CBB

LIVE ANALYSIS OCTOBER 29, 2026

TorchServe LPE: a fixed temp path, and a model server that trusts it

A predictable extraction directory in /tmp lets a local user swap the model handler and run code as the TorchServe service account. No race, no root, just a parent directory and rename(2).

#Security Research #Local Privilege Escalation #TorchServe #PyTorch #CVE #Linux

When /tmp Decides Who Runs Your Model

> Privilege escalation isn't always a kernel bug.
> Sometimes it's just a directory someone else got to first.



TorchServe loads a model by unpacking a `.mar` archive and importing a Python file out of it. That file, the handler, is code the model author wrote, and the worker runs it. This is by design: a handler is the glue around a model, so running it is the whole point.

The design assumes one thing that the code never enforces: that the only person who can put a handler on disk is the operator who deployed the model. On a default install this assumption breaks. The archive is extracted into a fixed, predictable, world-writable path, and a local user who gets there first decides which `handler.py` the worker imports. The worker runs it as the service account. That is the bug.

This affects TorchServe `0.12.0`, commit `62c4d6a1`, the last release. The project is archived and unmaintained. CVE assignment is pending, coordinated through VulnCheck.

Where the archive lands

A `.mar` is a zip. When a model is registered, the frontend unpacks it into a working directory under the JVM temp directory, `java.io.tmpdir`. The extracted tree is `<java.io.tmpdir>/models/<uuid>/`, and inside it sits `MANIFEST.json` plus the handler and the weights.

So the whole thing hinges on the value of `java.io.tmpdir`. Two pieces of code decide it.

The Python launcher builds the `java` command line. It only passes `-Djava.io.tmpdir` when the environment variable `TEMP` is set:

```
# ts/model_server.py:88 (abridged)
if "TEMP" in os.environ:
    # -Djava.io.tmpdir=$TEMP gets appended here
```

A `pip install torchserve` and the usual `systemctl` unit never set `TEMP`. Nothing is appended, and the JVM falls back to its own default, which on Linux is `/tmp`. So the extraction root is, literally, `/tmp/models`.

The second piece is the extraction itself. `ZipUtils` unpacks the archive and calls `forceMkdir` on the destination. `forceMkdir` creates the directory if it is missing and returns quietly if it already exists. It does not check who owns it or what mode it carries. If `/tmp/models` is already there, TorchServe extracts straight into it.

`/tmp` is world-writable with the sticky bit, mode `1777`. Any user can create entries in it. `/tmp/models` is not a name TorchServe reserves or protects. It is a name the first process to ask for it gets to own.

The sticky bit does not save you

The reflex objection here is the sticky bit. `/tmp` is `1777`, so a user cannot rename or delete another user's entries inside `/tmp`. True, and irrelevant, because the directory under attack is not `/tmp`. It is `/tmp/models`.

The sticky bit restricts renames and deletions of entries *inside* the sticky directory. It is a property of the parent, not an inherited flag. When an unprivileged user creates `/tmp/models`, that user owns it, and it is a plain `0777` directory with no sticky bit unless they set one. Everything the service account later writes *inside* `/tmp/models` lives in a directory the attacker owns.

And `rename(2)` only needs write and execute permission on the parent directory of the thing being moved. It does not need ownership of the entry itself, and it does not need write permission on that entry. The attacker owns `/tmp/models`, so the attacker can rename anything the service account extracts into it, no matter that those files come out owned by uid 1001.

That is the entire trick. You never write into the model directory. You own its parent, and the parent is not sticky.

```
# attacker, before TorchServe starts
$ mkdir -m 777 /tmp/models
$ ls -ld /tmp /tmp/models
drwxrwxrwt 1 root     root   ... /tmp          ← sticky, owned by root
drwxrwxrwx 1 attacker attacker ... /tmp/models  ← not sticky, owned by attacker
```

Swapping the handler

Once the attacker owns `/tmp/models`, the sequence is mechanical. Watch the directory, and the moment TorchServe extracts a model into it, replace the extracted tree with one the attacker controls before a worker imports the handler.

The steps:

- Create `/tmp/models` (mode `0777`) before the first model load.
- Watch it. When a `<uuid>/` appears, the frontend has just extracted a model there.
- Move the extracted `<uuid>/` aside (`mv <uuid> <uuid>.orig`), recreate `<uuid>/` owned by the attacker, copy back the original `MANIFEST.json` so the manifest still parses, and drop the attacker's `handler.py`.
- The worker imports `/tmp/models/<uuid>/handler.py`. Its top-level code runs as the service account.

```
# Esquisse du mécanisme. TODO: remplacer par ton script notify réel
# (inotifywait + swap) une fois que tu me le passes, pour que le post
# cite exactement ce que tu as exécuté au lab, pas une reconstruction.
inotifywait -m -e create --format '%f' /tmp/models | while read uuid; do
    [ -d "/tmp/models/$uuid" ] || continue
    mv "/tmp/models/$uuid" "/tmp/models/$uuid.orig"
    mkdir "/tmp/models/$uuid"
    cp "/tmp/models/$uuid.orig/MANIFEST.json" "/tmp/models/$uuid/"
    cp handler.py "/tmp/models/$uuid/"
done
```

There is no symlink involved, and that matters. A symlink-swap variant is stopped by `fs.protected_symlinks`, which is on by default on modern distributions. This is a real directory swap, owned by the attacker, so `protected_symlinks` has nothing to act on. That is what makes the finding hold where the obvious symlink version would be filtered out.

