

Call an Exorcist! My Robot's Possessed!

Mark Berezin | AUG 05, 2020 | 69 MIN READ

Overview

As part of our continued goal of helping developers provide safer products for businesses and consumers, we here at McAfee Advanced Threat Research (ATR) recently investigated *temi*, a teleconference robot produced by Robotemi Global Ltd. Our research led us to discover four separate vulnerabilities in the temi robot, which this paper will describe in great detail. These include:

1. CVE-2020-16170 – Use of Hard-Coded Credentials
2. CVE-2020-16168 – Origin Validation Error
3. CVE-2020-16167 – Missing Authentication for Critical Function
4. CVE-2020-16169 – Authentication Bypass Using an Alternate Path of Channel

Together, these vulnerabilities could be used by a malicious actor to spy on temi's video calls, intercept calls intended for another user, and even remotely operate temi – all with zero authentication.

Per McAfee's vulnerability disclosure policy, we reported our findings to Robotemi Global Ltd. on March 5, 2020. Shortly thereafter, they responded and began an ongoing dialogue with ATR while they worked to adopt the mitigations we outlined in our disclosure report. As of July 15, 2020, these vulnerabilities have been successfully patched – mitigated in version 120 of the temi's Robox OS and all versions after 1.3.7931 of the temi Android app. We commend Robotemi for their prompt response and willingness to collaborate throughout this process. We'd go so far as to say this has been one of the most responsive, proactive, and efficient vendors McAfee has had the pleasure of working with.

This paper is intended as a long-form technical analysis of the vulnerability discovery process, the exploits made possible by the vulns, and the potential impact such exploits may have. Those interested in a higher-level, less technical overview of these findings should refer to our summary blog post [here](#).

Contents

[Overview](#)
[Contents](#)
[What is temi](#)
[Normal Operation of temi](#)
[Initial Recon](#)
[Port Scanning](#)
[Capturing Traffic](#)
[Getting a Shell](#)
[Finding temi's Code](#)
[Reversing the Code for Video Calls](#)
[Brute-Forcing the Channel Name as an Attack Vector](#)
[Exploring MQTT Attack Vectors](#)
[Overview](#)
[Modifying the temi Phone App](#)
[The Relationship Between Robot IDs and MQTT Topics](#)
[How are MQTT Call Invite Messages Published?](#)
[Intercepting Calls](#)
[Problem: temi Won't Answer My Calls](#)
[Solution: Become an OWNER](#)
[Adding an OWNER: Phone App's Perspective](#)
[Adding an OWNER: temi's Perspective](#)
[Detour: Sneaking Onto temi's Contact List](#)
[Gaining OWNER Privileges](#)
[Refining the Exploits](#)
[Impact](#)
[Conclusion](#)

What is temi?

Robots.

The final frontier.

For an Android tablet 'brain' sitting atop a 4-foot-tall robot, temi packs a lot of sensors into a small form factor. These include 360° LiDAR, three different cameras, five proximity sensors, and even an Inertial Measurement Unit (IMU) sensor, which is a sort of accelerometer + gyroscope + magnetometer all-in-one. All these work together to give temi something close to the ability to move autonomously through a space while avoiding any obstacles. If it weren't for the nefarious forces of stairs and curbs, temi would be unstoppable.



Robotemi markets its robot as being used primarily for teleconferencing. Articles linked from the temi website describe the robot's applications in various industries: Connected Living recently partnered with temi for use in elder care, the Kellogg's café in NYC adopted temi to "enhance the retail experience", and corporate staffing company Collabera uses temi to "improve cross-office communication." Despite its slogan of "personal robot", it appears that temi is designed for both consumer *and* enterprise applications, and it's the latter that really got us at McAfee Advanced Threat Research interested in it as a

patient being diagnosed via robotic proxy? We placed our preorder and set out to find out.

Normal Operation of temi

Once it finally arrived, we got to setting it up the way any user might: we unboxed it, plugged in its charging dock, and connected it to WiFi. Normal operation of the temi robot is done through the use of its smart phone app, and at first startup temi prompted us to scan a QR code with the app. The phone used to scan the QR code becomes temi's "admin", allowing you to control the robot remotely by simply calling it. Temi can have many contacts outside of its singular admin, and becoming a contact is fairly straightforward. Whenever you launch the temi phone app, it scans your phone's contacts and automatically adds any numbers that have been registered with the temi app to the app's contact list. If any of those contacts happens to be a temi admin, you can call their temi simply by clicking that contact, as shown in [Figure 1](#).

