Despite misleading marketing, Israeli company TeleMessage, used by Trump officials, can access plaintext chat logs

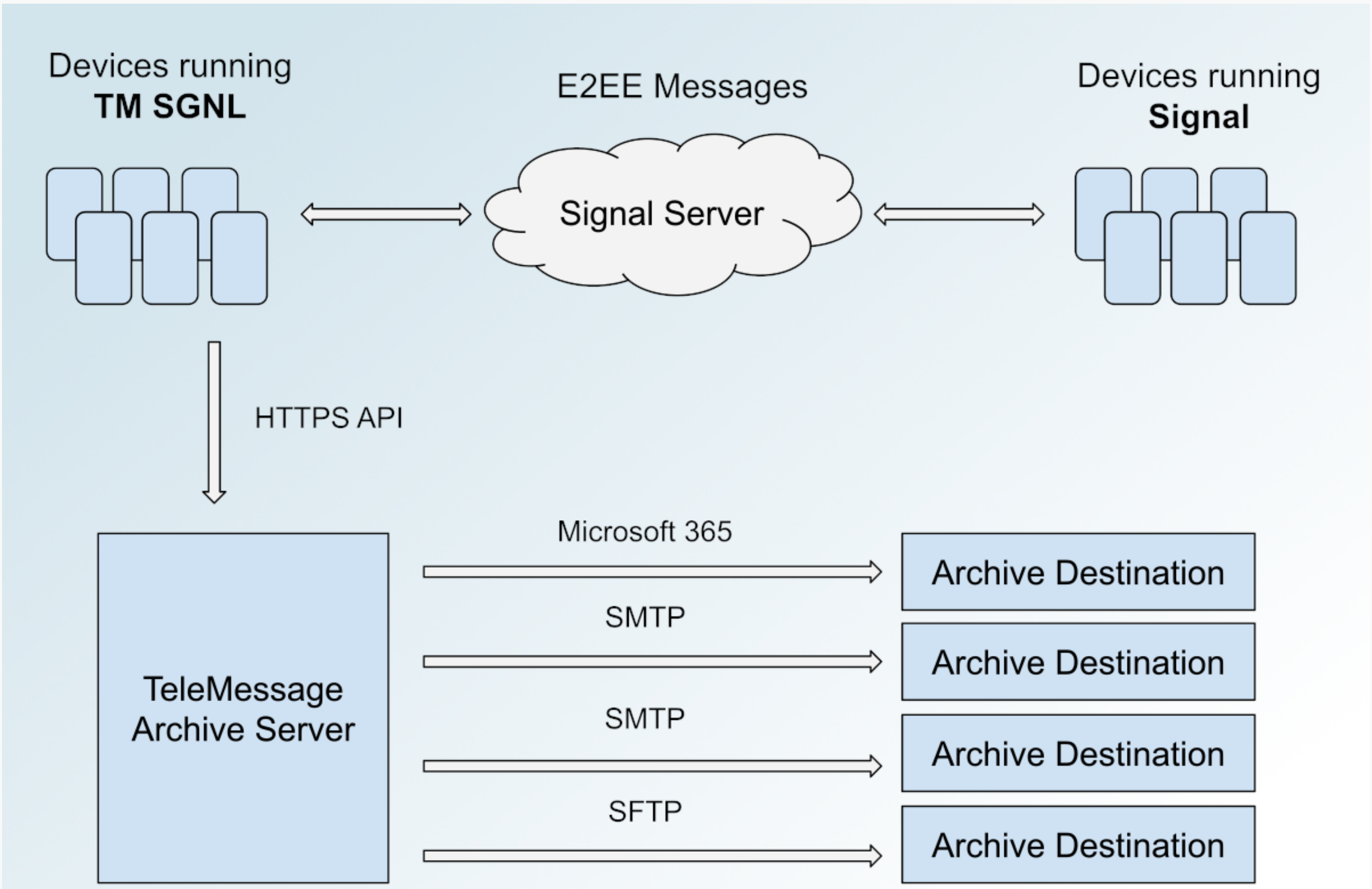


Diagram of how TeleMessage archives Signal messages, based on my analysis of the source code

Micah Lee

06 May 2025

Despite their misleading marketing, TeleMessage, the company that makes a modified version of Signal used by senior Trump officials, can access plaintext chat logs from its customers.

In this post I give a high level overview of how the TeleMessage fake Signal app, called TM SGNL, works and why it's so insecure. Then I give a thorough analysis of the source code for TM SGNL's Android app, and what led me to conclude that TeleMessage can access plaintext chat logs. Finally, I back

up my analysis with as-of-yet unpublished details about the hack of TeleMessage.

But first, here's a quick timeline of events.

- On Thursday, 404 Media reported that in the Reuters photo showing former National Security Advisor and war criminal Mike Waltz checking his Signal messages under the table, he was actually using an obscure modified Signal app called TM SGNL, and not the real and actually secure Signal app.
- On Friday, I wrote an analysis of everything I could find out about TM SGNL using OSINT, including the fact that it's nearly impossible to install without a device enrolled in an MDM service that's tied to an Apple Business Manager or a Google Enterprise account.
- On Saturday, after discovering that TeleMessage published the source code for the TM SGNL apps for Android and iPhone themselves, I re-published them on GitHub with the goal of making them easier to research. (It looks like the iOS source code is actually just unmodified Signal, so maybe they actually only published their Android code.)
- On Saturday night, an anonymous source told me they hacked TeleMessage.
- On Sunday, I, along with Joseph Cox, published an article about the hack to 404 Media (and to my blog).
- On Monday, NBC News reported that TeleMessage suspended its service after a second hacker breached TeleMessage and "downloaded a large cache of files."
- Today, Senator Ron Wyden published a letter, which cites the 404 Media article and my analysis of TM SGNL, to Attorney General Pam Bondi, requesting that the Justice Department investigate the "serious threat to U.S. national security posed by TeleMessage, a federal contractor that sold dangerously insecure communications software to the White House and other federal agencies."

The devastating hacks confirm the analysis that I'm sharing in this post: that TeleMessage's server – hosted on the public AWS cloud, run by an Israeli company that's led by a former IDF spook – has plaintext access to the Signal chat logs they're archiving (along with chat logs for Telegram, WeChat, and WhatsApp).

If you find this interesting, subscribe to get these posts emailed directly to your inbox. If you want to support my work, considering becoming a paid supporter.

Table of Contents

Overview

TeleMessage makes modified versions of popular messaging apps, including Signal, WhatsApp, Telegram, and WeChat. The modified Signal app is almost entirely identical to the authentic version of Signal, except it also archives copies of every message to a destination determined by the TeleMessage customer. (Presumably, WhatsApp, Telegram, and WeChat work in the same way, but I have only analyzed the TM SGNL for Android source code.)

The TM SGNL app

TM SGNL is interoperable with Signal. When a TM SGNL user registers a new account, they're registering it with the official Signal server. TM SGNL users can send messages to Signal users and visa versa. If you're a Signal user, you have no way of knowing when you're talking to a TM SGNL user, because the apps are nearly identical and use the same infrastructure.

This is how Mike Waltz could accidentally add The Atlantic editor-in-chief Jeffrey Goldberg to a group chat where they discussed bombing an apartment building full of civilians: Waltz was presumably using TM SGNL, and Goldberg was presumably using Signal.

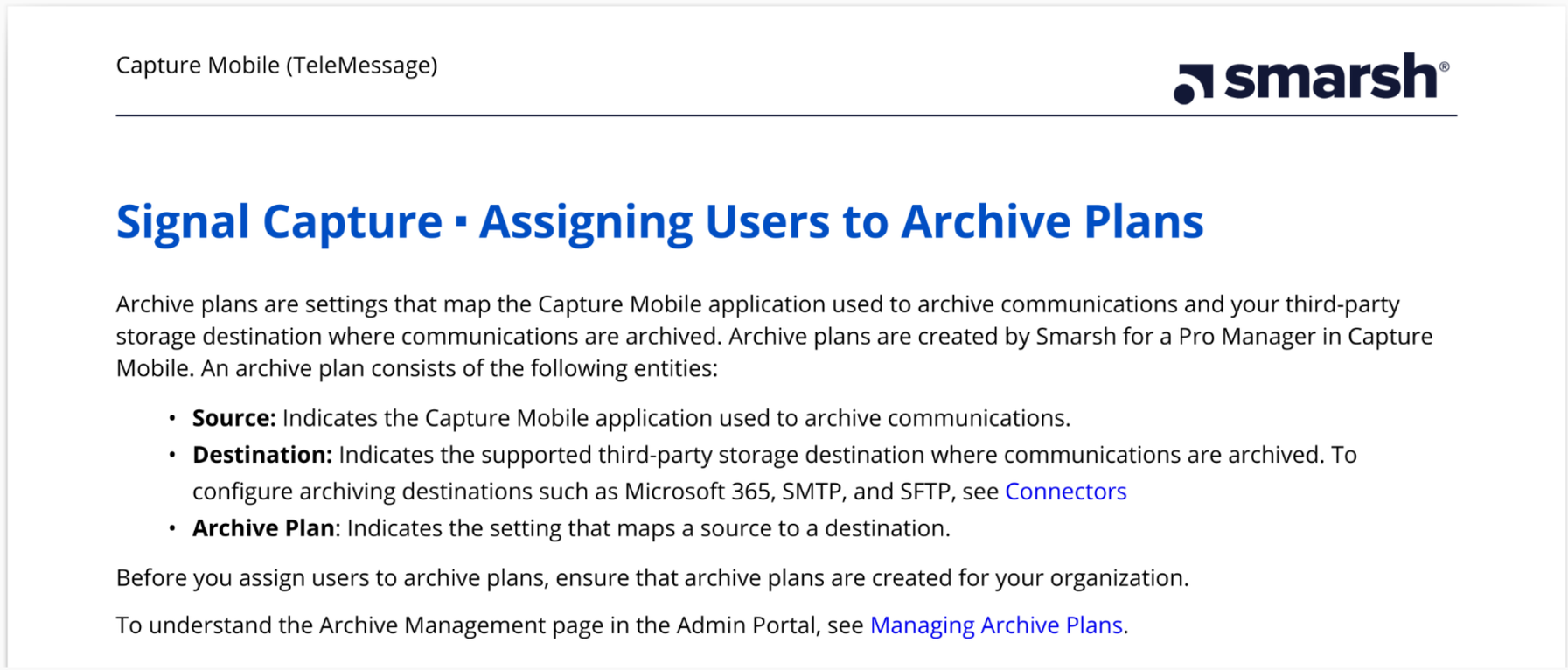
Note that we don't know how long Trump officials have been using TM SGNL. It's possible they've been using it since around Trump's inauguration on January 20. If this is the case, the chat logs for the Signal group where Waltz invited Goldberg will have been collected by TeleMessage. But it's also possible that they only started using TM SGNL after Signalgate, maybe as a way to attempt to start complying with record keeping laws. So far, we don't know the timeline. This would be good to get to the bottom of to understand what laws they have already broken.

