_name_##_debug(const char *file,    \
        const int line, const char *func,               \
        const char *fmt, ...)                           \
    {                                                   \
      if (!quiet) {                                     \
         va_list values;                                \
         va_start(values, fmt);
```

and

```
void
lwan_status_init(struct lwan *l)
{
    #ifdef NDEBUG
      quiet = l->config.quiet;
    #else
      quiet = false;
      (void) l;
```

The possible call trace is as follows:

lwan_init_with_config->lwan_job_thread_init()->pthread_create(&self, NULL, job_thread, NULL)->lwan_status_critical("Could not lock job wait mutex");

and

lwan_init_with_config->lwan_status_init(l);

Additionally, the global variable *use_colors* is used at four positions: status_out_msg, get_color_start_for_type, get_color_end_for_type and lwan_status_init. The last access is write, which may result in three data races whose traces are similar to the memtioned possible call trace.

We also found a benign data race at lwan_job_thread_shutdown and job_thread, which may be intentional.

zfs-fuse (this, no response). Its report is at Reports/zfs-fuse-umem_test.bc-bug.txt

Hi all,

Our bug scanner has reported a data race issue at umem.c#L2379 and umem_update_thread.c#L76.

Followings are code snippets.

```
void

umem_reap(void)
{
    ...
    if (umem_ready != UMEM_READY || umem_reaping != UMEM_REAP_DONE ||

        gethrtime() < umem_reap_next)

        return;

    (void) mutex_lock(&umem_update_lock);
```

and

```
static THR_RETURN

THR_API umem_update_thread(void *arg)

{
    ...
    (void) mutex_lock(&umem_update_lock);
    ...

                    case UMEM_REAP_ACTIVE:

                        umem_reap_next = gethrtime() +

                            (hrtime_t)umem_reap_interval * NANOSEC;

                        umem_reaping = UMEM_REAP_DONE;

                        break;
```

Although the two functions are in the same function *umem_reap* and the thread is created after the access of shared variable *umem_reaping*, I found the following call traces: vmem_populate->vmem_reap->umem_reap, vmem_mmap_alloc->vmem_reap->umem_reap and vmem_xalloc->vmem_reap->umem_reap. These call traces denotes that the function *umem_reap* may call serveral times and result in data race.

Additionally, some parts of the two functions (*umem_reap* and *umem_update_thread*) are protected by `mutex_lock(&umem_update_lock);`, which also indicated that the two lines are may-happens-in-parallel.

RedAlert (this, no response). Its report is at Reports/RedAlert-RAServer-bugs.txt

> Hi,
>
> Our code scanner has reported a data race at `workLoop` method of HeartbeatClient
>
> ```
> bool HeartbeatClient::start() {
>     int err = pthread_create(&_thread, NULL, &HeartbeatClient::workLoop, this);
>     if (err != 0) {
>         LOG(ERROR) << "Cannot start thread for heartbeat client";
>         return false;
>     }
>     _running = true;
>     LOG(INFO) << "Start sending heartbeats to '" << _remoteHost << ":" << _remotePort << "'
>     return true;
> }
> ```
>
> `_running` initialized with `false`,
> Could there is a chance that HeartbeatClient::workLoop access the `_running` before the main thread has assigned `true` into, the heart beat thread might just stopped before the main thread has updated the value of `_running`.
>
> May be we could move the assignment into thread body to avoid this issue.
>
> ```
> void* HeartbeatClient::workLoop(void *arg) {
>    ......
>     // client->_running = true;                 // <======    move here instead
>     while (client->_running) {
> ```