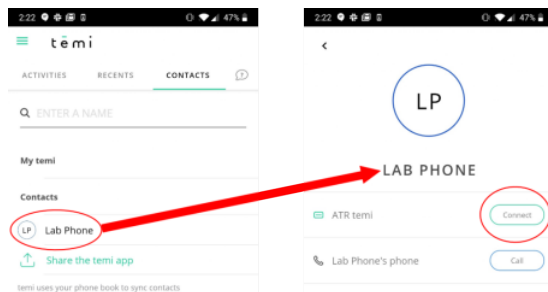


Figure 1: Selecting a contact from the temi phone app

In this way, users of the phone app besides temi's admin can still call a temi robot using this method. Since initiating a call with a temi allows you to remotely operate it in addition to seeing through its camera and hearing through its microphone, giving anyone the ability to call your temi could prove... problematic. Temi's creators address this exact issue on their FAQ page, as shown in [Figure 2](#).

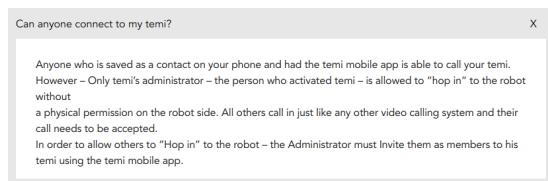


Figure 2: "Can anyone connect to my temi?", from the temi's FAQ page

The first line here is a bit misleading, since adding a temi's admin as a phone contact seems to be sufficient to call that temi – the admin does not need to also have you added as a phone contact for this to work. That being said, this doesn't mean that just anyone can start controlling your temi; "physical permission on the robot side" is required for calls made by users that aren't temi's admin or explicitly authorized by said admin. In practice, this corresponds to a call notification on the temi's screen that the user can either answer or decline. In other words, it's no more or less "secure" than receiving cold calls on your cell phone.

As for the last line, which refers to an admin's ability to grant certain users the option to "hop in" to the robot, this is also done through the phone app by simply selecting the "Invite New Member" option from the temi's contact entry in the app, and then selecting one or more users from the app's contact list to "invite", as shown in [Figure 3](#).

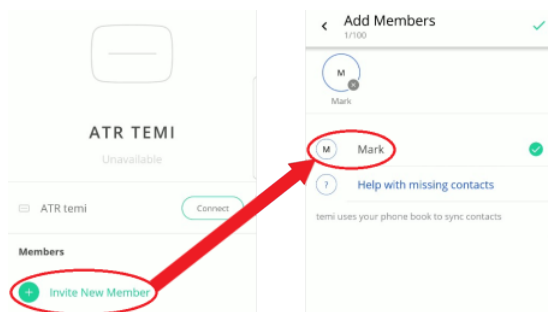


Figure 3: How to grant users special permissions from the phone app

Once a call has been established between the robot and a phone user, which either party may initiate, the phone user can do things like manually drive the robot around, add, remove, and navigate to saved locations, patrol between saved locations, control its volume, and more.



Initial Recon

Port Scanning

While a robot was certainly a bit different from our typical targets, we began our reconnaissance with methods that have stood the test of time: port scans, packet captures, and a local shell.

```
user@user-virtual-machine:~/Documents$ sudo nmap -sS 192.168.7.101

Starting Nmap 7.60 ( https://nmap.org ) at 2019-08-23 14:22 CDT
Nmap scan report for 192.168.7.101
Host is up (0.011s latency).
Not shown: 999 closed ports
PORT      STATE SERVICE
4443/tcp  open  pharos
MAC Address: 28:ED:E0:62:36:B8 (Unknown)
```

Figure 5: Running Nmap on the temi

An Nmap scan revealed only one open port: TCP port 4443. Immediately we knew that Nmap's classification of this port being used for Pharos, a "secure office printing solution for your business", was almost certainly wrong. Instead, it was likely that this port was being used for TLS communication as an alternative to the standard 443. While good to know, this didn't tell us much about the type of traffic temi expected on this port, or even what service(s) was handling that traffic, so we moved on.