Anyway, on Mike Waltz's phone – and quite likely, the phones of Pete Hegseth, Marco Rubio, Tulsi Gabbard, JD Vance, and many others, possibly even including Donald Trump – the TM SGNL app basically works like this:

- Every message the TM SGNL app sees gets stored in a SQLite database with a `status` column set to `WaitingToBeDelivered`.
- TM SGNL registers with the phone's background syncing service. On a regular basis, the app runs some sync code that selects messages that are `WaitingToBeDelivered`. If there are any, it delivers them to TeleMessage's archive server at `https://archive.telemessage.com`, and updates their status to `Sent`.

TeleMessage's archive server

As I discussed in my initial OSINT analysis, according to this documentation PDF, the admins for organizations that use TeleMessage set up archive plans and assign users to them.



Capture Mobile (TeleMessage)

smarsh®

Signal Capture ▪ Assigning Users to Archive Plans

Archive plans are settings that map the Capture Mobile application used to archive communications and your third-party storage destination where communications are archived. Archive plans are created by Smarsh for a Pro Manager in Capture Mobile. An archive plan consists of the following entities:

- **Source:** Indicates the Capture Mobile application used to archive communications.
- **Destination:** Indicates the supported third-party storage destination where communications are archived. To configure archiving destinations such as Microsoft 365, SMTP, and SFTP, see Connectors
- **Archive Plan**: Indicates the setting that maps a source to a destination.

Before you assign users to archive plans, ensure that archive plans are created for your organization.

To understand the Archive Management page in the Admin Portal, see Managing Archive Plans.

How TM SNGL archive plans work, on page 22 of the documentation PDF

Each archive plan has a source messaging app (like TM SGNL) and a destination, which is controlled by the TeleMessage customer. Destinations can include Microsoft 365, email servers (SMTP), or file servers (SFTP). The admin assigns TeleMessage users – like Mike Waltz – to an archive plan, which determines where their chat logs will get archived.

Once the TM SGNL app sends chat logs to the archive server, the archive server is supposed to do something like this: It looks up the user that sent the chat log, then looks up that user's archive plan, then forwards the messages to destination defined in the archive plan (via SMTP or SFTP), and presumably (but who really knows for sure) deletes the chat logs from the archive server.

Here's a diagram of how the whole system appears to work:

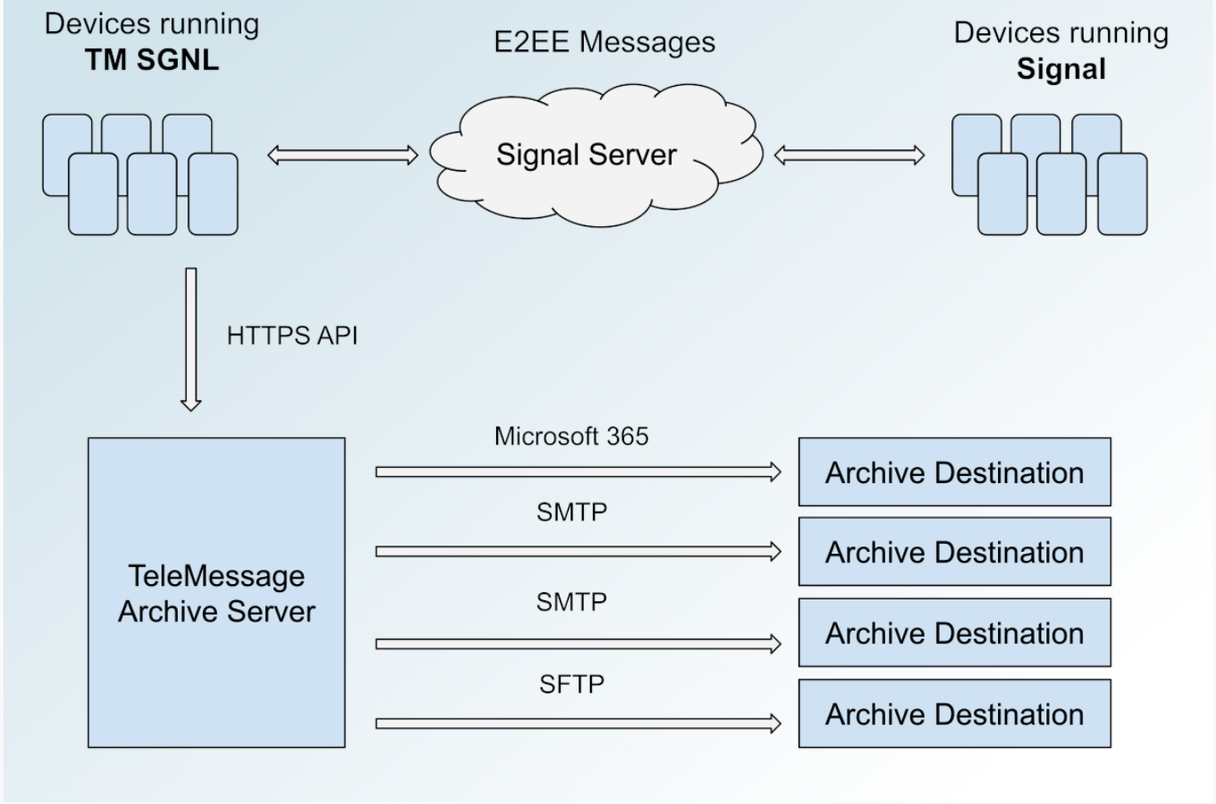


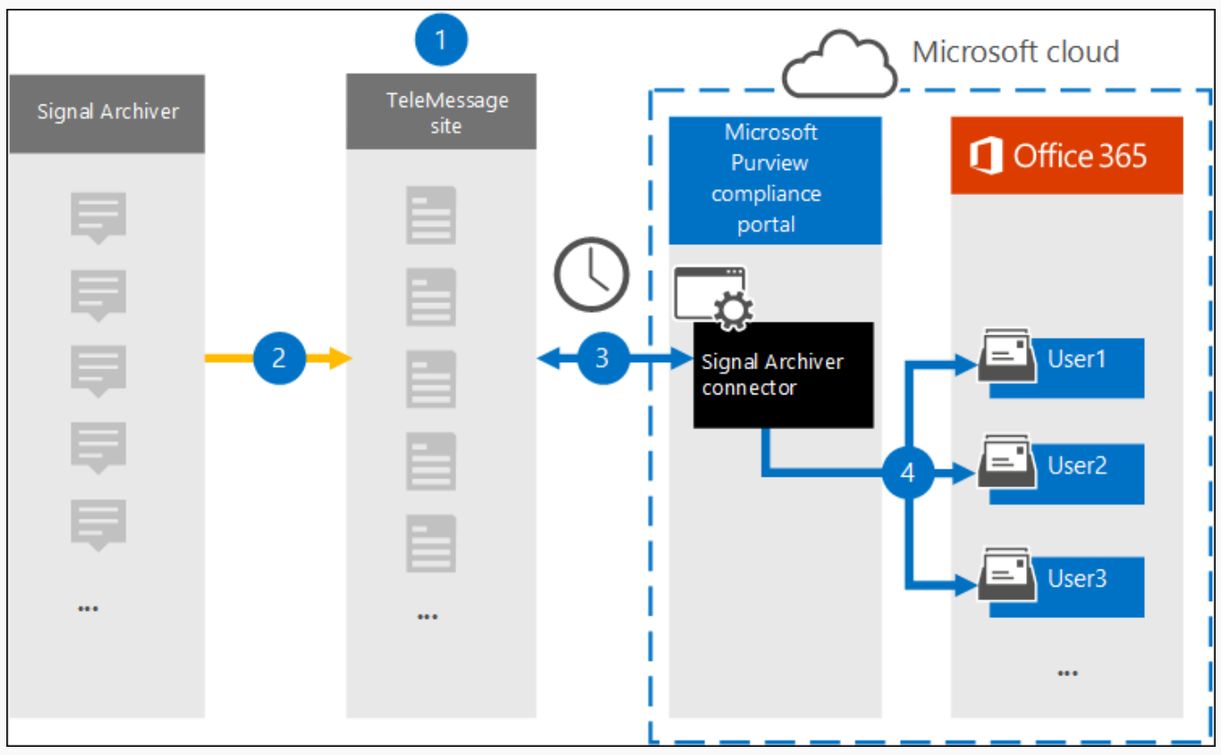
Diagram of how TeleMessage archives Signal messages, based on my analysis of the source code

Microsoft 365

The archive server appears to connect directly to SMTP and SFTP destinations to push chat logs. However, according to Microsoft's documentation, Microsoft 365 works the opposite way: Microsoft 365 logs into the archive server and pulls the chat logs once a day. Their docs state:

> The Signal Archiver connector that you create in the Microsoft Purview portal connects to the TeleMessage site every day and transfers the email messages from the previous 24 hours to a secure Azure Storage area in the Microsoft Cloud.

