# Detailed Case Analysis of Region Inconsistencies

Zhilei Xu
v-zhixu@microsoft.com

## 1 lklftpd `sess->user` dangling pointer

| | |
|---|---|
| **Type** | Temporary Inconsistency |
| **Infected Application** | lklftpd |

As in Figure 2, a `sess` is allocated in `sess->sess_pool`, and `sess->loop_pool` is a sub region of `sess->sess_pool`. But in Figure 1 we find that `sess->user` can temporarily point to a string in `sess->loop_pool`, which violates consistency. The inconsistency is temporary because the following `init_username_related_fields` call will correct `sess->user` to point to a string duplicated in `sess->sess_pool`.

## 2 lklftpd `sess->data_conn->data_sock` dangling pointer

| | |
|---|---|
| **Type** | Temporary Inconsistency |
| **Infected Application** | lklftpd |

As we can see in Figure 2, `sess->data_conn` is allocated in `sess->sess_pool`. But Figure 3 shows that `sess->data_conn->data_sock` may point to some `sock` allocated in `sess->loop_pool` and thus violates consistency.

When the ftp session is in non-PASV mode, `ftpdataio_get_port_fd()` is called for each `GET`, `STORE` or `LIST` command. To prevent memory leak, `sess->loop_pool` is cleared in each command handling process, thus after this command is processed, the `sess->data_conn->data_sock` becomes dangling pointer. But this dangling pointer is never dereferenced, because next time lklftpd needs a data socket, it'll create a new one. So we classify this inconsistency into temporary type.

## 3 diff `position->node` dangling pointer

| | |
|---|---|
| **Type** | Global Inconsistency |
| **Infected Application** | diff, diff3, diff4 |

As in Figure 4, `position->node` in `pool` points to a tree node that was allocated in `tree->pool` by `svn_diff__tree_insert_token()`.

```
// worker.c:get_username_password(sess)
    if(lfd_cmdio_cmd_equals(sess, "USER"))
    {
        user_ok = handle_user_cmd(sess);
    }
......
init_username_related_fields(sess);

// cmdhandler.c:handle_user_cmd(sess)
sess->user = apr_pstrdup(sess->loop_pool, sess->ftp_arg_str);

// worker.c:init_username_related_fields(sess)
sess->user = apr_pstrdup(sess->sess_pool, sess->user);
```

Figure 1: Code for initializing `sess->user`. At first `sess->user` points to a string in `sess->loop_pool`, and the consistency has been violated. But at last the invoking of `init_username_related_fields()` will make `sess->user` point to a string in `sess->sess_pool`, which is OK.

```
// sess.c:lfd_sess_create(plfd_sess, thd, sock)
// *plfd_sess passes the newly-created session out
sess_pool = apr_thread_pool_get(thd);
rc = apr_pool_create(&loop_pool, sess_pool);
......
*plfd_sess = sess = apr_pcalloc(sess_pool, sizeof(struct lfd_sess));
sess->sess_pool = sess_pool;
sess->loop_pool = loop_pool;
......
sess->data_conn = apr_pcalloc(sess_pool, sizeof(struct lfd_data_sess));
```

Figure 2: Code for creating a session. `sess` is allocated in the thread-specific global pool, and this pool is refered by `sess->sess_pool`. `sess->loop_pool` is for allocating per-command data, and it's a sub region of `sess->sess_pool`. `sess->data_conn` is allocated in `sess->sess_pool`.

```
// connection.c:ftpdataio_get_port_fd(sess, psock)
rc = get_bound_and_connected_ftp_port_sock(sess, &remote_fd);
......
init_data_sock_params(sess, remote_fd);
// get_bound_and_connected_ftp_port_sock allocate remote_fd in sess->loop_pool
// and init_data_sock_params make sess->data_conn->data_sock point to remote_fd

// connection.c:get_bound_and_connected_ftp_port_sock(sess, psock)
// *psock passes the newly-created sock out
rc = apr_socket_create(&sock, APR_INET, SOCK_STREAM, APR_PROTO_TCP, sess->loop_pool);
......
*psock = sock;
// *psock is allocated in sess->loop_pool

// connection.c:init_data_sock_params(sess, sock_fd)
sess->data_conn->data_sock = sock_fd;
// sess->data_conn->data_sock (which is in sess->sess_pool) points to sock_fd
```

Figure 3: `ftpdataio_get_port_fd()` causes `sess->data_conn->data_sock` (in `sess->sess_pool`) point to a `sock` newly-created in `sess->loop_pool`, which violates consistency.

```
// token.c:svn_diff__get_tokens(position_list, tree, diff_baton, vtable, datasource, pool)
// pool is the region for allocating position
    SVN_ERR(svn_diff__tree_insert_token(&node, tree,
                                        diff_baton, vtable,
                                        hash, token));
    position = apr_palloc(pool, sizeof(svn_diff__position_t));
    position->next = NULL;
    position->node = node;
// position is allocated in pool, and position->node accesses node

// token.c:svn_diff__tree_insert_token(node, tree, diff_baton, vtable, hash, token)
// *node passes the newly-create node out
new_node = apr_palloc(tree->pool, sizeof(*new_node));
*node = *node_ref = new_node;
// node is allocated in tree->pool
```