Capturing Traffic

Next on our checklist was to obtain some captures of the robot's network traffic, focusing on traffic generated during boot, during an update, and during a video call.

Traffic captured when temi was booting up but otherwise idling showed three unique external IP addresses being accessed: 98.137.246.7, 54.85.186.18, and 34.206.180.208.

Running [nslookup](#) on these IPs revealed that the first pointed to some Yahoo! media server, likely used for its news app, while the other two appeared to be AWS instances.

Figure 6: Running nslookup on IPs accessed by temi

As for the data being sent/received, there wasn't much to look at. The Yahoo! address was being accessed via HTTP (port 80), but the TCP packets had no payload. As for the AWS addresses, these were being accessed via TLS (port 443) and the data was encrypted.

For updates, temi accessed "temi-ota-updates.s3-accelerate.amazonaws.com", which was almost certainly a custom AWS instance set up by the folks at Robotemi. As with the other traffic to AWS, the updates were being encrypted.

Getting a Shell

In order to prod deeper, we needed a local shell on our temi. Fortunately, wireless connections via [Android Debug Bridge, or ADB](#), could be enabled through temi's settings from its touchscreen. Better still, the shell provided through ADB had root privileges, although the device itself was not "rooted". This was likely done to help facilitate "temi Developers", since the temi website has [an entire portal dedicated to helping users develop custom apps for their robot](#).

Unfortunately, the default shell granted via ADB was fairly stripped down, and each reboot would mean having to manually reopen temi's ADB port from its touchscreen before being able to connect to it again. Furthermore, this method required the temi software to be running in order to access our shell, which would leave us without recourse if our prodding somehow bricked that software. This was far from ideal.

While commands like [adb push](#) made moving custom ARM binaries for bash, [busybox](#), and ssh onto temi fairly trivial, getting anything to run on boot proved more challenging. For starters, temi did not have the standard `/etc/init.d/` directory. Additionally, the primary scripts that would run on boot, like `/init.rc`, would be loaded directly from the boot image, meaning any edits made to them would not persist across reboots. This was likely by design as part of Android's SELinux security model – we would have to be a little more creative if we wanted to convince temi to start an SSH server on boot.

Digging through the contents of `/init.rc`, however, gave us a clue:

Figure 7: Interesting entry in `/init.rc`

For those unfamiliar with Android's Init language,

"The init language is used in plain text files that take the `.rc` file extension. There are typically multiple of these in multiple locations on the system, described below. `/init.rc` is the primary `.rc` file and is loaded by the init executable at the beginning of its execution. It is responsible for the initial set up of the system." – [Android docs](#).

The entry shown in [Figure 7](#), one of the hundreds contained in temi's `init.rc`, tells the system to launch the service "flash_recovery" during boot with the argument `"/system/bin/install-recovery.sh"`. The "class main" is just a label used to group entries together, and "oneshot" just means "do not restart the service when it exits." This entry stood out to us because it was invoking a script located in `/system/bin/`, which was **not** loaded from the boot image, meaning any changes should stick. While the `/system` partition is read-only by default, our root privileges made it trivial to temporarily

Finding temi's Code

The first thing we did with our newfound freedom was run netstat:

Figure 8: Running netstat

It appeared that most networking, including the open port 4443 we found using Nmap earlier, was being handled by "com.roboteam.teamy.usa". Based on the name alone, this was likely the main temi software running on the robot. Additionally, the name looked more like the name of an Android package than a native binary. We confirmed this by running:

Figure 9: Trying to find the binary for com.roboteam.teamy.usa in /proc

app_process32 (or app_process64, depending on architecture), is the binary used by Android for running Android apps, meaning we were looking for an APK, not an ELF. Under this assumption, we instead tried to find the code for this process using Android's [pm](#) command:

Figure 10: Looking for the APK for "com.roboteam.teamy.usa"

Sure enough, we had our APK.

Typically, every installed app on Android has a corresponding data directory, and we were able to find the one for this package under /data/data/com.roboteam.teamy.usa:

Figure 11: data directory for "com.roboteam.teamy.usa"

The lib/ directory contained native code used by the package:

Figure 12: native code used by "com.roboteam.teamy.usa"

After looking into several of these libraries, libagora-rtc-sdk-jni.so in particular stood out to us since it was part of [Agora](#), a platform that provides SDKs for voice, video, and Real-Time-Messaging (RTM) services. It was likely that temi was using Agora to implement its video calling functionality – the functionality we were most interested in.