They even published their own diagram explaining how it works:



Using a connector to archive Signal communication data in Microsoft 365, from Microsoft's documentation

Why this is so terribly insecure

Signal is the gold standard of end-to-end encrypted messaging apps.

Messages are encrypted between endpoints – whether that's a phone running Signal, a computer running Signal Desktop, or even a phone

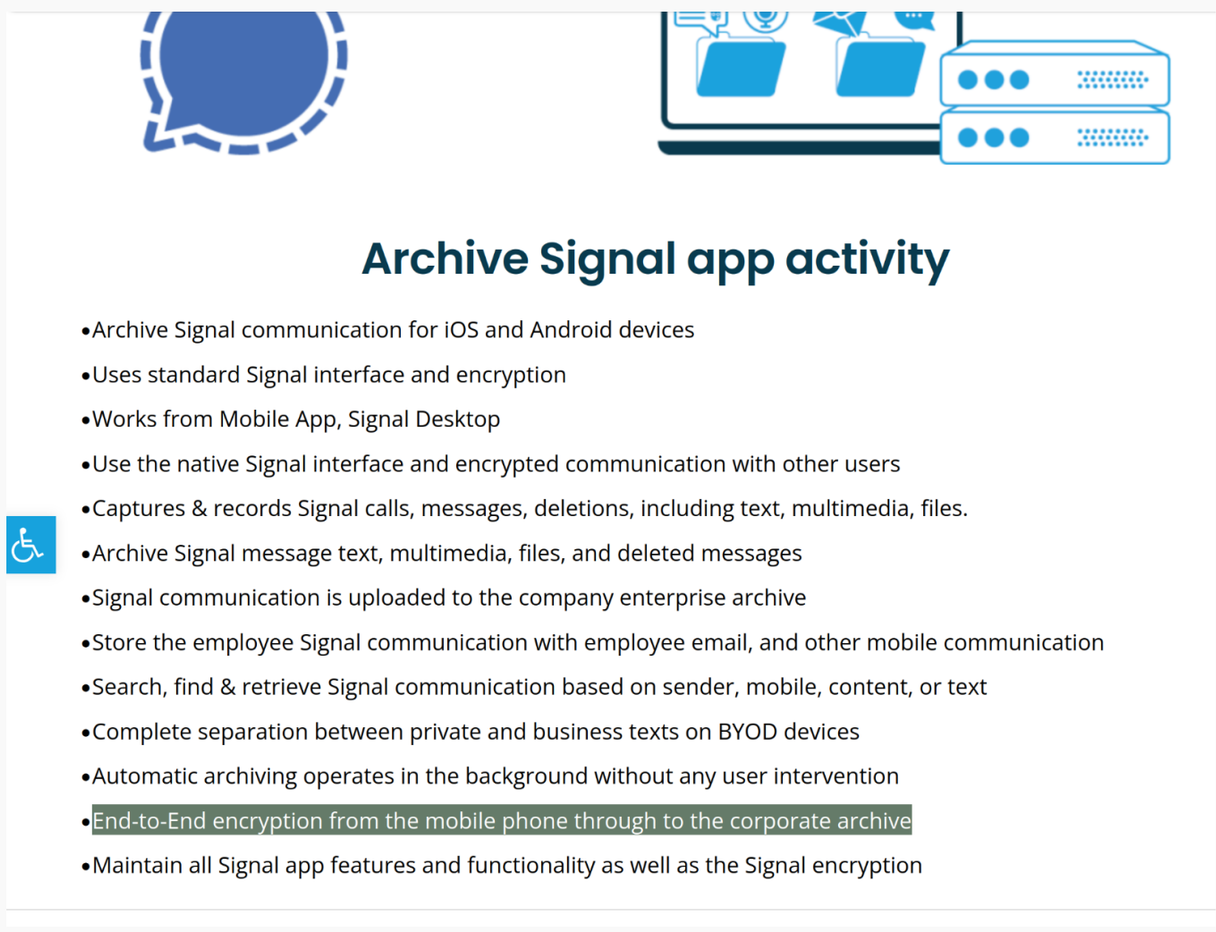
running TM SGNL. The Signal server, and any internet eavesdroppers, cannot access the chat logs.

However, once they're at an endpoint, they are in plaintext (if they weren't, you wouldn't be able to read your texts). At this point, they're protected by various forms of disk encryption depending on the device. This is how Signal messages sometimes end up as evidence in court records: someone's phone or laptop with Signal installed was searched, after the messages were already decrypted.

TM SGNL completely breaks this security. The communication between the TM SGNL app and the final archive destination is not end-to-end encrypted.

TeleMessage lies about this in their marketing material, claiming that TM SGNL supports "End-to-End encryption from the mobile phone through to the corporate archive."

Here's a screenshot from an archived version of their site (since they've taken down all of the content):



TeleMessage falsely claims that TM SGNL supports "End-to-End encryption from the mobile phone through to the corporate archive"

Instead, TM SGNL sends the plaintext, already decrypted versions of chat logs to the archive server. (There might be some encryption involved some of the time, which I go over the "Corroborative evidence from the hack" section at the bottom.)

The archive server then forwards these to the destination.

Anyway, after TM SGNL decrypts messages, it sends the plaintext chat logs to TeleMessage's archive server. **At this point, a lot of people might have access to the chat logs.**

The archive server was hosted in the public AWS cloud in their data center

in Northern Virginia. This is not an approved place to store classified information, and potentially rogue AWS employees could have had access. This server was open to the public – anyone in the world could send HTTP requests to it to try to get chat logs back in a response. On Saturday, one of those people did.

I'm saying the server "was" hosted in AWS because TeleMessage took down their archive server shortly after we published the story about the hack. As of this writing, their archive server is offline.

TeleMessage also might be sharing chat logs with Israeli intelligence.

TeleMessage is an Israeli company. It was founded in 1999 by Guy Levit, the same year he left his job in the Israel Defense Force where he "served as the head of the planning and development of one of the IDF's Intelligence elite technical units," according to his bio on the TeleMessage website (they have since taken all the content off their website).

Levit, and others at his firm, have access to the archive server and all of the chat logs it contains. I don't know what Israel's version of the Patriot Act is like, but I do know that it would be trivial for TeleMessage to add a bit of code that forwards a copy of everything to Israeli intelligence – far simpler than it was to add code to Signal to forward a copy to themselves.

To be clear, there's no evidence yet that TeleMessage is sharing chat logs with the Israeli government. **But the fact that they designed their archiving system to not be end-to-end encrypted, and that they lie about it, is quite a big red flag.**

This is the app that Mike Waltz uses. While we don't really know the timeline, it's possible that everyone in the 19-person Signalgate group, where they discussed bombing Yemen, were also using this app same app. These include JD Vance, John Ratcliffe, Marco Rubio, Pete Hegseth, Stephen Miller, Tulsi Gabbard, and others. It's plausible that Israeli intelligence has been reading the internal chat logs of the most powerful members of Trump's authoritarian government.

Analysis of the TM SGNL Android source code

I've only analyzed the the TM SGNL Android source code, not the iPhone source code. I'm assuming that the iPhone app works in the same way as the Android app, and anything that I learn from analyzing the Android app is applicable to the iPhone app as well.

I've focused on Android because it's much easier to reverse engineer Android apps than iPhone apps. There are excellent tools available for analyzing Android apps, like apktool, apkeep, and even the official Android Studio. Furthermore, Android apps are programmed in Java and Kotlin (which both compile to Java bytecode), and it's easy to decompile Java bytecode, turning it back into human readable source code, and making it so much easier to work with.

Also, while TeleMessage posted links to ZIP files claiming to be contain the Android and iOS source code on their website, the iOS version appears to actually be the source code for Signal for iOS itself, without any extra TeleMessage code added.

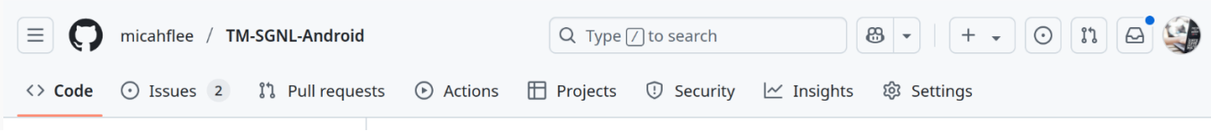
> As you can see in the LICENSE file included with the source code, TM SGNL is licensed under GNU Affero General Public License v3.0. This gives me, and everyone else, the unlimited right to access, analyze, reverse engineer, and pretty much do anything else we wish to with the code, so long as we release any derivative works under the same license.

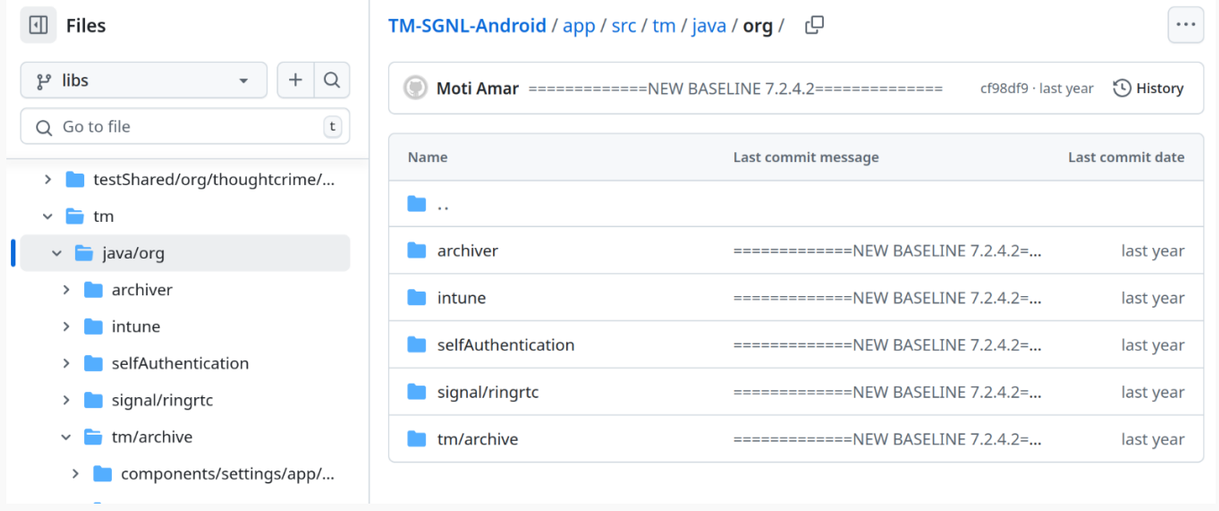
I've only had a few days to look at this code so far – and I paused work on analyzing it so I could break the story about TeleMessage getting hacked – so there's a lot that I still don't fully understand. It's also possible I have gotten things wrong – please let me know if that's the case. In any case, I'm showing all of my work here so that others can reproduce it and build on it.