There is also no race to win. If the swap misses the first import, TorchServe re-imports the handler on the next worker spawn, so the attacker-controlled `handler.py` is picked up at the following worker cycle. The window reopens on its own.

The proof handler

The published proof does one thing: show that the imported code runs as the service account. No socket, no exec, no shell. It writes the identity of the process importing it and then satisfies the handler interface so the model still loads and nothing looks wrong from the outside.

```
import os

with open("/tmp/ts01-proof.txt", "w") as f:
    f.write("code execution as uid=%d euid=%d\n" % (os.getuid(), os.geteuid()))

from ts.torch_handler.base_handler import BaseHandler

class ModelHandler(BaseHandler):
    pass
```

At the lab this reproduced end to end on `0.12.0` / `62c4d6a1`: the witness file appears owned by the service account, the worker's working directory is the attacker's `/tmp/models/<uuid>/`, and inference keeps serving as if nothing happened because the on-disk `.mar` is untouched. The official Docker image is not affected, because the container's temp handling is not the stock unset-`TEMP` case; this is an install-configuration exposure, not a property of every deployment.

What the service account reaches

The witness proves import, but the real question is what that account can do, and the honest framing is: this is cross-account code execution into the TorchServe service account, uid 1001, not an escalation to root. The severity comes from what that account holds, not from touching the kernel.

Concretely, anything readable by uid 1001 is exposed. TorchServe's own `key_file.json`, which carries the management and inference API tokens when token auth is on, is readable by the account that owns it, so an attacker in this position can drive the management API as the server. Any credentials the service account can read, an `~/.aws/credentials` at `0600` in its home for instance, are readable by definition. If the account has a writable home, `~/.ssh/authorized_keys` is a persistence primitive. And because the import point sits between loading and serving, the attacker

can quietly control inference output while the stored model stays pristine, which is the kind of tampering that does not show up in a file integrity check of the `.mar`.

I am describing reach, not shipping it armed. The proof handler stops at the witness file on purpose.

Severity

CVSS 3.1 `7.8`, vector `AV:L/PR:L/UI:N/S:U/C:H/I:H/A:H`. Local attacker, low privilege, no interaction, full confidentiality, integrity and availability impact on the service account's scope. Not root. Anyone who reads "LPE" and expects a jump to uid 0 should recalibrate: the win is lateral, user to service account, and it is scoped to what uid 1001 can reach.

Preconditions, stated plainly

This does not fire on every TorchServe box, and overstating it would be wrong. Two things have to hold.

First, `TEMP` is unset, so extraction lands in `/tmp/models`. That is the default for a `pip` install and for a plain `systemd` unit written without hardening. A unit with `PrivateTmp=yes` gives the service its own private `/tmp`, and the attack dies there because the attacker's `/tmp/models` is in a different namespace. TorchServe ships no official unit, so whether the unit is hardened is on whoever wrote it.

Second, the attacker has to own `/tmp/models` before TorchServe creates it. There are two realistic ways in. Either the service is started on demand, by an admin, while ordinary users are already logged in on the machine, so an attacker can plant `/tmp/models` ahead of the start. Or a `/tmp` cleaner removes `/tmp/models` out from under a running server and reopens the window at the next extraction. If TorchServe starts at boot, before any unprivileged user is on the box, it creates `/tmp/models` first, the sticky bit on `/tmp` then stops anyone from swapping it, and there is no window at all.

So: default-but-not-universal. Real on shared or interactively started deployments, closed on a boot-first, hardened one.

Remediation

Three independent fixes, any one is enough:

- Set `TEMP` (or pass `-Djava.io.tmpdir`) to a directory the service account owns and that is not world-writable.
- Harden the `systemd` unit with `PrivateTmp=yes`, so `/tmp` is namespaced per service.
- At the code level, extract into a directory owned by the service account and refuse to reuse a pre-existing extraction root that the service does not own, instead of `forceMkdir`-ing into whatever is already there.

The project is archived, so the first two are what operators actually have. The third is what a maintained fork would change.

Disclosure

Reported to Meta Whitehat and to `aws-security@amazon.com` on 2026-07-28, the two channels named in the repository `SECURITY.md`. No response from either.

Worth noting for the CNA question: PyTorch has been under the PyTorch Foundation (Linux Foundation) since 2022, so it is no longer squarely inside Meta's Whitehat scope, which likely explains the silence on that side and is part of why coordination moved to VulnCheck. CVE assignment is pending there.

← Back to archive