Firebase: Google Cloud's Evil Twin

Brandon Evans

Firebase: Google Cloud's Evil Twin

Author: Brandon Evans
Author of SEC510: Multicloud Security Assessment and Defense
October 7, 2020

Abstract

Firebase is the most popular developer tool that security has never heard of. We will bring its numerous flaws to light.

Firebase allows a frontend application to connect directly a backend database. Security wonks might think the previous sentence describes a vulnerability, but this is by design. Released in 2012, Firebase was a revolutionary cloud product that set out to "Make Servers Optional". This should raise countless red flags for all security professionals as the application server traditionally serves as the intermediary between the frontend and backend, handling authentication and authorization. Without it, all users could obtain full access to the database. Firebase attempts to solve this by moving authentication and authorization into the database engine itself. Unfortunately, this approach has several flaws.

Firebase: Google Cloud's Evil Twin

Firebase is the most popular developer tool that security has never heard of. We will bring its numerous flaws to light.

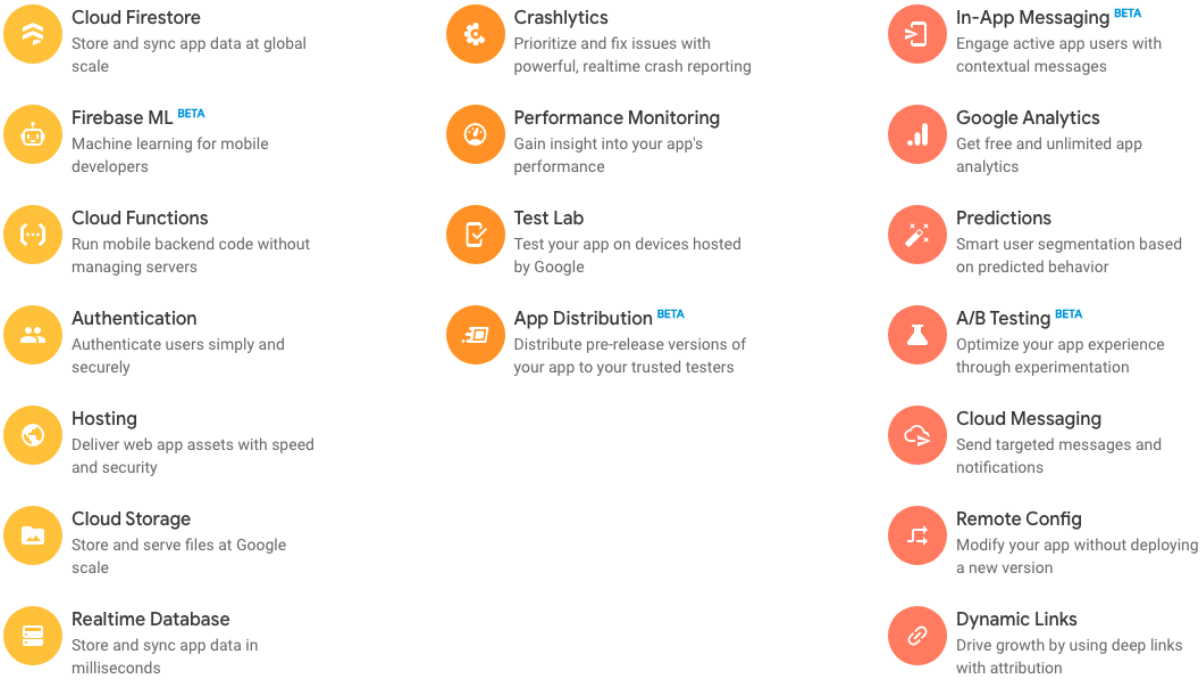
Firebase allows a frontend application to connect directly a backend database. Security wonks might think the previous sentence describes a vulnerability, but this is by design. Released in 2012, Firebase was a revolutionary cloud product with two core capabilities:

1. **Real-time synchronization:** When changes are made to the database, they are reflected instantly within the application's user interface, and vice-versa.
2. **Direct-data access:** According to Firebase's archives, they "Make Servers Optional" because "Data stored in Firebase is directly accessible from Javascript on the client."

The second statement should raise countless red flags for all security professionals. Security orthodoxy states that it is extremely dangerous to eliminate the application server, which serves as the intermediary between the frontend and backend. This intermediary is supposed to handle authentication and authorization; without it, all users could obtain full access to the database. Firebase attempts to solve this by moving authentication and authorization into the database engine itself. Unfortunately, this approach has several flaws, which we will explore later on in this paper.

Security professionals might be surprised (and horrified) to learn that Firebase was and is **extremely popular**. In 2014, two years after launching, Firebase had 100,000 developers on their platform. Firebase's success can be credited to its ease of use, especially for Single-Page Applications (SPAs) and mobile apps. Thanks to its accessibility, Firebase is commonly taught in non-traditional educational environments such as coding bootcamps and workforce accelerators. More experienced engineers might use it to create proof of concept applications with the intent of migrating to other technologies long-term. Of course, developers are busy, and as such, that task will likely only be prioritized when using Firebase is no longer sustainable due to scale.