Decompiling shared libraries

The TM SGNL source code is mostly the same as the Signal for Android source code, but there are some differences. Most of the new code can be found in these three places:

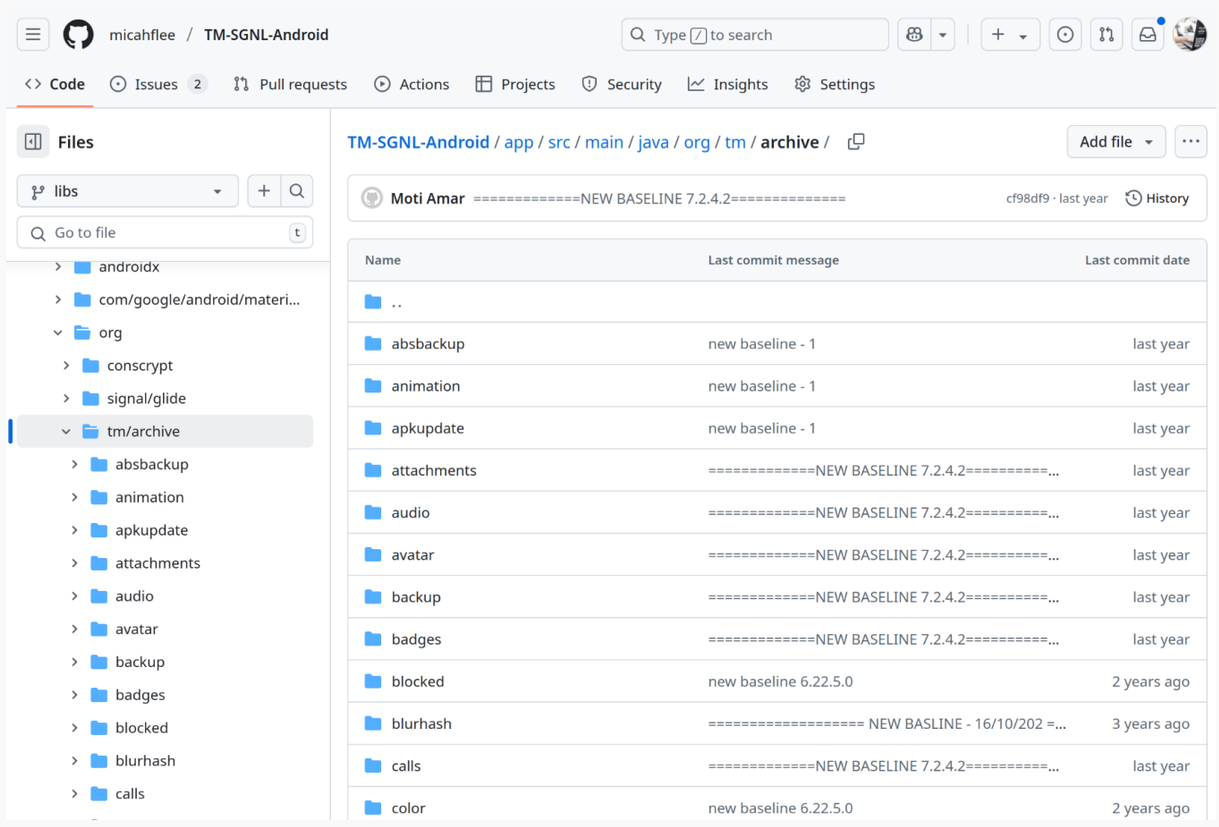
The app/src/tm/java/org/ folder contains TeleMessage code.





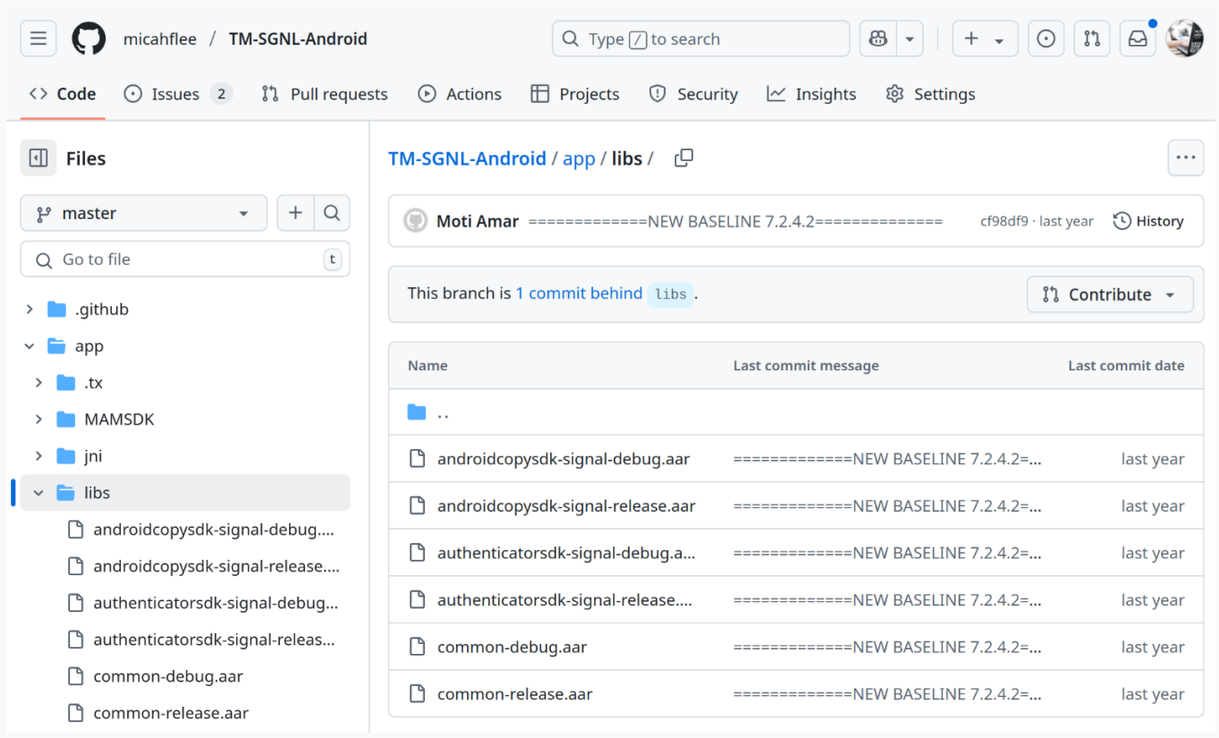
TeleMessage code in app/src/tm/java/org/

The app/src/main/java/org/tm/archive/ folder also contains TeleMessage code:



More TeleMessage code in app/src/main/java/org/tm/archive/

And finally, the app/libs/ folder contains TeleMessage's shared libraries in Android archive (`.aar`) format. These include `androidcopysdk-signal`, `authenticatorsdk-signal`, and `common`. Shared libraries are re-usable pieces of code that can be imported into different projects. These libraries appear to contain code that might also be shared in TeleMessage's other Android apps for WhatsApp, WeChat, and Telegram. The Android archive files are compressed Java bytecode.



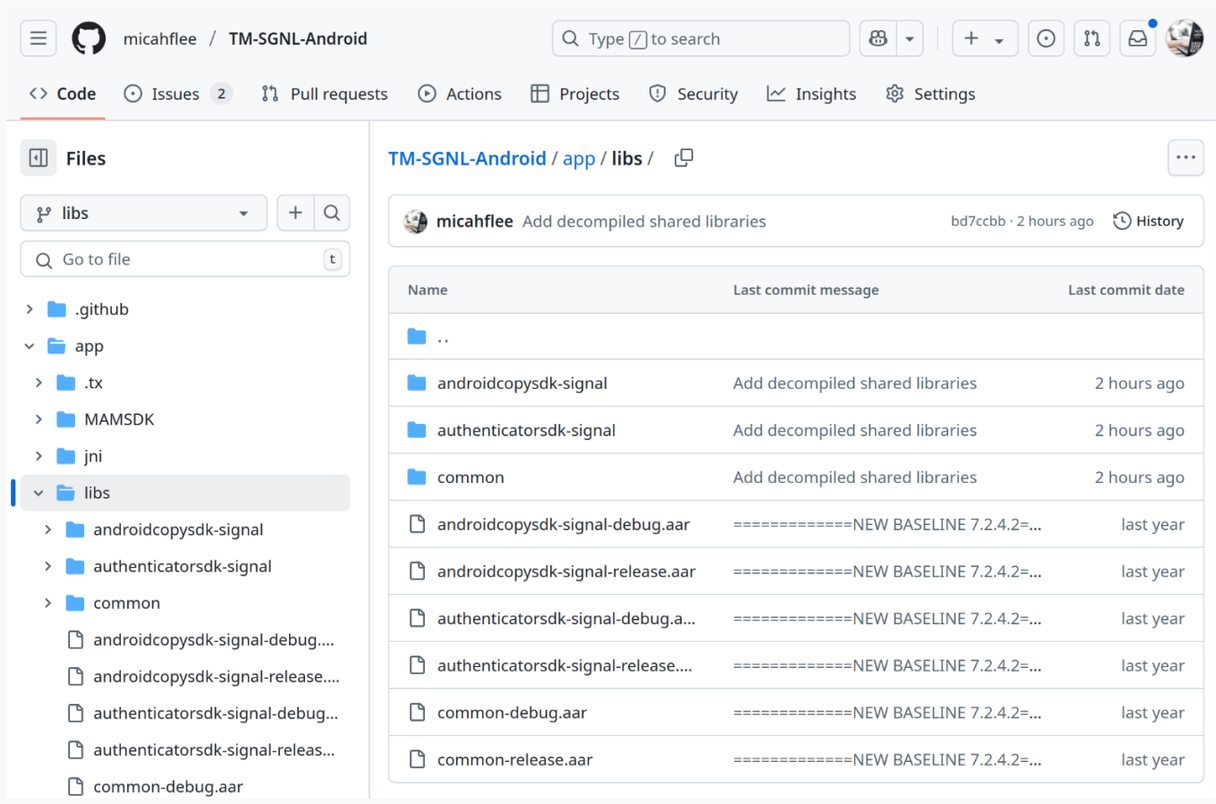
Shared libraries in app/libs/

The first step to reverse engineering this is to turn these shared libraries

into actual human-readable (or at least, human nerd-readable) Java code.