By looking at this binary's strings, we were also able to determine that temi was using version 2.3.1 of the Agora Video SDK:

Figure 13: Finding the Agora SDK version using strings

Of course, we were equally interested in the code for the temi phone app, which we obtained through the use of [APK Extractor](#) and ADB. We were also curious to see if the Android phone app used the same version of the Agora SDK, which we checked by comparing their MD5 hashes:

Figure 14: Comparing MD5 hashes for libagora between temi robot and temi phone app

Sure enough, they were the identical.

Reversing the Code for Video Calls

The next step was to begin reversing these apps in order to better understand how they worked. We decided to start by looking at the phone app's code, since it was easier to test the behavior of the phone app compared to the robot.

automatically)

- It handles modern Java features like nested classes and inline lambda functions better than most other decompilers
- For any class being viewed, you can click on the “small” tab to see the small code
- It displays line numbers synchronized with the corresponding .line directives in the bytecode, making it easier to map decompiled Java code back to the original bytecode
- You can right-click on any method, member, or class to see its usage or declaration

The temi Android phone app, like most non-trivial Android apps, was massive. Instead of groping in the dark and hoping to stumble upon some interesting code, we decided to take a more targeted approach. From the outset, we knew that our focus was on temi’s calling functionality since this would provide the greatest impact for any attacker. We also now knew that temi was using the Agora SDK to implement this calling functionality. After looking through [Agora’s Java API Reference for Android \(v2.3.1\)](#), we decided that the function `joinChannel()` would be a good place to start our investigation:

Figure 15: Agora’s documentation for `joinChannel()`

By opening `libagora-rtc-sdk-jni.so` in IDA and looking at its exports, we were able to find a function called `nativeJoinChannel()` at the offset `0xD4484`:

Figure 16: Some of libagora’s exports

Using JADX’s search feature, we found a function with the same name in the decompiled code, located on line 960 in the class `RtcEngineImpl`:

Figure 17: Native Agora methods in `RtcEngineImpl`

From here, we began the tedious process of working backwards to trace the call chain for video calls all the way to their entry points. We did this by looking for the method that invoked `nativeJoinChannel()`, then looking for the method that invoked that method, and so on. The return on investment for our tireless efforts was the following:

Figure 18: Code flow diagram for the various ways of making video calls using the temi phone app

At a high level, the code for outgoing calls has four entry-points, and these map 1-to-1 with the four ways of initiating a call from the temi phone app:

1. Select a contact and hit “Call”. This will call that contact’s phone directly, not their temi. This corresponds to the “Contact Details → Contact Call” code path outlined in the graph (green region).
2. Select a contact and, if they have a temi robot, hit “Connect”. This will call their temi and corresponds to the “Contact Details → Robot Call” code path outlined in the graph (orange region).
3. Go to the app’s “Recents” tab and select one of the contacts/robots you have recently called or have called you. This will call that contact or robot and corresponds to the “Recent Calls → Call” code path outlined in the graph (blue region).
4. If you are an admin, select your temi from beneath the “My temi” heading on the “Contacts” tab and hit the “Connect” button from the following screen, also known as the Robot Details screen. This will call your temi and corresponds to the “Robot Details → Call” code path outlined in the graph (red region).

The classes and methods contained within the colored regions handle rendering the screens from which calls can be initiated and the binding of these buttons to the code that actually handles calling. No matter the entry point, all outgoing calls converge at `TelepresenceService.initiateCall()`, so it’s worth taking a closer look at this method:

Figure 19: TelepresenceService.initiateCall()

And here are the four ways it is invoked:

Contact Details → Contact Call

Figure 20: Invoking initiateCall() from Contact Details → Contact Call

Contact Details → Robot Call

Figure 21: Invoking initiateCall() from Contact Details → Robot Call

Recent Calls → Call

Figure 22: Invoking initiateCall() from Recent Calls → Call

Robot Details → Call

Figure 23: Invoking initiateCall() from Robot Details → Call

The first parameter is an ID used to identify the callee. For temi robots, this appears to be something called a "robot ID", whereas for contacts it is obtained via a call to `getContactId()`. Interestingly, for recent calls, which can be either a contact or a robot, the ID is obtained via a call to `getMd5PhoneNumber()`, which is a bit strange since temi does not (as far as we can tell) have a phone number. We will expand on this later.