Firebase quickly got the attention of a technology giant, Google, who acquired them in late 2014. Prior, Firebase expanded their service offering to include static website hosting and authentication. Afterwards, Google acquired web hosting platform Divshot and consolidated it into the Firebase Hosting offering. Initially, it appeared that Google was trying to incorporate all of these novel services into their own cloud platform, the Google Cloud Platform (GCP). However, six years later, not only have Firebase's original three services remained distinct, but Firebase has added 15 services.



Firebase's current service offerings.

Firebase is most accurately described as a fully-fledged cloud platform that is distinct from GCP. At the same time, Firebase is loosely integrated with GCP, and even enabling Firebase in a GCP project can expose your GCP project to risk. When this risk is combined with that posed by Firebase's most popular services, it is incumbent on cloud and application security experts to understand how to mitigate it.

If you think your company does not use Firebase, think again. This platform is lurking in a shadow cloud account somewhere outside of your security organization's purview. By accepting this reality and learning about Firebase, we can guide lean development teams to move both quickly and safely with security guardrails. This paper is a starting point for your journey towards Firebase security and risk management.

Case Study: Real-time Customer Service Chat

This paper will reference a real-time customer service chat ReactJS component that we have open-sourced. This is an ideal use-case for Firebase. In fact, Firebase was formed by the team behind the now defunct group chat widget Envolve. While developing Envolve, they found their inspiration for Firebase: "We discovered our most exciting customers wanted to do more with real-time data than just send chat messages. They wanted to build real-time games, collaboration tools, analytics products, and much more."

Readers are encouraged to connect the component to their own Firebase project and follow along to gain hands-on experience with the technology. The exercises detailed here were originally featured in a SANS Tech Tuesday Workshop.

Realtime Database vs. Cloud Firestore – A Tale of Two Databases

Firebase's namesake offering is now referred to as the Firebase Realtime Database. In 2017, inspired by issues with running the original database at scale, Firebase announced the Cloud Firestore, incorporating "what developers love most about the Firebase Realtime Database while also addressing its key limitations like data structuring, querying, and scaling." It became generally available in 2019.

The Cloud Firestore is **not** the successor to the Realtime Database. Instead, Firebase is marketing it as an alternative database engine that has distinct use-cases. To help customers decide which database is best for them, Google has provided a survey that they can fill out to get a recommendation. Because both engines are here to stay, security professionals must familiarize themselves with both.

In 2018, security researchers reported a new variant of a vulnerability they dubbed "HospitalGown". To call it a "vulnerability" is slightly misleading as the issue was caused by Firebase operating properly according to its design. HospitalGown describes a frontend application that otherwise might be secure, but it leverages a wide-open backend database that has not been configured to require proper authentication or authorization. The report provided alarming numbers regarding the widespread nature of this misconfiguration of the Firebase Realtime Database:

- 3,000 iOS and Android apps were leaking data from 2,300 unsecured Firebase databases.
- 620 million downloads occurred of vulnerable Android applications alone.
- Multiple app categories were affected: "tools, productivity, health and fitness, communication, finance, and business apps."
- 62% of enterprises were claimed to be affected, although the report did not make clear how this was determined.
- 2.6 million plaintext user IDs and passwords, over 4 million "Protected Health Information" records, 25 million GPS location records, and 50,000 financial records were leaked.

Of the applications compromised, perhaps the most notable was Periscope, a mobile app owned by Twitter. The vulnerability was responsibly disclosed by Deeptiman Pattnaik. The disclosure is now public on HackerOne.

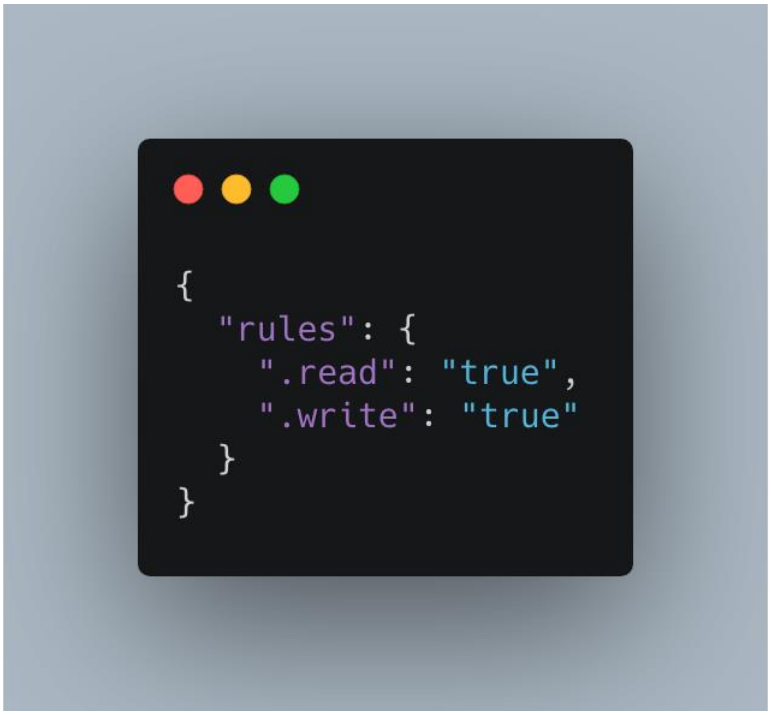
We can reproduce the HospitalGown misconfiguration using the real-time customer service chat mentioned above. After following the steps for Testing Locally, we must provide our Realtime Database with security rules that make the database world-readable and writable. There are several options that meet this qualification. For example, when a user creates a Firebase database with either engine, they can either start with *Locked mode* (no one can read from or write to the database) or *Test mode* (world-readable and writable for 30 days). Using *Test mode* results in the following HospitalGown misconfiguration:

Allow reads and writes as long as the current time is prior to 30 days from the instantiation of the Realtime Database (as measured by a 13-digit Unix Timestamp containing milliseconds).

Our case-study application also forwards all messages received by the Realtime Database to the Cloud Firestore. This might be useful as a stopgap when migrating from one engine to the other. The Cloud Firestore also has a *Test mode*, which has equivalent security rules:

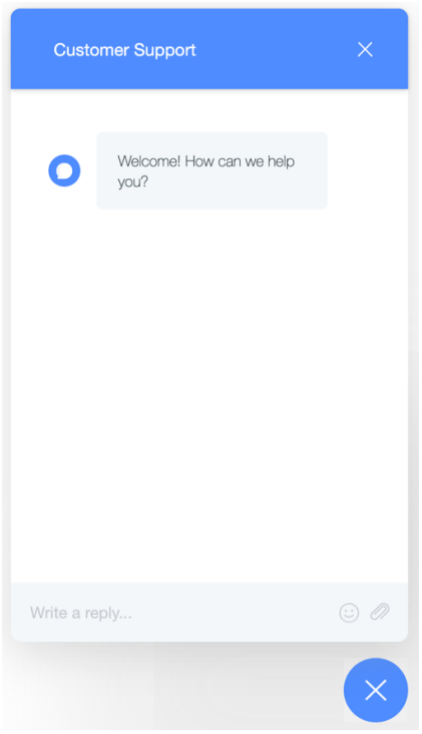
Allow reads and writes as long as the current time is prior to 30 days from the instantiation of the Cloud Firestore.

Prior to the addition of T*est mode*, several development guides recommended making the database world-readable and writable indefinitely:

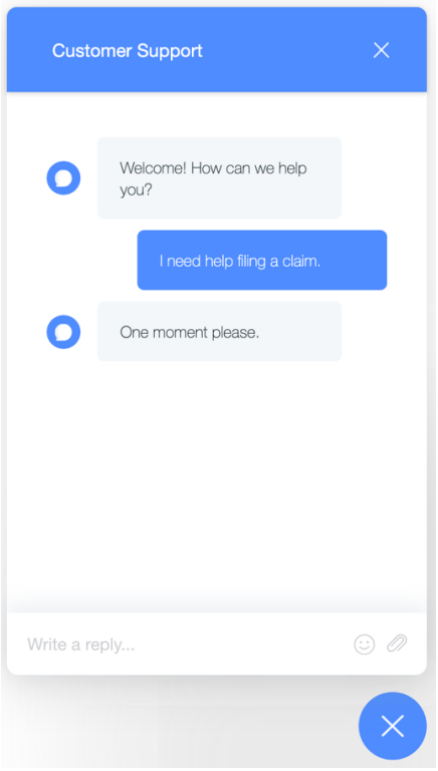


Allow reads and writes. No stipulations.

With the Realtime Database and Cloud Firestore misconfigured, three seconds after loading the chat application, we will receive a message from a fictional Customer Service agent:



Every time we send a message to the agent, we will automatically receive a reply stating, “One moment please.”:



If the frontend application can connect directly to the database, so can the attacker. First, they would need to locate the database’s endpoint. This can be done trivially in both Firefox and Chrome by right-clicking the page and selecting *View page source*. The result will look something like this:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="An implementation of kingofthestack/react-chat-window with a Firebase backend."
    />
    <title>firebase-react-chat-window</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  <script src="/static/js/bundle.js"></script><script src="/static/js/0.chunk.js"></script><script
src="/static/js/main.chunk.js"></script></body>
</html>
```

Clicking the link to */static/js/main.chunk.js* will open the minified JavaScript file. In it, we will be able to find the database config by searching for *firebaseio.com*:

This code is ugly (the technical term is uglified), but we can clearly make out the URL: ***https://<Database name>.firebaseio.com***

It is relatively easy to retrieve application secrets stored in web application code like HTML and JavaScript. It is harder to do this with mobile applications as these apps are compiled. However, it is still possible, as described by Carlos Polop's Firebase Pentest Methodology.

With the endpoint URL for a wide-open Realtime Database, the attacker merely has to append *.json* to it and load it. After loading *https://<Database name>.firebaseio.com/.json*, they will see something like the following:

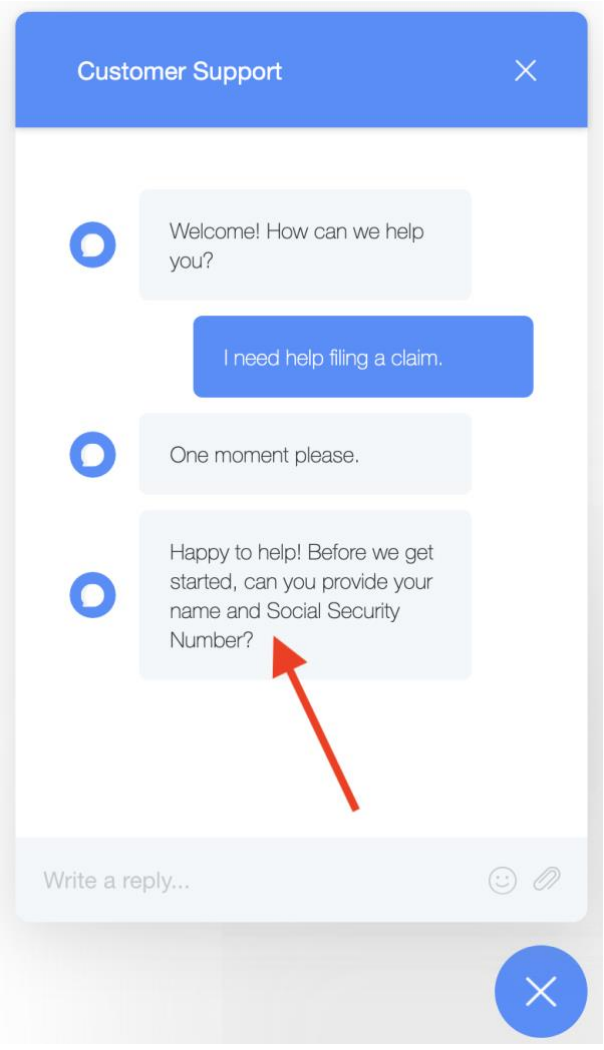
The private conversation between the customer and the agent is made public via the insecure backend Firebase database.

Because this database is also world-writable, the attacker could also hijack the session by adding a message that looks like it came from the agent. This too can be accomplished with a simple HTTP request, this time using the *POST* method:

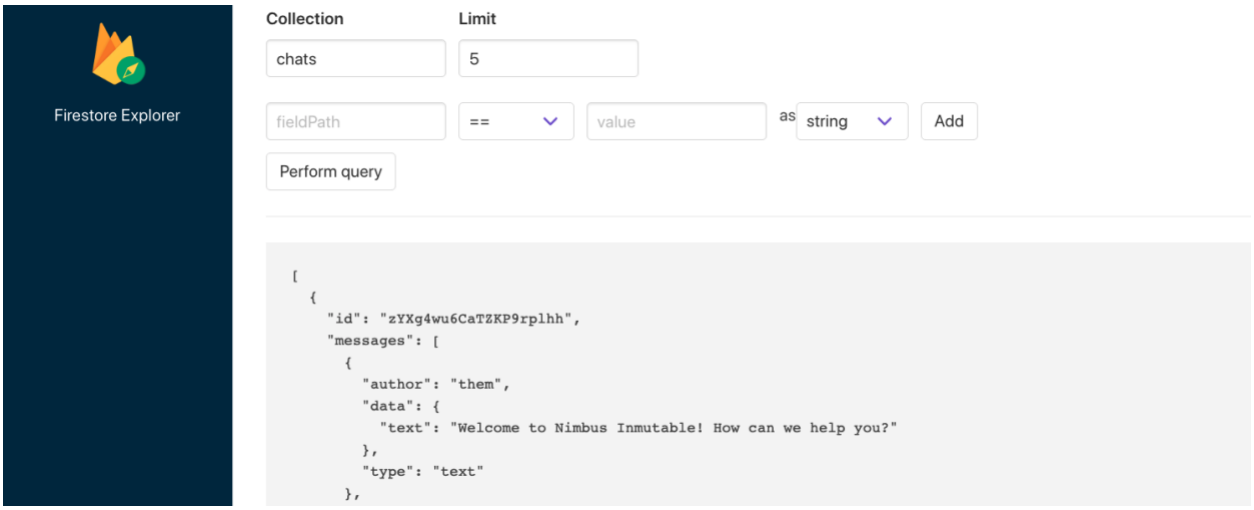
```
curl -X POST -d '{"author": "them", "type": "text", "data": { "text": "Happy to help! Before we get
started, can you provide your name and Social Security Number?" }}' \
  'https://<Database name>.firebaseio.com/chats/<Session ID>.json'
```

Session ID refers to the top-level key nested within the "chats" object (in the previous example, this would be ZJeBoN7bg6PGyOrjG0wcBShw3j22). If the attacker does not have access to curl, they could alternatively make the request with a web application such as Hoppscotch.

This request would instantly result in a message being sent to the customer prompting for highly sensitive information:



The Cloud Firestore is not quite as dangerous as the Realtime Database. It does not allow you to dump the entire contents of the database due to its structured nature, and it has no simple HTTP-based Application Programming Interface (API). However, if the attacker can extract the database config like above and knows the name of the collection they would like to query, they can retrieve all of the records using an application called the Firebase Explorer by Victor Nascimento:



The attacker ran a query fetching all records under the "chats" collection. They could have determined the name of this collection using enumeration or by compromising a Google Service Account to use with the Firebase Admin SDK.

To corrupt a Cloud Firestore dataset, the attacker can write custom client code using Node.js like so:

```
const firebase = require('firebase/app');
require('firebase/firestore');

firebase.initializeApp(<Configuration from application>);
const firestore = firebase.firestore();

(async () => {
  const collection = await firestore.collection('chats');

  await collection.add({
    foo: 'bar'
  });
})();
```

After configuring the client in the same way that the real application was configured, the attacker can add fraudulent records to the "chats" collection.

Adding Authentication

Firebase Security Rules cannot be constructed properly without Firebase Authentication. This is a Firebase-managed service that allows users to login through various methods and create an associated user record in the Firebase platform. Currently, Firebase Authentication supports three custom providers: *Email / Password*, *Phone*, and *Anonymous*. They also support authenticating through many social login providers: *Google* (Account), Google *Play Games*, Google *Game Center*, *Facebook*, *Twitter*, *GitHub*, *Yahoo*, *Microsoft*, and *Apple ID*.

The logos of the social login providers supported by Firebase Authentication.

We will focus on the Anonymous Provider. This provider “authenticates” a user without any user interaction. This is appropriate when any user, even those that we cannot identify, can use the application, but it stores private data for each anonymous user that should not be accessible by others. We can associate this “session data” with the user’s ID and only make it accessible to that user. Although this might initially seem dangerous, this is surprisingly effective for protecting the privacy of communications that must be unauthenticated, such as in our case-study.

With Firebase Authentication, developers can craft Firebase Database Security Rules that reference whether the user accessing the data is authenticated and what their user ID is. Using authentication, however, does not guarantee security. As recent as 2019, instead of letting developers choose between *Locked mode* and *Test mode*, newly created Realtime Databases would come with these default security rules:

```
{
  "rules": {
    ".read": "auth.uid != null",
    ".write": "auth.uid != null"
  }
}
```

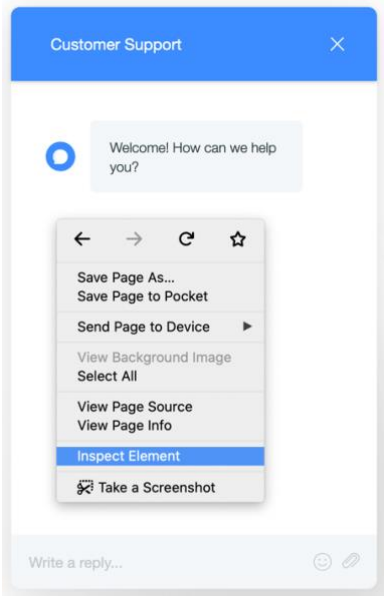
The equivalent Cloud Firestore rules are as follows:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if
          request.auth.uid != null;
    }
  }
}
```

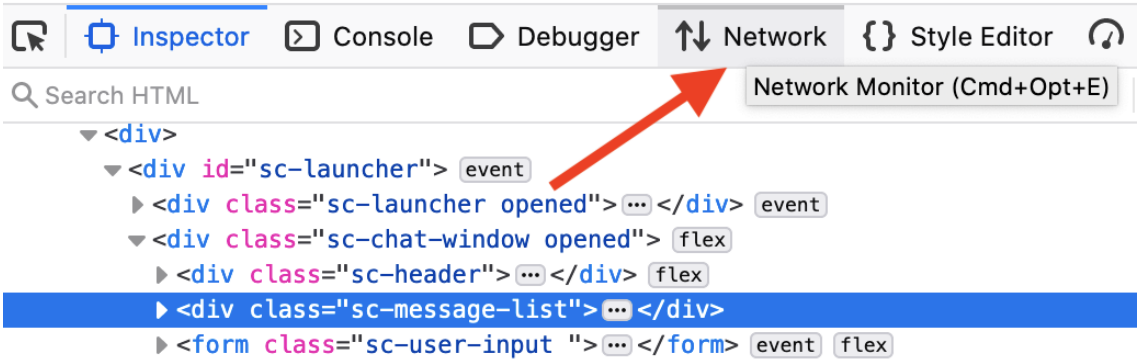
Although these rules require *authentication*, they do not validate *authorization*. Our case-study application is using the *Anonymous* authentication provider to automatically authenticate all users. This does not protect the application from Horizontal Authorization Bypass. In other words, authenticated users can view and corrupt data from other authenticated users.

