The problematic source codes

In short, a child thread accessing memory that's been freed by main thread causes heap-use-after-free.
I'm going to present what main thread and the child thread do respectively.

What does main threads do?

See figure below, it's a section of *init_server_components* and the main thread

1. calls *plugin_register_dynamic_and_init_all* to initialize plugins and child thread is to be created. We'll see what the child thread does in next section.
2. calls *disable_resource_group* if thread_handling != one-thread-per-connection.

```
5619   if (plugin_register_dynamic_and_init_all(&remaining_argc, remaining_argv,
5620                                            flags)) {
5621     // Delete all DD tables in case of error in initializing plugins.
5622     if (dd::upgrade_57::in_progress())
5623       (void)dd::init(dd::enum_dd_init_type::DD_DELETE);
5624
5625     if (!opt_validate_config)
5626       LogErr(ERROR_LEVEL, ER_CANT_INITIALIZE_DYNAMIC_PLUGINS);
5627     unireg_abort(MYSQLD_ABORT_EXIT);
5628   }
5629   dynamic_plugins_are_initialized =
5630       true; /* Don't separate from init function */
5631
5632   LEX_CSTRING plugin_name = {STRING_WITH_LEN("thread_pool")};
5633   if (Connection_handler_manager::thread_handling !=
5634           Connection_handler_manager::SCHEDULER_ONE_THREAD_PER_CONNECTION ||
5635       plugin_is_ready(plugin_name, MYSQL_DAEMON_PLUGIN)) {
5636     auto res_grp_mgr = resourcegroups::Resource_group_mgr::instance();
5637     res_grp_mgr->disable_resource_group();
5638     res_grp_mgr->set_unsupport_reason("Thread pool plugin enabled");
5639   }
 "sql/mysqld.cc" 11295 lines --49%--
```

create child threads,
see asan-8.0.20.txt for call stack

precondition : thread_handling != one-thread-per-connection

to free memory

What's in *disable_resource_group*?

See the following 3 figures.
The main thread calls *disable_resource_group* to release *m_resource_group_hash*. *m_sys_default_resource_group* and *m_usr_default_resource_group* are freed as they are owned by *m_resource_group_hash*.

```
123   /**
124     Disable and deinitialize the resource group if it was initialized before.
125   */
126
127   void disable_resource_group() {
128     if (m_resource_group_support) {
129       LogErr(INFORMATION_LEVEL, ER_RES_GRP_FEATURE_NOT_AVAILABLE);
130       deinit();
131       m_resource_group_support = false;
132     }
133   }
"sql/resourcegroups/resource_group_mgr.h" 462 lines --20%--
```

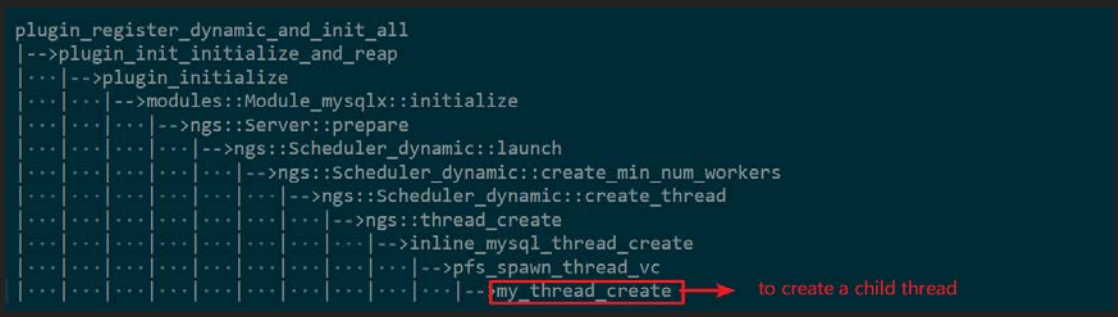
```
267 void Resource_group_mgr::deinit() {
268   if (m_resource_group_support && m_notify_svc != nullptr) {
269     m_notify_svc->unregister_notification(m_notify_handle);
270     m_registry_svc->release(m_h_res_grp_svc);
271     m_h_res_grp_svc = nullptr;
272
273     m_registry_svc->release(m_h_notification_svc);
274     m_h_notification_svc = nullptr;
275
276     mysql_plugin_registry_release(m_registry_svc);
277     delete m_resource_group_hash;
278     mysql_rwlock_destroy(&m_map_rwlock);
279   }          m_sys_default_resource_group is released
280 }
"sql/resourcegroups/resource_group_mgr.cc" 602 lines --40%--
```

```
400   /**
401     Pointer to USR default and SYS default resource groups.
402     Owernship of these pointers is resource group hash.
403   */
404   Resource_group *m_usr_default_resource_group;
405   Resource_group *m_sys_default_resource_group;
406
407   /**
408     Map mapping resource group name with it's corresponding in-memory
409     Resource_group object
410   */
411   collation_unordered_map<std::string, std::unique_ptr<Resource_group>>
412       *m_resource_group_hash;
"sql/resourcegroups/resource_group_mgr.h" 462 lines --81%--
```

So far, we know that the main thread releases m_sys_default_resource_group. We are going to see how the child thread could access the released memory.