Figure 4: Code for creating `position` and `node`. `position` is in pool, `node` is in `tree->pool`, and `position` accesses `node`.

```
// diff.c:svn_diff_diff(diff, diff_baton, vtable, pool)
subpool = svn_pool_create(pool);
treepool = svn_pool_create(pool);
// subpool and treepool are siblings

svn_diff__tree_create(&tree, treepool);
// pool for tree is treepool

SVN_ERR(svn_diff__get_tokens(&position_list[0],
                             tree,
                             diff_baton, vtable,
                             svn_diff_datasource_original,
                             subpool));
// pool for position is subpool
SVN_ERR(svn_diff__get_tokens(&position_list[1],
                             tree,
                             diff_baton, vtable,
                             svn_diff_datasource_modified,
                             subpool));
......
svn_pool_destroy(treepool);
......
svn_pool_destroy(subpool);


// token.c:svn_diff__tree_create(tree, pool)
*tree = apr_pcalloc(pool, sizeof(**tree));
(*tree)->pool = pool;
// tree->pool is the treepool in svn_diff_diff()
```

Figure 5: Main code of `diff`, and creation of `tree`. Region for holding `position` is `subpool`, and for holding `node` is `treepool`, where `subpool` and `treepool` are sibling regions.

But in Figure 5 we see that `position` is in `subpool` and `node` is in `treepool`, and these are two sibling regions. In fact `treepool` lives shorter than `subpool`. So after `treepool` is destroyed, `position->node` becomes dangling pointer.

The problem does not lead to crash because `position->node` is not used *as a pointer* after `treepool` is destroyed. In fact it *is* used (in `svn_diff__lcs()`), but not *as an integer type* instead of `pointer type`, so it's not dereferenced. The programmer seemed to make use of `position->node` in this way to save memory space, but it's error-prone anyway.

```
// log.c:run_log(adm_access, rerun, diff3_cmd, pool)
struct log_runner *loggy = apr_pcalloc(pool, sizeof(*loggy));
parser = svn_xml_make_parser(loggy, start_handler, NULL, NULL, pool);
......
loggy->parser = parser;
svn_xml_free_parser(parser);

// xml.c:svn_xml_make_parser(baton, start_handler, end_handler, data_handler, pool)
/* ### we probably don't want this pool; or at least we should pass it
   ### to the callbacks and clear it periodically.  */
subpool = svn_pool_create(pool);
svn_parser = apr_pcalloc(subpool, sizeof(*svn_parser));
```

Figure 6: `loggy` is in pool while `loggy->parser` points to a xml parser created from `subpool`, a sub region of `pool`.

# 4 svn `loggy->parser` dangling pointer

| **Type** | Permanent Inconsistency |
|---|---|
| **Infected Application** | svn |

As we can see in Figure 6, the `loggy` in `pool` access a parser in `subpool`, which is a subregion of `pool`. `loggy` lives longer than `parser`, so after `svn_xml_free_parser()` has been called, `loggy->parser` becomes dangling pointer.

The code authors do realize of this problem, and they've mentioned it in the comment (see the "###" lines).

# 5 svn `opt->x_value` dangling pointer

| **Type** | Temporary then Global Inconsistency |
|---|---|
| **Infected Application** | svn |

`make_string_from_option()` and `expand_option_value()` are two mutually-recursive functions, and `make_string_from_option()` is the function provided for end-user, with `expand_option_value()` as its helper function.

As we can see in Figure 7, the last parameter to these two functions, named `x_pool`, is usually obtained from the end-user (such as `svn_config_get()`) as `NULL`, and the upmost call to `make_string_from_option()` set it to `tmp_pool`, a newly-created temporary sub region of `cfg->x_pool`. Then the `tmp_pool` is passed down as the `x_pool` parameter to every call of `expand_option_value()` and `make_string_from_option()`. Thus `opt->x_value` first access a string in `tmp_pool` (a subregion of `cfg->x_pool`, then finally access a string in `cfg->x_pool`.