After applying the above security rules, reloading the database endpoint (*https://<Database name>.firebaseio.com/.json*) will result in the following error:

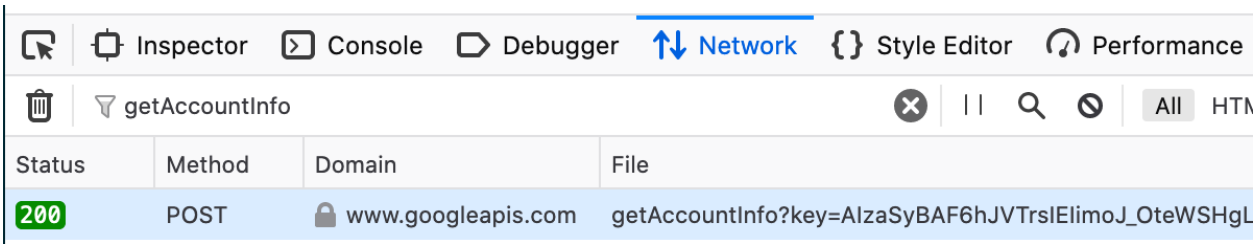
This is because our request is not authenticated. However, it is possible to authenticate these API calls using the same token used by the application. First, the token must be obtained by analyzing the application's traffic. Using Firefox, the attacker can right-click on the webpage and select *Inspect Element*:



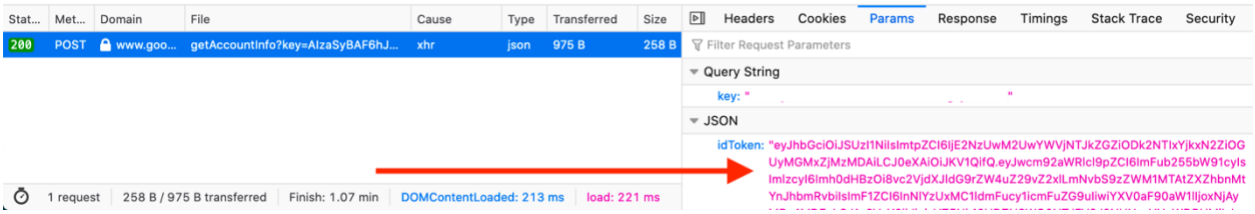
From there, they can navigate to the *Network* tab:



Then, they could filter traffic for *getAccountInfo*, the API call made to Google by Firebase applications to exchange an authentication token for more information on the user. After refreshing the page, they will see a *POST* request that they can select:



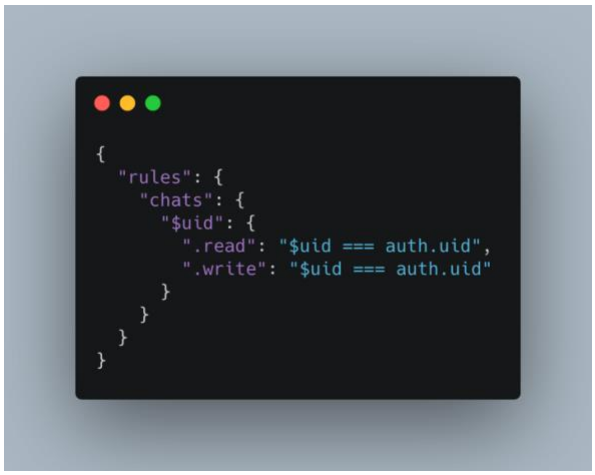
Next, in the side panel that appears, they can select the *Request* or *Params* tab depending on their version of Firefox to reveal the *idToken*:



Using this *idToken*, the attacker can craft an authenticated request that retrieves all of the data in the database:

https://<Database name>.firebaseio.com/.json?auth=<idToken value>

In our case-study, all users need access to some of the chat sessions, but no user should have access to all of the chat sessions. This can be fixed with an improved set of security rules:



These rules ensure that in order to access the session for *$uid*, the user must be authenticated with a matching user ID. This can be accomplished in the Firestore with the following rules:

Realtime Database Reconnaissance

Even with perfect authentication and authorization rules, the original Firebase Realtime Database has a serious, insurmountable weakness: it is far too easy for an attacker to find one. All Firebase Realtime Database endpoints have a predictable format: *<Something>.firebaseio.com*. There is no capability for taking these endpoints off of the public internet. Further, the database has several common enumeration flaws.