There are various local tools that can do this, but I've found using online services to decompile Android apps is the simplest approach. I ran the release versions of the shared libraries through this Android archive decompiler, and it gave me zip files with the decompiled source code.

To make things easier for others to follow along, I have made a new git branch in my TM-SGNL-Android repository called `libs` and I committed the decompiled shared library code there. I've made the `libs` branch the default branch, too. You can now access all of the shared library code in the apps/libs/ folder of the `libs` branch:



The app/libs/ folder, with folders full of human-readable Java source code

Now that we all have access to the same source code, I will try to give a brief tour of what I've found to be the relevant pieces of code to bring me to my conclusion.

Important components

Here are some important components in the codebase:

- `SignalDatabase`: This class represents the SQLite database that stores all of the Signal data (messages, attachments, media, threads, identities, drafts, groups, recipients, stickers, reactions, and so on), for delivering to the archive server.
- `DataGrabber`: This class (in the `androidcopysdk` shared library) seems to do a few things.
  - Many of its 2,577 lines of code are devoted to capturing the phone's SMS and MMS messages, as well as call logs – though I don't think this code is used in the SGNL app.
  - It also has a `setMessage` method, which the SGNL app does use. In this method, it uses a `ContentResolver` to store messages in a separate SQLite database, staging messages before they get sent to the archive server. I'll call this the staging database.

- `TeleMessageApplicationDependencyProvider`: This class provides access to

the SDK module (an object with pointers to various singleton components like the `DataGrabber` and `SignalDatabase` objects).

  - It also provides access to the `messageStoreObserver`. This object can have processors, which are basically hooks that get executed when message statuses change.
  - `ArchiveMessagesProcessor`: This is a `messageStoreObserver` processor. When messages in `SignalDatabase` get created (or edited or deleted), this runs `DataGrabber.setMessage` to store the message in the staging database.

- `SyncAdapter`: This class (also in the `androidcopysdk` shared library) is an Android sync adapter. It defines a background service that, every time the `onPerformSync` method is called, uses the `ContentResolver` to select `WaitingToBeDelivered` data from the staging database, and then submits it to the TeleMessage's archive server.

Now I'm going to go into a bit more detail about these components, and how they tie together.

SignalDatabase

Here's the beginning of the `SignalDatabase` class definition. As you can see, it defines the tables that will be used to store different types of Signal data:

```
open class SignalDatabase(private val context: Application, da
  SQLiteOpenHelper(
    context,
    DATABASE_NAME,
    databaseSecret.asString(),
    null,
    SignalDatabaseMigrations.DATABASE_VERSION,
    0,
    SqlCipherErrorHandler(DATABASE_NAME),
    SqlCipherDatabaseHook(),
    true
  ),
  SignalDatabaseOpenHelper, IDatabase<Long> { // TM_SA impleme

  val messageTable: MessageTable = TeleMessageTable(context, t
  val attachmentTable: AttachmentTable = TeleAttachmentTable(c
  val mediaTable: MediaTable = MediaTable(context, this)
  val threadTable: ThreadTable = ThreadTable(context, this)
  val identityTable: IdentityTable = IdentityTable(context, th
  val draftTable: DraftTable = DraftTable(context, this)
  val groupTable: GroupTable = GroupTable(context, this)
  val recipientTable: RecipientTable = RecipientTable(context,
  val groupReceiptTable: GroupReceiptTable = GroupReceiptTable(
  val preKeyDatabase: OneTimePreKeyTable = OneTimePreKeyTable(
  val signedPreKeyTable: SignedPreKeyTable = SignedPreKeyTable(
  val sessionTable: SessionTable = SessionTable(context, this)
  val senderKeyTable: SenderKeyTable = SenderKeyTable(context,
  val senderKeySharedTable: SenderKeySharedTable = SenderKeySh
  val pendingRetryReceiptTable: PendingRetryReceiptTable = Pen
  val searchTable: SearchTable = SearchTable(context, this)
  val stickerTable: StickerTable = StickerTable(context, this)
  val storageIdDatabase: UnknownStorageIdTable = UnknownStorag
  val remappedRecordTables: RemappedRecordTables = RemappedRec
  val mentionTable: MentionTable = MentionTable(context, this)
```

```
val paymentTable: PaymentTable = PaymentTable(context, this
val chatColorsTable: ChatColorsTable = ChatColorsTable(conte
val emojiSearchTable: EmojiSearchTable = EmojiSearchTable(co
val messageSendLogTables: MessageSendLogTables = MessageSend
val avatarPickerDatabase: AvatarPickerDatabase = AvatarPicke
val reactionTable: ReactionTable = ReactionTable(context, th
val notificationProfileDatabase: NotificationProfileDatabase
val donationReceiptTable: DonationReceiptTable = DonationRec
val distributionListTables: DistributionListTables = Distril
val storySendTable: StorySendTable = StorySendTable(context,
val cdsTable: CdsTable = CdsTable(context, this)
val remoteMegaphoneTable: RemoteMegaphoneTable = RemoteMegap
val pendingPniSignatureMessageTable: PendingPniSignatureMess
val callTable: CallTable = CallTable(context, this)
val kyberPreKeyTable: KyberPreKeyTable = KyberPreKeyTable(co
val callLinkTable: CallLinkTable = CallLinkTable(context, th
```

The `onCreate` method creates the tables if they don't exist.

I'll need to dig into the code further to see the exact places where it happens, but I believe whenever the TM SGNL app receives or sends Signal messages, they get inserted into this database.

DataGrabber

Here is the `setMessage` method in the `DataGrabber` class:

```
public synchronized void setMessage(MessageDetailsArchive mess
    Log.d("info", "setTextMessage start");
    String lasttime = getLastMessageByType(this.mContext, Mess
    Long.valueOf(lasttime).longValue();
    ContentValues contentValues = prepareValues(messageDetails
    contentValues.put("type", MessageType.SMS.name());
    Uri path = this.mContext.getContentResolver().insert(Messa
    Log.d("grabber", "insert message and id and time , id: " +
    SharedPreferences.Editor editor = PreferenceManager.getDef
    editor.putString(MessageType.SMS.name() + DATE_OF_MESSAGE_
    Log.d("info", "setTextMessage end");
    CommonUtils.startBackupService(this.mContext);
}
```

Notice that even though we're storing Signal messages, this line explicitly sets the message type to SMS, which will be important later:

```
contentValues.put("type", MessageType.SMS.name());
```

This line of code inserts the message into the staging database:

```
Uri path = this.mContext.getContentResolver().insert(MessageCo
```

And this line of code triggers the `SyncAdapter` to run:

```
CommonUtils.startBackupService(this.mContext);
```

ArchiveMessagesProcessor

This is a processor in the `messageStoreObserver`. Whenever a message in `SignalDatabase` is created/modified, the `processAfterMessageStateChanged` method gets called:

```
@Override // com.tm.androidcopysdk.device.MessageStoreProcesso
protected void processAfterMessageStateChanged(@NotNull Archiv
    Intrinsics.checkNotNullParameter(message, "message");
    String cleanAccountPhoneNumber = message.getCleanAccountPh
    if (cleanAccountPhoneNumber == null || cleanAccountPhoneNu
        Log.d(getTag(), "ignoring archive message " + message
        return;
    }
    boolean isNewEdit = message.isNewEdit(existing);
    if (!isArchivingSupported(message, isNewEdit) || message.g
        Log.d(getTag(), "ignoring unsupported message " + mess
    } else if (message.hasDeletions()) {
        archiveDeletedMessage(message, existing);
    } else if (isNewEdit) {
        archiveEditMessage(message, existing);
    } else {
        ArchiveMessageType type = message.getType();
        switch (type == null ? -1 : WhenMappings.$EnumSwitchMa
            case 1:
                archiveMessage(message, existing);
                return;
            case 2:
                archiveMmsMessage(message, existing);
                return;
            case 3:
                archiveCallMessage(message, existing);
                return;
            default:
                Log.w(getTag(), "not sure how to handle " + me
                return;
        }
    }
}
```

As you can see, this code will archive messages when they get deleted (`archiveDeletedMessage`), when they get edited (`archiveEditMessage`), as well as when they get created (`archiveMessage`).

Here's the code in `archiveMessage`:

```
private final void archiveMessage(ArchiveMessage message, Arch
    if (existing != null) {
        Log.d(getTag(), "message " + message.getArchiveId() +
    } else if (message.getStatus() == MessageStatus.Sending &&
        Log.d(getTag(), "message " + message.getArchiveId() +
    } else {
        MessageDetailsArchive details = this.detailsConverter
        Log.d(getTag(), "archiveMessage " + message + ' ' + de
        this.module.getDataGrabber().setMessage(details);
    }
}
```

When a message is archived, it runs `setMessage` on `DataGrabber`, which, as described above, inserts the message into the staging database and triggers the sync adapter.