The second parameter is a string denoting the caller's type. Since we were looking at exclusively phone app code here, we assumed that "typeUser" denotes a caller using the phone app.

The third parameter is the callee's display name – straightforward enough.

The fourth and final parameter is of a custom enum type that denotes the *callee's* type. If calling a temi, its type is `CallContactType.TEMI_CALL`; otherwise, it's `CallContactType.CONTACT_CALL`. Note that for the Recent Calls entry point, this value is obtained dynamically, which makes sense since this path handles calls to both contacts and temis.

It's also worth noting that `initiateCall()` invokes `InvitationManager.createInvitation()` in order to create an invitation object which represents the "call invitation" to be sent to the callee:

Figure 24: InvitationManager.createInvitation()

From there, as seen on line 223 of [Figure 19](#), `initiateCall()` passes the result of `Utils.getRandomSessionId()` as the first parameter to `createInvitation()`, which becomes the `sessionId`:

Figure 25: Utils.getRandomSessionId()

All this is doing is randomly generating an integer between 100,000 and 999,999, inclusive. In other words, the `sessionId` is always a positive, six-digit decimal number.

By tracing this value all the way down the call chain to Agora's `nativeJoinChannel()`, we discovered that this `sessionId` becomes the "channelName" parameter described in [Figure 15](#) – the unique identifier for the chatroom for that particular call. The propagation of this channel name can be seen in [Figure 18](#); the **bolded** parameter in each method call contains the channel name.

Brute-Forcing the Channel Name as an Attack Vector

So why were we so interested in the Agora channel name? Well, if we look back at [Figure 15](#), we can see that it is one of only two fields needed to join a channel since the third and fourth fields are labeled optional. The only other required field is the "token", but looking closer at the documentation reveals that "in most circumstances, the static App ID suffices", and in those cases the token "is optional and can be set as null":

Figure 26: Agora documentation for joinChannel's "token" parameter

Our next question was, does temi use a token or does it use a static App ID? We quickly answered that question by looking at how the temi phone app was calling `joinChannel()`:

This was starting to look promising – if the static App ID and channel name are all that's needed to join an Agora video call and we already knew that temi's channel names are restricted to six-digit values, then it might be plausible to join existing temi calls through brute force means. All we would need is this "App ID".

Looking back at the Agora docs, we looked for which API functions actually use this App ID. As it turned out, there was only one: `RtcEngine.create()`.

Figure 28: Agora documentation for `RtcEngine.create()`

Put briefly, the App ID is a static value issued to developers that is unique to each project and is used as a sort of namespace, segregating different users of Agora's servers. This ensures that temi users can only use Agora to call other temi users. Since any temi phone app user should be able to call any temi (or other user), there should be a single App ID shared by all temi robots. We decided to take a look at how temi's code was invoking `RtcEngine.create()` to see if we could track down the App ID:

Figure 29: `AgoraEngine.ensureRtcEngineReadyLock()`

Well, that's not good.

In our view, this was already a vulnerability, denoted by CVE-2020-16170 and having a CVSS score of 8.2. A dedicated attacker would have no problem iterating over all 900,000 possible channel names. Worse yet, doing so would let the attacker "spray and pray", allowing them to connect to any ongoing temi call without needing to know anything about their victims.

While we certainly couldn't test such an exploit against a production server, we did decide to test whether an attacker could join an existing temi call knowing only the App ID and channel name in advance. To facilitate this, we used an Android phone to call our temi, making sure to run `logcat` on the phone before the call started. Doing so, we were able to capture the invitation message containing the channel name (labeled "sessionId") as it was being sent to the `OkHttp` client, which then logged it:

Figure 30: Finding the channel name for the call using `logcat`

Using the hardcoded App ID and the channel name obtained from the logs we were able to successfully join an ongoing call and get both audio and video from at least one of the other callers, proving that this is a viable attack vector.

Although exploitation of this vulnerability utilizes Agora's video calling API, its existence is a result of temi's specific implementation of the SDK. While the decision to hardcode temi's Agora App ID into the phone app is the root cause of this vulnerability, its impact could also have been substantially mitigated by utilizing a token or by allowing a broader range of channel names. Either of these would have rendered the brute-force attack vector incredibly difficult, if not impossible.

