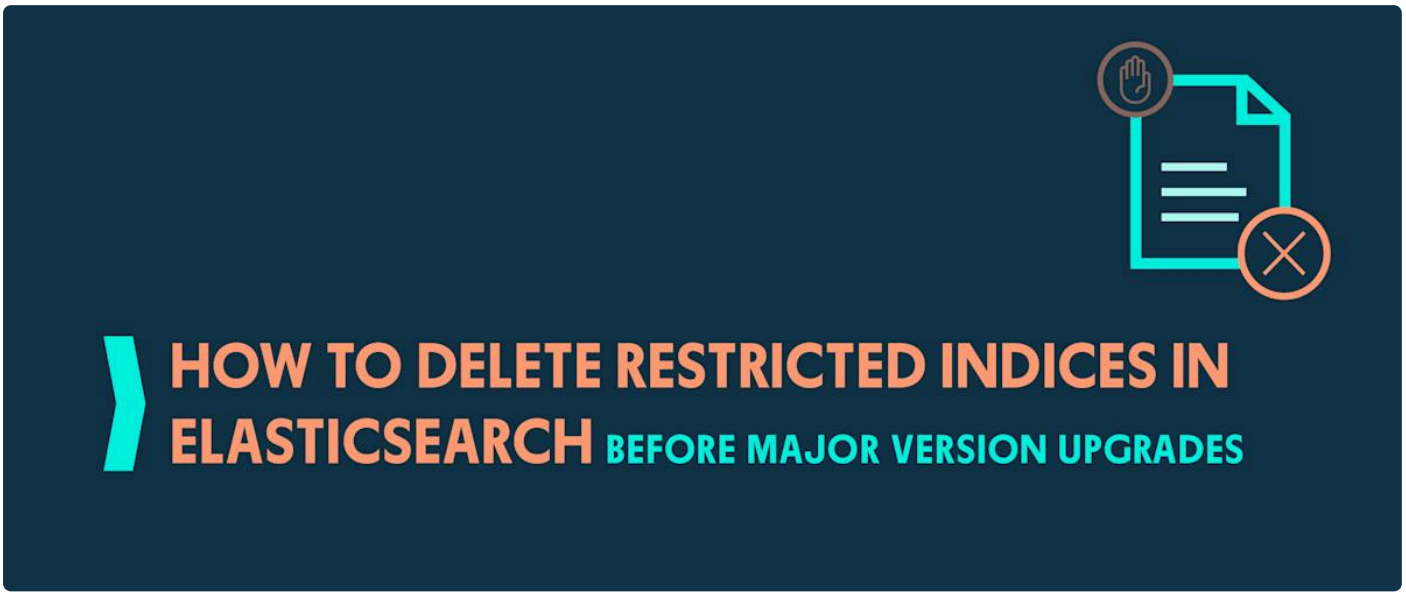
How to Delete Restricted Indices in Elasticsearch Before Major Version Upgrades

Jochen Kressin · 2026-10-01



Major version upgrades in Elasticsearch can surface unexpected compatibility issues, particularly when dealing with indices created in older versions. One common scenario you might face is the need to clean up legacy system indices before upgrading, especially when these indices are protected as "restricted indices" that require special permissions to delete.

In this article, we'll walk through a real-world scenario: upgrading from Elasticsearch 8.18.0 to 9.0.1 while dealing with old `.reporting-2023*` indices that were created with Elasticsearch 7.x compatibility. You'll learn why standard delete operations fail on these indices and how to properly configure permissions to remove them.

Problem Description

Imagine you're preparing to upgrade your Elasticsearch cluster from version 8.18.0 to 9.0.1. During the pre-upgrade check, you discover that several indices have a compatibility version of `7090199` (Elasticsearch 7.x), which prevents the cluster from starting on version 9.

You identify the culprits: several `.reporting-2023*` indices that need to be deleted before the upgrade can proceed. When you attempt to delete these indices using a standard API key, even one with seemingly comprehensive permissions, you encounter this error:

© 2026 floragunn GmbH · search-guard.com

```
{
  "error": {
    "root_cause": [{
      "type": "security_exception",
      "reason": "action [indices:admin/delete] is unauthorized for API key id [<hidden>] of user
[elastic] on restricted indices [.reporting-2023<hidden>], this action is granted by the index
privileges [delete_index,manage,all]"
    }],
    "type": "security_exception",
    "reason": "action [indices:admin/delete] is unauthorized for API key id [<hidden>] of user
[elastic] on restricted indices [.reporting-2023<hidden>], this action is granted by the index
privileges [delete_index,manage,all]"
  },
  "status": 403
}
```

The frustrating part? You've already created an API key with what appears to be full permissions, including `"allow_restricted_indices": true` in the role definition. Yet Elasticsearch still refuses to delete the indices.