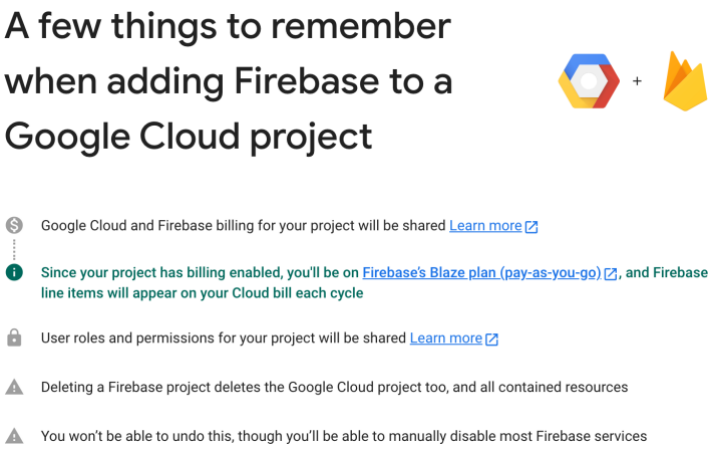
Firstly, when an attacker requests an endpoint, they can immediately determine whether or not it is valid. If they request */.json* on a valid endpoint the database is wide-open, they will receive all of the database contents. If the database does not exist, they will receive a *404 Not Found* response. However, if the database exists and is locked down, they will receive a *Permission denied* response, signaling to them that, with the appropriate access key, they would be able to retrieve the underlying data. Best practice dictates that the response in both the “Existing project, unauthenticated” and the “Database does not exist” case is identical. This way, the attacker would not be able to differentiate between these cases.

Secondly, Firebase previously exposed all subdomains of *firebaseio.com* via DNS A records. Although these no longer appear via DNS queries, searching DNSDumpster currently yields 345 cached records of firebaseio.com subdomains.

Finally, these database endpoints are indexed by various search engines. An article from March of 2020 indicated that searching for *site:firebaseio.com* yielded many Realtime Database endpoints on both Google and Bing. Although Google appears to have removed these records, using this query in Bing still yields many results.

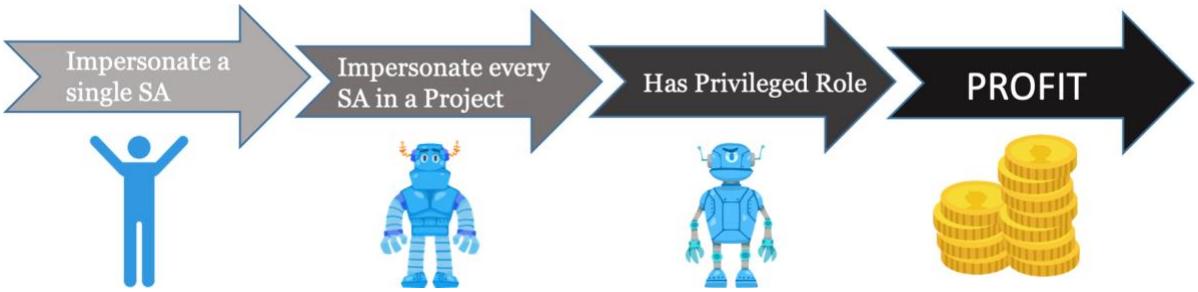
GCP Integration

Simply turning on Firebase can expose an organization using GCP to risk. Customers can choose to create Firebase projects as standalone resources or as a service within an existing GCP project. For management purposes, some might consider the latter option to be the better practice. Unfortunately, this comes with several consequences:



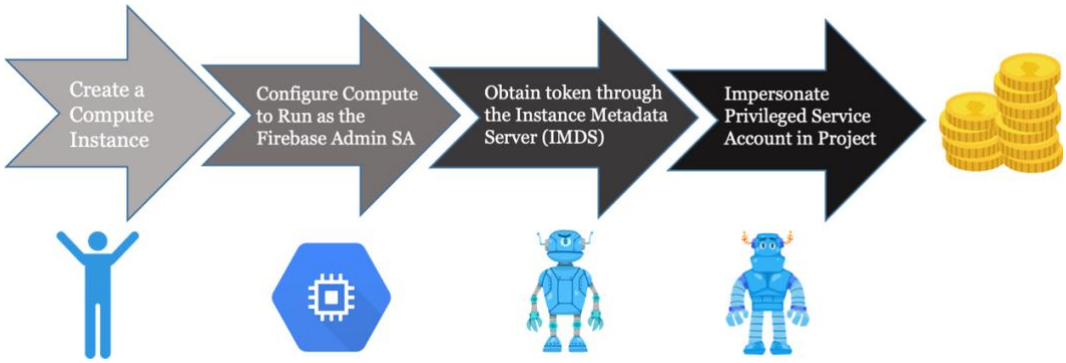
The concerning item from a security perspective is that "User roles and permissions for your project will be shared." To understand why this could introduce issues, one must familiarize themselves with permissions in GCP. Specifically, in GCP, *Service Accounts* are used to grant GCP permissions to non-user entities, such as services and infrastructure. For example, a service account is used to give a Virtual Machine (VM) in the Google Compute Engine (GCE) access to a bucket in Google Cloud Storage.