Exploring MQTT Attack Vectors

Overview

Outside of a brute-force method involving joining existing temi calls by trying every possible channel ID, it would be useful to have a more targeted attack vector. Moreover, while joining a call using the brute-force method would let an attacker spy on users, it would not grant them control of the robot itself – we wanted a method that would let us do both. Fortunately, this level of control is already available during normal operation of the temi robot + phone app.

Although temi uses Agora to facilitate video calls, notifying users of incoming calls, through ringing or otherwise, is not a feature implemented by Agora. In the case of temi, this functionality is implemented using `MQTT`. MQTT is a publish/subscribe (pub/sub) connectivity protocol designed for "machine-to-machine (M2M)/Internet of Things" communication. In it, communications are categorized into "topics" and clients can either "subscribe" to these topics to receive all their related messages, "publish" their own messages to these topics, or both. These topics are structured into a hierarchy delineated in a manner very similar to UNIX-style filesystem naming schemes, with a forward slash ("/") separating different topic "levels". Communication between clients is facilitated via a system known as a "broker", usually implemented as a remote server, which handles establishing and closing connections, message delivery, and client authentication.

need a way to pick out Bob's temi from every other temi. A softer but equally important requirement is that this identifier must be one that an attacker could plausibly obtain. The more difficult the identifier is to obtain, the more contrived and unrealistic any attack vectors that rely on this identifier become. As an example, a name or phone number might be a plausible identifier; a social security number, less so.

Reversing temi's MQTT implementation in its phone app code had revealed a promising potential identifier. We discovered that each robot has its own MQTT topics that it listens on, which are identified using its MQTT client ID, otherwise known as its robot ID. The robot ID for a temi can be easily obtained through normal operation of the phone app by simply adding one of its users as a phone contact, a method we had described previously. This is because the temi phone app allows the user to call any temi registered to a phone number in the user's contacts list, which requires that temi's robot ID. If the phone app already has access to this information locally, then it should be hypothetically possible to pull this information from the app.

Second, we needed a way to communicate with temi. If calling is facilitated through the publishing of MQTT messages to certain topics, we needed to find a way to publish our own messages to these same topics. On that note, if all MQTT messages must first pass through the broker, we needed a way to trick the broker into thinking we were a trusted MQTT client and bypass any authentication that might be in place.

One way to overcome this hurdle would be to alter the existing temi phone app. [Modifying third party Android apps](#) is a well-known process and would let us leverage the code already present for sending MQTT messages instead of writing this code from scratch. Furthermore, since the phone app requires nothing more than the phone number of one of temi's users in order to call it (and thus, publish MQTT messages on its topics), this means that the phone app must have a way of authenticating with the broker. If we could send custom MQTT messages to arbitrary topics from the context of a "trusted" phone app, then we likely wouldn't need to worry about authentication.

This left us with our third and final hurdle: privilege escalation. Although cold calling a temi robot is not difficult to achieve, it does not grant us the ability to remotely operate the robot on its own. This is because calling temi in this way causes it to ring and requires the call to be accepted on temi's end via its touchscreen. Only temi's admin, the user who registered it via QR code scan, and privileged users manually selected by this admin may directly control temi without any user interaction on the other end. Thus, we needed to find a way to escalate our privilege so that temi would pick up our calls automatically.

If MQTT is the primary means of communication between temi and its phone app users, and admins can manage the privilege levels of users directly from the phone app, it stands to reason that privilege management is likely performed through MQTT. Thus, if we could alter the phone app to spoof a privilege escalation MQTT message, we could overcome this hurdle, as well.

Modifying the temi Phone App

Since our entire plan hinged on us being able to alter the bytecode of the phone app without either breaking the app or preventing it from authenticating with the various remote servers it communicates with, we decided to first confirm that this was even possible. To accomplish this, we used a combination of ADB, [Apktool](#), [Keytool](#), and [Jarsigner](#).

We began by unpacking the APK file for the temi phone app using Apktool:

Figure 31: Unpacking the temi phone app

Next, we searched the unpacked APK for the file or code we wanted to modify. For our proof-of-concept, we decided to simply change the label for the "Call" button, since it would be immediately obvious whether or not it worked. In many Android apps, the strings used for buttons are typically not hardcoded and are instead loaded from a resource file. In this case, they were loaded from the file `res/values/strings.xml`:

Figure 32: Searching for the call button label in strings.xml

It looked like line 108 contained the label we wanted to change. We simply replaced "Call" with "PWN" and saved our changes.