Root Cause Analysis

To understand what's happening here, we need to explore how Elasticsearch handles system indices, restricted indices, and how API key permissions are actually scoped.

What Are Restricted Indices?

Restricted indices are special system indices that Elasticsearch uses internally. These include indices such as:

- `.kibana*` (Kibana system indices)
- `.reporting*` (Kibana reporting indices)
- `.security*` (security configuration)
- `.tasks*` (task management)

Elasticsearch protects these indices because they contain critical system data. Accidentally modifying or deleting them could break your cluster's functionality. By default, no generic role grants access to them, and even the `superuser` role does not have unrestricted write access to restricted indices.

The Permission Puzzle

The `allow_restricted_indices` flag is not the problem here—and it's important to be precise about what it does. When you set `"allow_restricted_indices": true` on an index privilege entry, the name patterns in that entry are allowed to cover restricted indices as well. Combined with a privilege that includes the delete action (`delete_index`, `manage`, or `all`), that genuinely does grant the ability to delete restricted indices.

The catch is how API keys derive their permissions:

© 2026 floragunn GmbH · search-guard.com

An API key's effective permissions are the intersection of the role descriptors you assign to the key and the privileges of the user who created it, captured at creation time. An API key can never grant more than its owner actually has.

This is the missing piece. You created the key as the `elastic` user, which holds the `superuser` role. And since Elasticsearch 8.0, the `superuser` role was deliberately changed so that it no longer has write access—including delete—to restricted system indices. A superuser can still read and monitor them, but nothing more.

So the chain looks like this: your role descriptor asks for `delete_index` on restricted indices, but the key's owner (`elastic` / `superuser`) has no delete access to restricted indices to begin with. The intersection of "what you requested" and "what the owner can actually do" is empty for that action. No matter how generous the role descriptor looks, the key inherits the owner's blind spot and you get the 403.

Why a Directly Authenticated User Works

The fix is to stop relying on an API key owned by a superuser and instead authenticate directly as a user whose role genuinely carries the delete privilege over the restricted indices.

This works because direct user authentication isn't subject to the API-key intersection rule, the user's role privileges apply as written. So a user assigned a role that grants `delete_index` with `"allow_restricted_indices": true` on `.reporting*` can delete those indices, full stop. That role can be a minimal custom role you create for the task, or a built-in role such as `kibana_system` that already happens to include the right grants over `.kibana*` and `.reporting*`. Both rely on exactly the same mechanism; the custom role just gives you control over the scope.

Solution

Here's how to successfully delete restricted system indices before your upgrade. We'll use a minimal, purpose-built role (the recommended approach), and note the `kibana_system` shortcut afterward.

Step 1: Create a Minimal Role and a Dedicated User

Instead of using an API key, create a role that grants delete access to only the indices you need, then assign it to a dedicated user. You can do this through the Kibana UI or the REST API.

First, the role—scoped tightly to the reporting indices and explicitly opting into restricted indices:

```
POST /_security/role/restricted_index_admin
{
  "indices": [
    {
      "names": [ ".reporting-2023*" ],
      "privileges": [ "monitor", "read", "view_index_metadata", "delete_index" ],
      "allow_restricted_indices": true
    }
  ]
}
```

Then a user that holds this role:

```
POST /_security/user/maintenance_user
{
  "password" : "your_secure_password",
  "roles" : [ "restricted_index_admin" ],
  "full_name" : "Maintenance User",
  "email" : "maintenance@example.com"
}
```

Shortcut: If you'd rather not define a role, assigning the built-in `kibana_system` role to the user also works, since it already carries the necessary privileges over `.kibana*` and `.reporting*`. Be aware that it grants far more than deleting a few reporting indices requires, so the minimal custom role above is the better fit for the principle of least privilege.