Given the necessary permissions, it is possible to "impersonate" a Service Account in GCP. If impersonating one Service Account allows an attacker to impersonate a second Service account, the attacker effectively has all of the permissions granted to that second account. GCP security expert and course contributor for SEC510: Multicloud Security Assessment and Defense Kat Traxler covers this concept at-length in her SANS webcast Privilege Escalation in GCP – A Transitive Path.



If a Service Account has desirable or profitable permissions, can be impersonated by another Service Account, and the attacker can impersonate that other Service Account, by the transitive property, the attacker has the profitable permissions. Images provided by Jumsoft (https://www.jumsoft.com/support/).

As soon as a customer activates Firebase within their GCP account, a Service Account is created called the *firebase-adminsdk*. This Service Account has the ability to impersonate any other Service Account in that GCP project. So, if the attacker merely has the iam.serviceAccounts.actAs permission, they can launch a GCE VM that uses this powerful Service Account. From there, if the attacker can establish a Secure Shell (SSH) connection to the VM, which is likely given GCP's wide-open default network firewall rules, they can extract temporary credentials for the *firebase-adminsdk* Service Account from the VM's Instance Metadata Server (IMDS). This would give the attacker the permissions necessary to impersonate any other Service Account in the GCP project.



Compliance Concerns

Using Firebase can also expose an organization to compliance violations. Most notably, Firebase does not allow the customer to define where its data is stored and processed. The original three Firebase services (Realtime Database, Hosting, and Authentication) operate solely out of the United States. As Firebase Hosting utilizes a Content Delivery Network (CDN), its data is nonetheless distributed globally: "Firebase Hosting caches response data in globally distributed content delivery servers to provide the service, but processes all requests in the United States."

Every other Firebase service is global, meaning that the data stored within it could be processed at any datacenter owned by Google and the Google Cloud Platform. This accounts for approximately 50 distinct regions. Although this is appealing for product development as it

helps with high availability, this inflexibility makes it impossible for companies that follow certain compliance regimes to use Firebase.

The Firebase documentation referenced below claims that customers can select the data location for a variety of global services. Unfortunately, the link provided does not reveal any details on how this can be done.

Summary

Firebase, although popular with developers for their unique and innovative features, can often be at-odds with security. Technically, Firebase provides many security features and controls, shifting the responsibility for security to the customer. All cloud providers do this. However, Firebase is unique in the degree to which their services are insecure by default and easy to misconfigure. Further, their flagship product has several inherent security concerns, including the ability to easily scan the internet for databases. Finally, the global nature of most of their services can make it impossible to use Firebase while staying compliant in many cases.

Organizations using Firebase need to determine if it is viable to continue using them long-term. Regardless, this section covers ways to minimize the damage caused by active Firebase resources. If this is sufficient, that is excellent. If not, this reduces risk until the organization can rearchitect their applications to use alternative technologies.

About the Author: Brandon Evans is the lead author of *SEC510: Multicloud Security Assessment and Defense* and an instructor for *SEC540: Cloud Security and DevOps Automation*. His full-time role is as a Senior Application Security Engineer at Asurion, where he provides security services for thousands of his coworkers in development across several global sites responsible for many web applications. This includes performing secure code reviews, conducting penetration tests, and evangelizing the importance of creating secure products. Read his full bio here.

Thanks to Kat Traxler (@NightmareJS) for her contributions regarding Firebase's GCP integration and to Tarl Smith for providing details on Firebase data sovereignty and compliance.

Last Updated: October 12th, 2020

Upcoming SANS Training

Click here to view a list of all SANS Courses

SANS Community CTF	**,**	**Oct 15, 2020 - Oct 16, 2020**	**Self Paced**
SANS Sydney 2020	**Sydney, AU**	**Nov 02, 2020 - Nov 14, 2020**	**Live Event**
SANS Secure Thailand	**Bangkok, TH**	**Nov 09, 2020 - Nov 14, 2020**	**Live Event**
APAC ICS Summit & Training 2020	**Singapore, SG**	**Nov 13, 2020 - Nov 28, 2020**	**Live Event**
SANS Community CTF	**,**	**Nov 19, 2020 - Nov 20, 2020**	**Self Paced**
SANS Local: Oslo November 2020	**Oslo, NO**	**Nov 23, 2020 - Nov 28, 2020**	**Live Event**
SANS Wellington 2020	**Wellington, NZ**	**Nov 30, 2020 - Dec 12, 2020**	**Live Event**
SANS OnDemand	**OnlineUS**	**Anytime**	**Self Paced**
SANS SelfStudy	**Books & MP3s OnlyUS**	**Anytime**	**Self Paced**