What does the child thread do?

Following the preceding section, we know the main thread calls *plugin_register_dynamic_and_init_all* and child thread is to be created.
The figures below are simplified call stacks and you can refer asan-8.0.20.txt for details.

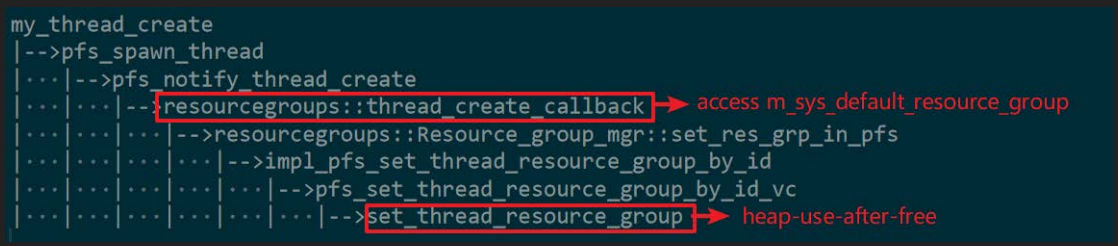


plugin_register_dynamic_and_init_all makes preparations and calls *my_thread_create* to create a child thread factually. Let's see what's in my_thread_create.

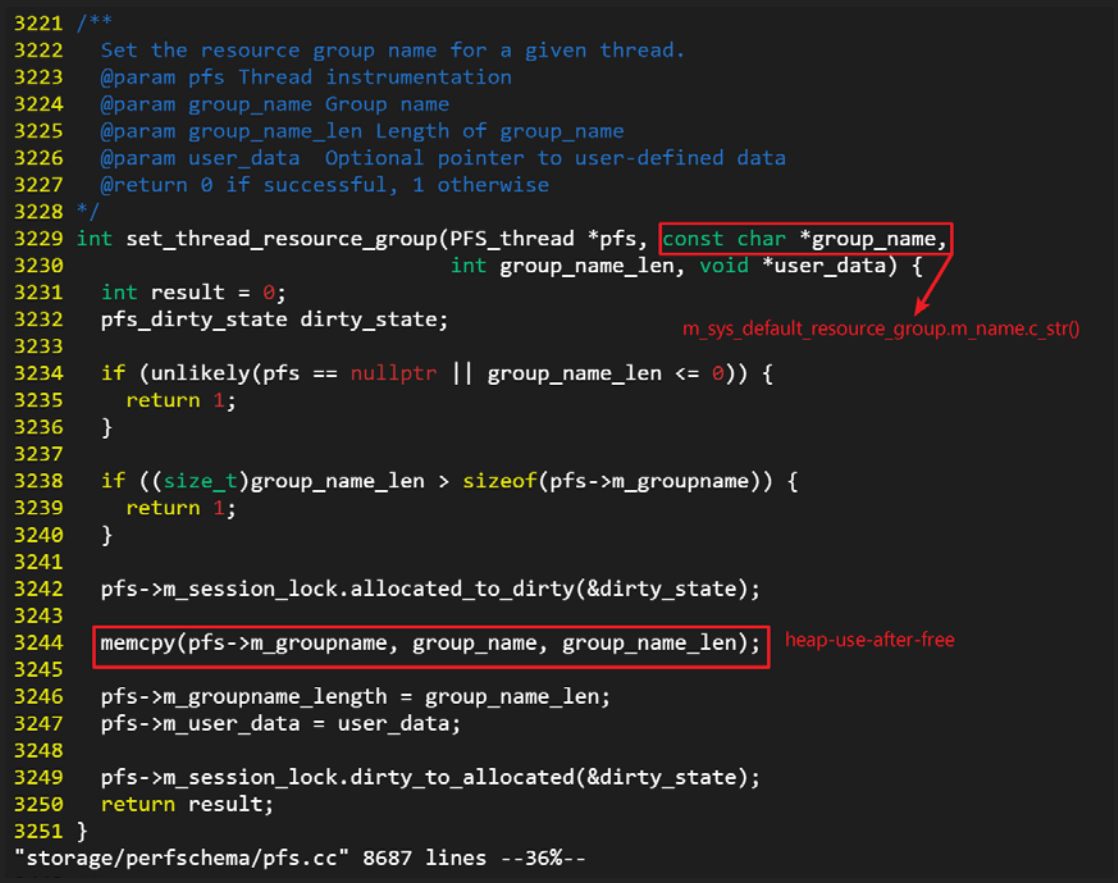
What's in my_thread_create?