Step 2: Verify the User's Permissions

Confirm that the user has the role you assigned:

```
GET /_security/user/maintenance_user
```

The response should show:

```
{
  "maintenance_user" : {
    "username" : "maintenance_user",
    "roles" : [ "restricted_index_admin" ],
    "full_name" : "Maintenance User",
    "email" : "maintenance@example.com",
    "enabled" : true
  }
}
```

Step 3: Delete the Restricted Indices

Now authenticate as this user (not via an API key owned by `elastic`) and delete the problematic indices. First, identify all the reporting indices you need to remove:

```
GET /_cat/indices/.reporting-2023*?v
```

Then delete them one by one or using a wildcard pattern:

```
DELETE /.reporting-2023.01.01
```

Or delete multiple indices at once:

```
DELETE /.reporting-2023*
```

You should now receive a successful response:

© 2026 floragunn GmbH · search-guard.com

```
{
  "acknowledged" : true
}
```

Step 4: Clean Up

After you've deleted all the necessary indices and completed your upgrade, remember to either disable or remove the maintenance user (and the role) if you no longer need them:

```
PUT /_security/user/maintenance_user/_disable
```

Or delete the user entirely:

```
DELETE /_security/user/maintenance_user
DELETE /_security/role/restricted_index_admin
```

Understanding Alternative Approaches

While the dedicated-user approach works well for this scenario with `.reporting*` indices, it's worth understanding what doesn't work and why.

Why the API Key Fell Short

Creating an API key with `"allow_restricted_indices": true` failed not because the flag is wrong, but because of how API keys are scoped:

- An API key's permissions are the intersection of its role descriptors and the creating user's privileges.
- The key was created by `elastic`, which holds `superuser`—and since 8.0 the `superuser` role has no delete access to restricted indices.
- Intersecting "delete on restricted indices" (requested) with "no delete on restricted indices" (what the owner actually has) yields nothing, so the action is denied.

In short, the limitation isn't API keys in general, it's that a key can't out-privilege its owner. Direct user authentication with a role that genuinely holds the privilege sidesteps this entirely.

The Role of the Superuser

You might wonder why the `elastic` superuser can't just delete these indices directly. Since Elasticsearch 8.0, the `superuser` role was deliberately changed to remove write, delete, and close access to restricted system indices; superusers retain only read and monitor access to them. This is a security feature designed to prevent accidental data loss and to ensure no out-of-the-box role can tamper with critical system indices.

That's why even authenticating directly as `elastic` won't delete a `.reporting*` index, you need a role that explicitly grants the delete action with `allow_restricted_indices` enabled, which is exactly what the maintenance role above provides.

© 2026 floragunn GmbH · search-guard.com

Takeaways

1. Plan for system index cleanup during major upgrades Before performing major version upgrades (like moving from Elasticsearch 8.x to 9.x), identify indices created with older compatibility versions. Use the `GET /_cat/indices?v` command to inspect your indices and combine it with index settings checks when you need the exact creation version.

2. Understand how API key permissions are scoped API keys are convenient for application-level access, but their effective permissions are the intersection of the key's role descriptors and the creating user's privileges. For restricted indices this matters a lot: a key created by a superuser inherits the superuser's lack of restricted-index write access, so no role descriptor can grant it delete rights. For administrative tasks on restricted indices, use direct user authentication instead.

3. Grant restricted-index access explicitly—and minimally The cleanest way to manage restricted indices is a custom role that grants the specific action (e.g., `delete_index`) with `"allow_restricted_indices": true` on the exact index pattern you need, assigned to a dedicated user. Built-in roles like `kibana_system` also work, but they grant far more than the task requires. Prefer a minimal role for better security and a clearer audit trail.

4. Create temporary maintenance users for sensitive operations When you need to perform a one-time administrative task like deleting restricted indices, create a dedicated user with the minimal necessary permissions. This approach provides better audit trails and allows you to easily revoke access after the task is complete.

5. Document your upgrade procedures Maintain clear documentation of any special steps required for your Elasticsearch upgrades, including handling legacy indices and permission requirements. This knowledge becomes invaluable when planning future upgrades or troubleshooting similar issues.

Need help securing your Elasticsearch cluster during upgrades and beyond? Search Guard provides enterprise-grade security features that give you fine-grained control over your Elasticsearch environment. Visit search-guard.com to learn how we can help you manage complex security scenarios with confidence.

This article was published on the Search Guard blog: https://search-guard.com/blog/delete-restricted-indices-elasticsearch-upgrade/
Search Guard is Security and Alerting for Elasticsearch and Kibana — search-guard.com · Start a free trial