SyncAdapter

The `SyncAdapter`'s `onPerformSync` method is too long to quote in full here, so I will just quote pieces of it at a time. This is the method that Android will call in the background at regular intervals, or that the app itself can trigger, such as when a new message comes in.

Near the beginning of the method, this code creates a `NetworkManager` object, which is the class used to communicate with TeleMessage's archive server.

```
String baseurl = bundle.getString("baseurl");
String keeperUrl = PreferenceManager.getDefaultSharedPreferen
if (!TextUtils.isEmpty(keeperUrl)) {
    baseurl = keeperUrl;
}
NetworkManager nm = new NetworkManager(this.mContext, baseurl
String myNumber = FlavorSettings.getInstance().getMSISDN(Pref
```

When it creates the `NetworkManager` object, it passes in the base URL. How it determines this URL is a bit confusing. But ultimately, if `keeperUrl` is not empty, then it uses that value as the base URL. As I will describe below, the `keeperUrl` value comes from `ArchiveConstants`:

```
const val prodKeeper = "https://archive.telemessage.com"
```

It also sets `myNumber` to be the current user's phone number.

Next in `SyncAdapter`'s `onPerformSync` method, this code makes a SQL query to the staging database, selecting all messages that are waiting to be delivered (for the sake of easier reading, I've added some newlines to the code displayed here, but I haven't modified any of it):

```
long installation_date = PrefManager.getLongPref(
    this.mContext.getApplicationContext(),
    PrefManagerConstants.SHARED_PREFERENCE_INSTALLATION_DATE_
    PreferenceManager.getDefaultSharedPreferences(this.mConte
);
Log.d(TAG, "installation_date: " + installation_date);
Log.d(TAG, "baseurl: " + baseurl);
String[] condition = {
    String.valueOf(installation_date),
    MessageContentProvider.MessageDeliveryStatus.WaitingToBeD
};
Cursor cur = this.mContentResolver.query(
    MessageContentProvider.CONTENT_URI,
    null,
    "date >= ? AND status = ? ",
    condition,
    "date DESC"
```

```
);
Log.d(TAG, "count: " + cur.getCount());
boolean z = 0 != (this.mContext.getApplicationInfo().flags &
cur.moveToFirst();
```

First, it pulls `installation_date` from the app preferences – this was the timestamp that TM SGNL was installed.

Then it defines `condition` to be an array with the first item a string version of the `installation_date` timestamp, and the second item the message delivery status `WaitingToBeDelivered`.

Then, using the `ContentResolver`, it queries the staging database to select all messages where the date is greater than or equal to `installation_date`, and the status is `WaitingToBeDelivered`, with the results sorted by date in descending order.

Next in `onPerformSync`, this code starts looping through every message found in the SQL results, submitting them to the archive server:

```
do {
    String typestr = cur.getString(cur.getColumnIndex("type")
    if (MessageType.valueOf(typestr) == MessageType.SMS) {
        BodyBase sr = Packager.packENATextMessage(cur, myNumbe
        DBKeepAliveQueryHelper.updateSendToServerTime(this.mCo
        Log.d("network", "send message:" + ((TelemessageArchiv
        Response<Void> res = nm.start(sr, this, getContext().g
        Log.d("network", "after sent object");
```

The code checks if the message type is `MessageType.SMS` here, but as I described in the `DataGrabber` section above, Signal messages are treated as SMS messages.

It then creates a variable `sr` which contains the current message row from the SQL query, along with the user's phone number. In the following two lines, notice that it typecasts `sr` to `TelemessageArchiverMessage`, meaning that `sr` is of type `TelemessageArchiverMessage`.

Finally, it runs `nm.start` in the `NetworkManager` object, passing in the `sr` variable with the message data. This is the where the actual HTTP request to the base URL (the TeleMessage archive server) happens.

NetworkManager

Let's take a quick peak over in `NetworkManager.start`:

```
public <T> retrofit2.Response<T> start(BodyBase message, Handl
    this.mListener = listener;
    Log.d("network", "started api call");
    INetworkProvider networkProvider = context.getApplication(
    TMCredentialsStore credentialsStore = TMCredentialsStore.g
    networkProvider.headersInterceptor().setAuthentication(cre
    AndroidCopyServerAPI TMAndroidCopyAPI = (AndroidCopyServer
```

```
Call caller = null;
if (message instanceof TeleMessageMMSArchive) {
    if (!CommonUtils.isUserArchive(context)) {
        caller = TMAndroidCopyAPI.postTelemessageMMSMessag
    }
} else if (message instanceof TelemessageArchiverMessage)
    if (!CommonUtils.isUserArchive(context)) {
        networkProvider.headersInterceptor().setMessageId
        caller = TMAndroidCopyAPI.postSMSMessage(NetworkMa
    }
} else if (message instanceof SMSMessageRecord) {
```

This is the code that makes an API call. There's a big if/else block that checks for different types of messages. If the message type is `TelemessageArchiverMessage` (which it is for these Signal messages), it runs this line of code:

```
caller = TMAndroidCopyAPI.postSMSMessage(NetworkManagerAPIUti
```

If you look at `AndroidCopyServerAPI.postSMSMessage`, you'll see that it makes a POST request to `https://archive.telemessage.com/api/rest/archive/telemessageincomingmessage/`, with the message content in the body:

```
@POST("api/rest/archive/telemessageincomingmessage")
Call<Void> postSMSMessage(@Body SMSMessageRecord sMSMessageRe
```

To be clear, this is where TM SGNL sends the plaintext, already decrypted Signal message to TeleMessage's archive server.

Tying it all together

That was a lot. Now I'm going to tie it all together from the beginning, the `TeleMessageSignalApplication` object, which is created when the app is created.

Here's the `onCreate` method, which is called when the app starts:

```
override fun onCreate() {
  super.onCreate()
  Log.createInstance(applicationContext)
  ArchiveLogger.sendArchiveLog("TeleMessage logger created")

  initializeSdk()
  initArchiveUrlsAndStartArchive()
}
```

This runs `initializeSdk`, followed by `initArchiveUrlsAndStartArchive`. Here's `initializeSdk`:

```
private fun initializeSdk() {
  val module = getSdkModule(requireNotNull(SignalDatabase.inst
```

```
    val messageObserver = TeleMessageApplicationDependencyProvi
    messageObserver.addProcessor(ArchiveMessagesProcessor(modul
    messageObserver.addProcessor(SendSignatureProcessor(module)
    messageObserver.initialize(module)
}
```

This initializes the SDK module, passing in the `SignalDatabase` instance, which initializes that too.

Then it creates the message observer, and adds the `ArchiveMessagesProcessor` as a processor, passing in the SDK module (and hence, the `SignalDatabase`).

Now whenever messages get added to `SignalDatabase`, the following will happen:

- `ArchiveMessageProcessor.processAfterMessageStateChanged` will run, calling `DataGrabber.setMessage`.
- `DataGrabber.setMessage` will save the message in the staging database and trigger the sync adapter to run.
- `SyncAdapter.onPerformSync` will select the message from the staging database and pass it into `NetworkManager.start`.
- `NetworkManager.start` will send the message to TeleMessage's archive server at `https://archive.telemessage.com`.

Next, the `TeleMessageSignalApplication`'s `onCreate` method calls `initArchiveUrlsAndStartArchive`. You can read the code for that method here, but the important part is where it it sets the URLs:

```
CommonUtils.setUrl(context, ArchiveConstants.charlieProductio
```

These values are defined in `ArchiveConstants`:

```
const val charlieProduction = "https://rest.telemessage.com"
const val prodKeeper = "https://archive.telemessage.com"
```

So basically, I stated above, `prodKeeper` is set to `https://archive.telemessage.com`, and this is ultimately where chat logs get sent. And finally, this method starts the sync adapter:

```
CommonUtils.startBackupService(context)
```

And then the app starts, with Signal messages getting synced to the archive server in the background.

Conclusion

There's still a lot of this codebase that I haven't fully wrapped my mind around

But after all of this analysis, it sure looked to me like TM SGNL is simply submitting plaintext chat logs to the TeleMessage archive server, completely breaking Signal's E2EE.

Still, I've only manually analyzed the source code. I haven't tried running it, much less setting up a dummy archive server to see what data the app tries to send (something that is worth trying, but that I haven't had nearly enough time to implement).

But on Saturday night, a hacker exploited a vulnerability in the archive server and exfiltrated plaintext messages from it, including Signal messages, as I reported in 404 Media. This confirms my findings that communication between the TM SGNL app and the archive destination is not end-to-end encrypted, and that the archive server has access to plaintext chat logs.

Corroborative evidence from the hack

As Joseph Cox and I reported, a hacker was able to obtain "snapshots of data passing through TeleMessage's servers at a point in time." The snapshots contained fragments of data, including chat logs in JSON format, that was in the archive server's memory at the moment it was exploited.

To prove that the archive server has plaintext access to chat logs, I'm sharing redacted samples of some of that data here. Below is a plaintext Signal message, a plaintext Telegram message, and a plaintext WhatsApp message.

I'm also sharing an encrypted WhatsApp message. As I hinted at earlier in this post, some of the chat logs have encrypted content, and I have not yet uncovered how that works.

The snapshots of data from my source contained fragments of WeChat messages, but no complete WeChat messages, so I don't have an example of those to show.

Finally, I'm also sharing a redacted screenshot of a private key I found in the snapshots of data from the archive server.

A plaintext Signal message

We mentioned a Signal message that was present in one of those snapshots:

> A message sent to a group chat called “Upstanding Citizens Brigade”

> included in the hacked data says its “source type” is “Signal,” indicating it came from TeleMessage’s modified version of the messaging app. The message itself was a link to this tweet posted on Sunday which is a clip of an NBC Meet the Press interview with President Trump about his memecoin. The hacked data includes phone numbers that were part of the group chat.