But as we can see in Figure 8, `opt` resides in `cfg->pool`, which is a parent region of `cfg->x_pool`. So `opt->x_value` accesing a string from `tmp_pool` (sub-sub region of `cfg->pool`) and `cfg->x_pool` (sub region of `cfg->pool`) both

```
// config.c:svn_config_get(cfg, valuep, section, option, default_value)
    make_string_from_option(valuep, cfg, sec, opt, NULL);

// config.c:make_string_from_option(valuep, cfg, section, opt, x_pool)
// *valuep passes the created (and manipulated) string out
if (!opt->expanded)
{
  apr_pool_t *tmp_pool = (x_pool ? x_pool : svn_pool_create(cfg->x_pool));
  expand_option_value(cfg, section, opt->value, &opt->x_value, tmp_pool);
// calling expand_option_value make opt->x_value point to a string in tmp_pool

  opt->expanded = TRUE;
  if (!x_pool)
  {
    if (opt->x_value)
      opt->x_value = apr_pstrmemdup(cfg->x_pool, opt->x_value,
                                    strlen(opt->x_value));
      // the string in tmp_pool is duplicated in cfg->x_pool
      // then opt->x_value points to a string in cfg->x_pool
      svn_pool_destroy(tmp_pool);
  }
}
if (opt->x_value)
  *valuep = opt->x_value;
else
  *valuep = opt->value;


// config.c:expand_option_value(cfg, section, opt_value, opt_x_valuep, x_pool)
// opt_x_valuep passes the manipulated string out
......
    make_string_from_option(&cstring, cfg, section, x_opt, x_pool);
    len = name_start - FMT_START_LEN - copy_from;
    if (buf == NULL)
    {
      buf = svn_stringbuf_ncreate(copy_from, len, x_pool);
      cfg->x_values = TRUE;
    }
    else
      svn_stringbuf_appendbytes(buf, copy_from, len);
    // string is allocated and appended in the x_pool, which is exactly the tmp_pool
......
if (buf != NULL)
{
  svn_stringbuf_appendcstr(buf, copy_from);
  *opt_x_valuep = buf->data;
  // the string in buf (which is in x_pool) is passed out
}
```

Figure 7: A complex process of option string manipulation. `opt->x_value` points to a string in `tmp_pool`, a sub region of `cfg->x_pool`, then to a string in `cfg->x_pool`

```
// config.c:svn_config_set(cfg, section, option, value)
opt = apr_palloc(cfg->pool, sizeof(*opt));
......
opt->x_value = NULL;
// opt is allocated in cfg->pool

// config.c:svn_config_read(cfgp, file, must_exist, pool)
// *cfgp passes the newly-created cfg out
svn_config_t *cfg = apr_palloc(pool, sizeof(*cfg));
......
cfg->pool = pool;
cfg->x_pool = svn_pool_create(pool);
// cfg->x_pool is a subregion of cfg->pool
```

Figure 8: `opt` is allocated in `cfg->pool`, and `cfg->x_pool` is a subregion of `cfg->pool`

violates consistency, and the formal one is temporary, while the latter global. Note that `svn` doesn't delete `cfg->pool` or `cfg->x_pool` at all, so we consider them both global region.

# 6 svn hash iterator `hi->ht` dangling pointer and memory leak

| **Type** | Permanent Inconsistency |
|---|---|
| **Infected Application** | svn |

As we can see in Figure 9, the iterator `hi` is created in `pool`, but it can access `ht`, which is created in `subpool`, a subregion of `pool`. `subpool` is deleted before `pool`, then `hi->ht` becomes dangling pointer.

This usage of hash table and its iterator is controversial: Anyway, an iterator is useful only if its associating hash table is valid. If the `hi` is used after `ht` is destroyed, the dangling pointer may cause a *crash*; if it is not used, then the memory space it occupies cannot be reclaimed as the user does to `ht`, thus leads to a potential *memory leak*. This usage is even *dangerous* because the end-user may think that all the memory occupied by things related to a hash table is destroyed with the deallocation of the hash table itself, thus put the iterator allocation/using in some unbounded loop (even an *infinite* event serving loop), that will finally consume all the memory. A better usage of iterator is to put it in a subregion of the region that holds hash table.

```
// xml.c:svn_xml_make_open_tag_v(str, pool, style, tagname, ap)
  apr_pool_t *subpool = svn_pool_create(pool);
  apr_hash_t *ht = svn_xml_ap_to_hash(ap, subpool);
// ht is created in subpool

  svn_xml_make_open_tag_hash(str, pool, style, tagname, ht);
  svn_pool_destroy(subpool);

// xml.c:svn_xml_make_open_tag_hash(str, pool, style, tagname, attributes)
for (hi = apr_hash_first(pool, attributes); hi; hi = apr_hash_next(hi))
{
  ......
}
// hi is created in pool

// apr_hash.c:apr_hash_first(pool, ht)
if (p)
    hi = apr_palloc(p, sizeof(*hi));
else
    hi = &ht->iterator;

hi->ht = ht;
// hi accesses ht
```

Figure 9: `hi` is created in `pool`, while `hi->ht` points to `ht`, which is allocated in `subpool`.