See figures below.

my_thread_create calls *pfs_notify_thread_create* so that PFS knows the newly created thread. *resourcegroups::thread_create_callback* gets a reference to *m_sys_default_resource_group.m_name* and passes it to *set_thread_resource_group*. *set_thread_resource_group* copies from the memory pointed by *m_sys_default_resource_group.m_name.c_str()*, which could've been freed by the main thread in function *Resource_group_mgr::deinit*.




```
65 namespace resourcegroups {
66 Resource_group_mgr *Resource_group_mgr::m_instance = nullptr;
67
68 void thread_create_callback(const PSI_thread_attrs *thread_attrs) {
69   auto res_grp_mgr = resourcegroups::Resource_group_mgr::instance();
70
71   if (!res_grp_mgr->resource_group_support()) return;
72
73   if (thread_attrs != nullptr) {
74     auto res_grp = thread_attrs->m_system_thread
75                        ? res_grp_mgr->sys_default_resource_group()
76                        : res_grp_mgr->usr_default_resource_group();
77     res_grp_mgr->set_res_grp_in_pfs(res_grp->name().c_str(),
78                                     res_grp->name().length(),
79                                     thread_attrs->m_thread_internal_id);
80   }
81 }
'sql/resourcegroups/resource_group_mgr.cc" 602 lines --9%--
```



Thread concurrency

Main thread releasing m_resource_group_hash and child thread accessing m_sys_default_resource_group are not controlled.

Heap-use-after-free takes place if main thread and child thread proceed in such an order:

1. [main thread] *plugin_register_dynamic_and_init_all*. Child thread is created.
2. [child thread] *thread_create_callback*. A reference to *m_sys_default_resource_group.m_name.c_str()* is acquired and passed down.
3. [main thread] *Resource_group_mgr::deinit*. *m_resource_group_hash* is released.
4. [child thread] *set_thread_resource_group*. Tries to memcpy from *m_sys_default_resource_group.m_name.c_str()* that has been released.

Why mtr on original code works well

1. One precondition of this problem is that thread_handling isn't one-thread-per-connection. But it is one-thread-per-connection mostly in mtr, because mtr config files, like `mysql-test/include/default_mysqld.cnf`, don't assign thread_handling explicitly, then it's as the default value defined in sql/sys_vars.cc.

```
3575 static const char *thread_handling_names[] = {
3576     "one-thread-per-connection", "no-threads", "loaded-dynamically", nullptr};
3577 static Sys_var_enum Sys_thread_handling(
3578     "thread_handling",
3579     "Define threads usage for handling queries, one of "
3580     "one-thread-per-connection, no-threads, loaded-dynamically",
3581     READ_ONLY GLOBAL_VAR(Connection_handler_manager::thread_handling),
3582     CMD_LINE(REQUIRED_ARG), thread_handling_names, DEFAULT(0));   → default : one-thread-per-connection
3583
"sql/sys_vars.cc" 6850 lines --51%--
```

(the reason why thread_handling can't be one-thread-per-connection has been given above.)

2. CPU and scheduling matters. Considering that thread_handling is not one-thread-per-connection, if child threads have always finished memcpy before main thread frees the memory, this problem would not happen. It's totally up to CPU performance and thread scheduling, and it's difficult to control CPU and scheduling manually. This is what makes the problem hard to reproduce.
3. Is ASAN on?

Why repeat-8.0.20.patch is reasonable

1. repeat-8.0.20.patch slows down the child thread, so that the main thread calls *Resource_group_mgr::deinit* before the child thread calls *set_thread_resource_group*, which makes the heap-use-after-free observable.
2. If there is a concurrency control, no matter how much child thread is slowed down, heap-use-after-free should never happen.

Why fix-8.0.20.patch works

fix-8.0.20.patch doesn't use lock or anything like that to do the exclusion. Instead, it avoids the concurrency problem.

m_sys_default_resource_group.m_name is a constant "SYS_default" and *m_usr_default_resource_group.m_name* is a constant "USR_default". So *get_sys_default_resource_group_name()* and *get_usr_default_resource_group_name()*, returning "SYS_default" and "USR_default" directly and respectively, are introduced.