Here's the JSON object version of that Signal message. I have redacted the phone numbers in this JSON object by replacing them with `==redacted==`.


```
{
    "typ": "RawMessage",
    "gatewayReceivedDate": 1746387273449,
    "partner": "NONE",
    "securityContent": null,
    "sourceService": null,
    "internalSecurityData": {
        "version": "0.0.2",
        "internalDecryptionData": {
            "typ": "nothing",
            "encryptionType": "DO_NOTHING",
            "params": {}
        }
    },
    "networkType": "SIGNAL",
    "sourceType": "SIGNAL",
    "ownerExtClassId": null,
    "body": {
        "owner": {
            "value": "==redacted==",
            "type": "PHONE"
        },
        "messageId": "==redacted==",
        "messageType": "APP_MESSAGE",
        "messageTime": 1746387273000,
        "sender": {
            "value": "==redacted==",
            "type": "PHONE"
        },
        "recipients": [
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            }

        ],
```

```
        "direction": "IN",
        "subject": "Signal message from ==redacted== to chat g
        "textField": {
            "extractor": {
                "typ": "WrapperExt",
                "data": "https://x.com/degeneratenews/status/1
            },
            "length": 60
        },
        "attachment": [
            {
                "name": "C2-C233F59B-FEC7-4A78-90E0-F29342B15(
                "contentType": "image/jpeg",
                "digest": null,
                "content": "<Kafkafied>:/shared/cloud/apigateu
                "attachmentSize": {
                    "attachmentSizeType": "BASE64",
                    "sizeInBytes": 72936
                },
                "attachmentDate": null
            }
        ],
        "messageStatus": "NA",
        "callInfo": null,
        "partner": null,
        "groupData": {
            "name": "Upstanding Citizens Brigade",
            "id": "",
            "type": "BROADCAST"
        },
        "threadID": "tm-1441784229",
        "threadName": null,
        "subUserId": 0,
        "participantEnrichments": {},
        "originalMessageData": null,
        "ban": null,
        "acceptedPayloadIdentifier": "8b6ab08e-edef-4d44-a0fe-
        "groupName": "Upstanding Citizens Brigade",
        "groupId": "",
        "groupMessage": false,
        "text": "https://x.com/degeneratenews/status/19191095(
    },
    "kafkafied": true
}
```


This plaintext chat log contains the following information, extracted from the archive server:

- `networkType` and `sourceType` are both `SIGNAL`
- `messageTime` is `1746387273000`, a Unix timestamp that corresponds to Sunday, May 4th, 2025, at 3:34pm Eastern time
- It includes a phone number for the `owner` and `sender`, along with the 5 `recipients` in the Signal group
- The email `subject` line, `Signal message from ==redacted== to chat group Upstanding Citizens Brigade`
- `groupName` of `Upstanding Citizens Brigade`
- `text` of `https://x.com/degeneratenews/status/1919109508120371209?s=46`

Clearly, TeleMessage's archive server has plaintext access to Signal messages.

A plaintext Telegram message

In our reporting, we mentioned that some of the data exfiltrated from the archive server appears to belong to Coinbase. Here's an example of a redacted plaintext Telegram message, apparently from a Coinbase employee.


```
{
    "typ": "RawMessage",
    "gatewayReceivedDate": 1746322842371,
    "partner": "NONE",
    "securityContent": null,
    "sourceService": null,
    "internalSecurityData": {
        "version": "0.0.2",
        "internalDecryptionData": {
            "typ": "nothing",
            "encryptionType": "DO_NOTHING",
            "params": {}
        }
    },
    "networkType": "TELEGRAM",
    "sourceType": "TELEGRAM",
    "ownerExtClassId": null,
    "body": {
        "owner": {
            "value": "==redacted==",
            "type": "PHONE"
        },
        "messageId": "EDIT-1746100232_7221627375_0-1",
        "messageType": "APP_MESSAGE",
        "messageTime": 1746114419000,
        "sender": {
            "value": "UNKNOWN",
            "type": "ALPHANUMERIC"
        },
        "recipients": [
            {
                "value": "==redacted==",
                "type": "PHONE"
            }
        ],
        "direction": "IN",
        "subject": "Telegram message from [Intl Ex] Elk (Falc
        "textField": {
            "extractor": {
                "typ": "WrapperExt",
                "data": "Hi @==redacted== please find the late
            },
            "length": 211
        },
        "attachment": null,
        "messageStatus": "NA",
        "callInfo": null,
        "partner": null,
        "groupData": {
            "name": "",
            "id": "",
```

```
            "type": "CHAT"
        },
        "threadID": "815213310",
        "threadName": null,
        "subUserId": 0,
        "participantEnrichments": {
            "{\"value\":\"UNKNOWN\",\"type\":\"ALPHANUMERIC\"}
                "firstName": "==redacted==",
                "lastName": "==redacted=="
            }
        },
        "originalMessageData": null,
        "ban": null,
        "acceptedPayloadIdentifier": "f298b776-d5d4-42da-89b3-
        "groupName": "",
        "groupId": "",
        "groupMessage": false,
        "text": "Hi @==redacted== please find the latest repo
    },
    "kafkafied": true
}
```


This plaintext chat log contains the following information, extracted from the archive server:

- `networkType` and `sourceType` are both `TELEGRAM`
- `messageTime` is `1746114419000`, a Unix timestamp that corresponds to May 1, 2025 at 11:46 AM Eastern time (I'm honestly not sure why a message from May 1 was in the archive server's memory at this moment)
- It includes the phone number of the sender under `owner`
- The email `subject` line, `Telegram message from [Intl Ex] Elk (Falcon Capital) - Coinbase to channel [Intl Ex] Elk (Falcon Capital) - Coinbase`
- `text` of `Hi @==redacted== please find the latest report: https://coinbase.sendsafely.com/receive/?==redacted==` (this message included the link to a shared document on Coinbases's SendSafely account, which I have redacted)

A plaintext WhatsApp message

I don't know much about this WhatsApp message, or the group it was sent to. But here's a redacted sample:


```
{
    "typ": "RawMessage",
    "gatewayReceivedDate": 1746319698418,
    "partner": "NONE",
    "securityContent": null,
    "sourceService": null,
    "internalSecurityData": {
        "version": "0.0.2",
        "internalDecryptionData": {
            "typ": "nothing",
            "encryptionType": "DO_NOTHING",
            "params": {}
        }
    },
```

```
    "networkType": "WHATSAPP_CLOUD_ARCHIVER",
    "sourceType": "WHATSAPP_CLOUD_ARCHIVER",
    "ownerExtClassId": null,
    "body": {
        "owner": {
            "value": "==redacted==",
            "type": "PHONE"
        },
        "messageId": "544323f6830025dd76a3",
        "messageType": "APP_MESSAGE",
        "messageTime": 1746319698000,
        "sender": {
            "value": "==redacted==",
            "type": "PHONE"
        },
        "recipients": [
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            },
            {
                "value": "==redacted==",
                "type": "PHONE"
            }
        ],
```

```
        ],
        "direction": "IN",
        "subject": "WhatsApp message from ==redacted== to Yen
        "textField": {
            "extractor": {
                "typ": "WrapperExt",
                "data": "And u look so pretty Dani"
            },
            "length": 25
        },
        "attachment": null,
        "messageStatus": "NA",
        "callInfo": null,
        "partner": null,
        "groupData": {
            "name": "Yenta AF",
            "id": "17862479908-1457977547@g.us",
            "type": "CHAT"
        },
        "threadID": "tm-643721925",
        "threadName": "Yenta AF",
        "subUserId": 0,
        "participantEnrichments": {
            "{\"value\":\"==redacted==\",\"type\":\"PHONE\"}"
                "firstName": "==redacted==",
                "lastName": "==redacted=="
            },
            "{\"value\":\"==redacted==\",\"type\":\"PHONE\"}"
                "firstName": "==redacted==",
                "lastName": "==redacted=="
            },
            "{\"value\":\"==redacted==\",\"type\":\"PHONE\"}"
                "firstName": "==redacted==",
                "lastName": "==redacted=="
            },
            "{\"value\":\"==redacted==\",\"type\":\"PHONE\"}"
                "firstName": "==redacted==",
                "lastName": "==redacted=="
            },
            "{\"value\":\"==redacted==\",\"type\":\"PHONE\"}"
                "firstName": "==redacted==",
                "lastName": "==redacted=="
            },
            "{\"value\":\"==redacted==\",\"type\":\"PHONE\"}"
                "firstName": "==redacted==",
                "lastName": "==redacted=="
            },
            "{\"value\":\"==redacted==\",\"type\":\"PHONE\"}"
                "firstName": "==redacted==",
                "lastName": "==redacted=="
            },
            "{\"value\":\"==redacted==\",\"type\":\"PHONE\"}"
                "firstName": "==redacted==",
                "lastName": "==redacted=="
            },
            "{\"value\":\"==redacted==\",\"type\":\"PHONE\"}"
                "firstName": "==redacted==",
                "lastName": "==redacted=="
            },
            "{\"value\":\"==redacted==\",\"type\":\"PHONE\"}"
                "firstName": "==redacted==",
                "lastName": "==redacted=="
            },
            "{\"value\":\"==redacted==\",\"type\":\"PHONE\"}"
```

```
                "firstName": "==redacted==",
                "lastName": "==redacted=="
            },
            "{\"value\":\"==redacted==\",\"type\":\"PHONE\"}"
                "firstName": "==redacted==",
                "lastName": "==redacted=="
            },
            "{\"value\":\"==redacted==\",\"type\":\"PHONE\"}"
                "firstName": "==redacted==",
                "lastName": "==redacted=="
            }
        },
        "originalMessageData": null,
        "ban": null,
        "acceptedPayloadIdentifier": "62db9b07-89c7-46a0-853b-
        "groupName": "Yenta AF",
        "groupId": "17862479908-1457977547@g.us",
        "groupMessage": true,
        "text": "And u look so pretty Dani"
    },
    "kafkafied": true
}
```


This plaintext chat log contains the following information extracted from the archive server:

- `networkType` and `sourceType` are both `WHATSAPP_CLOUD_ARCHIVER`
- `messageTime` is `1746319698000`, which is a Unix timestamp that corresponds to May 3, 2025 at 8:48 PM Eastern time
- `sender`'s phone number
- Phone numbers and first and last names for all 12 members of the WhatsApp group
- The email `subject` line, `WhatsApp message from ==redacted== to Yenta AF chat group`
- `groupName` is `Yenta AF`
- `text` of `And u look so pretty Dani`

TeleMessage's archive server has plaintext access to at least some WhatsApp messages.

An encrypted WhatsApp message

Some of the data passing through TeleMessage's archive server is encrypted, and I haven't yet uncovered exactly why. Here's an example of an encrypted WhatsApp message.

I've redacted the plaintext metadata in this sample. I haven't redacted the encrypted session key or the ciphertext of the message.


```
{
    "typ": "RawMessage",
    "gatewayReceivedDate": 1746322853699,
    "partner": "DEFAULT",
    "securityContent": {
        "encryptionData": {
```

```
            "keyId": "08a7fe36a6ddd8bca98ef407e254ecc230ef80b0
            "encSessionKey": "y7BbPA1mpNLkFGjcVRZWTRNbf22lWvjI
        },
        "integrityHeaderContent": "7C967D5A3BD5C59284EAB8D2CE5
    },
    "sourceService": null,
    "internalSecurityData": {
        "version": "0.0.2",
        "internalDecryptionData": {
            "typ": "hybrid",
            "params": {
                "ENC_SESSION_KEY": {
                    "typ": "encskey",
                    "value": "y7BbPA1mpNLkFGjcVRZWTRNbf22lWvjI
                    "type": "ENC_SESSION_KEY"
                },
                "PUBLIC_KEY_ID": {
                    "typ": "pubkey",
                    "value": "08a7fe36a6ddd8bca98ef407e254ecc1
                    "type": "PUBLIC_KEY_ID"
                }
            },
            "encryptionType": "HYBRID"
        }
    },
    "networkType": "WHATSAPP_ARCHIVER",
    "sourceType": "WHATSAPP_ARCHIVER",
    "ownerExtClassId": null,
    "body": {
        "owner": {
            "value": "==redacted==",
            "type": "PHONE"
        },
        "messageId": "b849fa65750cbed6cf5be5cb9a69d943",
        "messageType": "APP_MESSAGE",
        "messageTime": 1746322850000,
        "sender": {
            "value": "==redacted==",
            "type": "PHONE"
        },
        "recipients": [
            {
                "value": "==redacted==",
                "type": "PHONE"
            }
        ],
        "direction": "IN",
        "subject": "STATUS - WhatsApp message from ==redacted=
        "textField": {
            "extractor": {
                "typ": "WrapperExt",
                "data": "CrlDZrXBN1jL+V7Yht6lDpx4oFHQ05M6kVM7I
            },
            "length": 88
        },
        "attachment": [
            {
                "name": "eee44412-b56f-47ff-9548-f00fc3ed1cd3
                "contentType": "image/jpeg",
                "digest": null,
                "content": "<Kafkafied>:/shared/cloud/apigatev
                "attachmentSize": {
                    "attachmentSizeType": "BASE64",
                    "sizeInBytes": 44352
```

```
            },
            "attachmentDate": null
        }
    ],
    "messageStatus": "NA",
    "callInfo": null,
    "partner": null,
    "groupData": {
        "name": "",
        "id": "status@broadcast",
        "type": "CHAT"
    },
    "threadID": "1790853601",
    "threadName": null,
    "subUserId": 0,
    "participantEnrichments": {
        "{\"value\":\"==redacted==\",\"type\":\"PHONE\"}"
            "firstName": "==redacted==",
            "lastName": "==redacted=="
        },
        "{\"value\":\"==redacted==\",\"type\":\"PHONE\"}"
            "firstName": "==redacted==",
            "lastName": "==redacted=="
        }
    },
    "originalMessageData": null,
    "ban": null,
    "acceptedPayloadIdentifier": "b7b88772-f3b7-4e20-8bc8
    "groupName": "",
    "groupId": "status@broadcast",
    "groupMessage": true,
    "text": "CrlDZrXBN1jL+V7Yht6lDpx4oFHQ05M6kVM7HjecbWY1
},
"kafkafied": true
}
```


This encrypted WhatsApp chat log is different than the plaintext one in a few ways:

- The `networkType` and `sourceType` are both `WHATSAPP_ARCHIVER`, while in the plaintext WhatsApp message they were `WHATSAPP_CLOUD_ARCHIVER`
- The encrypted message contains encryption information in `securityContent` and `internalSecurityData`, while in the plaintext WhatsApp message they didn't
- `text` is a Base64 block of encrypted text instead of plaintext message content

Even with encryption, this chat log contains the following plaintext information:

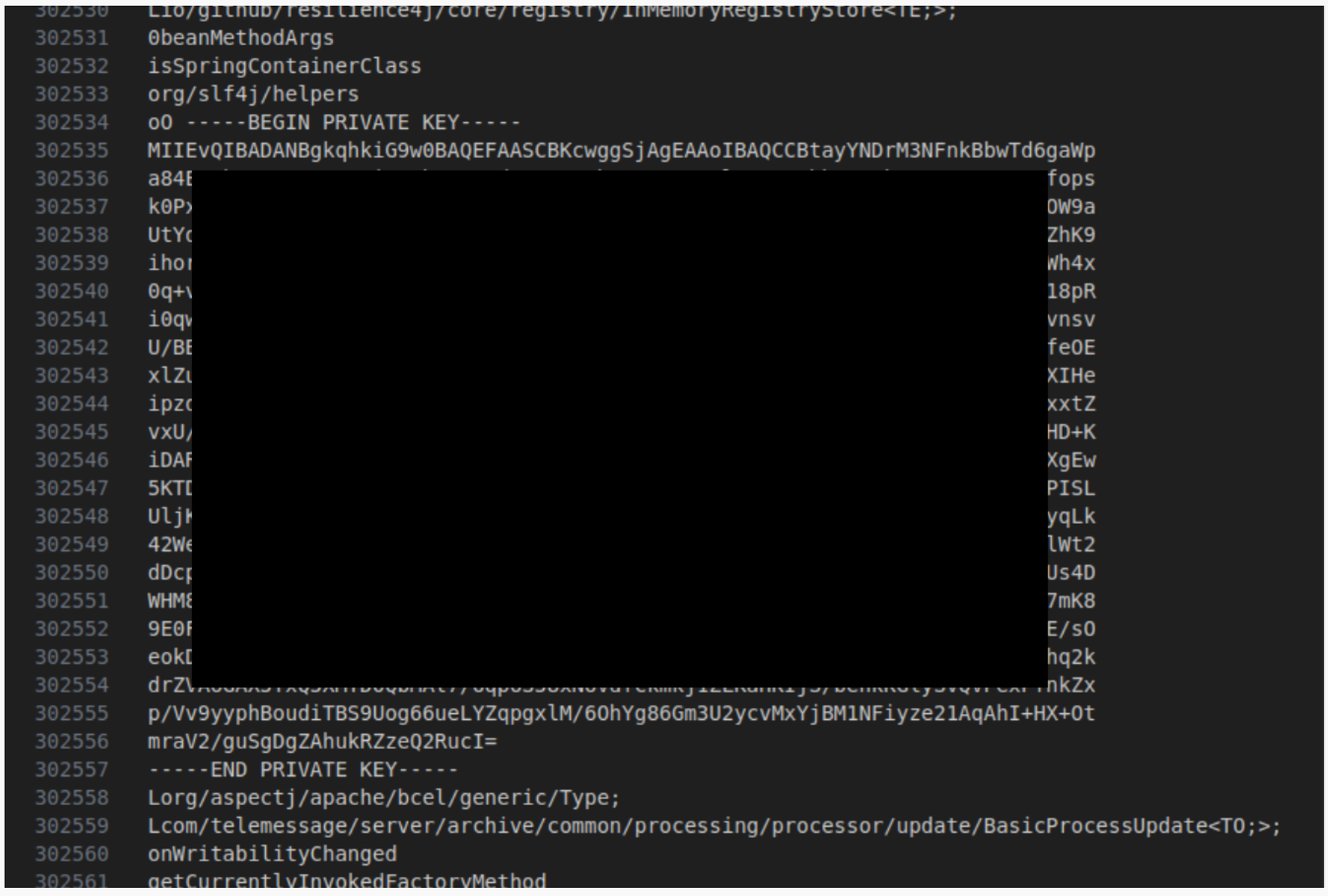
- The `sender` and `recipients` phone numbers and full names
- The email `subject` line, `STATUS - WhatsApp message from ==redacted== to ==redacted==`

- `messageTime` is `1746322850000`, a Unix timestamp that corresponds to May 3, 2025 at 9:40 PM Eastern time

Private key material

As Joseph and I mentioned in our reporting in 404 Media, the snapshots of data from the archive server contained more than just chat logs. There were usernames and plaintext passwords – the hacker used these to login to archive site, gaining access to a list of apparently 747 Customs and Border Protection employees.

But also, I found private key material in the snapshots, though I have not yet uncovered what the keys are for. Here's one of the private keys I found, redacted, of course:



A redacted private key

If you found this interesting, subscribe to get these posts emailed directly to your inbox. If you want to support my work, considering becoming a paid supporter.

1 comment

Join the discussion

Become a member of **micahflee** to start commenting.

Sign up now

Already a member? **Sign in**

Cap Dap · 8 May

Did the hacker on Saturday breach TeleMessage through HTTP requests or did they find a server vulnerability? The article seems to suggest both at the same time, especially with the 404media article claiming that the hacker accessed debug logs

0 Reply

← Previous

Next →