*Note: For less trivial modifications, like the ones we would need to make later, this process would naturally be more involved. Since even the most sophisticated Java decompilers are unable to produce code that will actually compile for non-trivial apps, meaningful modifications usually mean having to read and modify [smali](#), the assembler for Android's Dalvik bytecode. That being said, we found that the best approach to making meaningful changes to a complex app like temi's is to **read Java**, **write smali**. By this, we mean that it's better to do your reversing on decompiled Java code and only look at the high-hieroglyphic smali code once you know exactly what changes you want to make and what class to make it in.*

Once our modification had been made, we used Apktool again to repack the APK:

Figure 33: Repacking our modified app

Our next step was to sign the modified APK. This is because Android refuses to install any unsigned APKs, even through ADB. First, we generated a key using Keytool:

Figure 34: Generating the key we will use to sign the modified app

and then we used our key to sign the APK using Jarsigner:

Figure 35: Signing the modified app

Finally, we installed the modified APK onto an Android phone using ADB:

Figure 37: Testing the modified app

Furthermore, the change we made did not seem to impact the app's functionality; we could still make and receive calls, add contacts, etc.

In our view, this was a vulnerability, later denoted by CVE-2020-16168 and having a CVSS score of 6.5. An altered and potentially malicious app could still access the various cloud resources and user information made available to the temi app because no integrity checks were performed by either the app or the remote servers to ensure that the app had not been modified. As we'll demonstrate later, the presence of this vulnerability made various other attack vectors possible.

The Relationship Between Robot IDs and MQTT Topics

In the overview section, we made the claim that "each robot has its own MQTT topics that it listens on, which are identified using temi's MQTT client ID, otherwise known as its robot ID." Here we will outline how we came to this conclusion and the specific format of a few of the topics that temi and its phone app subscribe/publish to.

Since we knew that temi uses MQTT to handle calling notifications, a natural place to start our investigation was the code used by the phone app to initiate calls. Moving down the call chain, we saw that the robot ID was being passed to the `sendInitInvitation()` method of the `TelepresenceService` class:

Figure 38: `InvitationManagerImpl.sendInitInvitation()`

Here, the robot ID is used in two different places. On line 82, an `MqttDelegateApi.Topic` object is created with the name `"users/<ROBOT_ID>/status"`, implying that each robot has its own MQTT topic category for its status and the robot ID itself is used to uniquely identify these topics. Next, on line 87, we see one of many [RxJava](#) functions (the others have been omitted) with the robot ID passed as a parameter to it. This function's only purpose is to call `InvitationManagerImpl.sendInviteMsg()`, passing along the invitation and the robot ID as its arguments:

Figure 39: `InvitationManagerImpl.sendInviteMsg()`

This function is of particular interest because we see the construction of another MQTT topic name on line 332, this time taking the form `"client/<ROBOT_ID>/invite"`. Presumably, this is the topic format used when publishing call invitations to specific temi robots (and, likely, phone contacts).

Additionally, the anonymous function executed via `doOnSuccess()` uses `MqttManager.publish()` (line 359) to actually publish the call invitation on the callee's call invite topic. This information would become useful when we later tried to send custom MQTT messages to our temi in order to facilitate privilege escalation.

To determine what was actually being passed to `publish()`, we needed to go through what each of the RxJava functions in `InvitationManagerImpl.sendInviteMsg()` was doing. Referring back to [Figure 39](#):

- On lines 331-339, [Single.zip\(\)](#) is called, which simply creates a Pair from the MQTT topic string ("client/<ROBOT_ID>/invite") and the Invitation object.
- On lines 340-355, [Single.flatMap\(\)](#) is called, which adds a timestamp to the Invitation object inside the Pair.
- Presuming successful execution of the previous `flatMap()` call, [Single.doOnSuccess\(\)](#) is called on lines 356-372. As mentioned previously, this is where the call to `MqttManagerImpl.publish()` occurs. Since this `doOnSuccess()` operates on the value returned by the previous call to `flatMap()`, the arguments being passed to `publish()` are:

Argument	Description
(String) pair.first	The MQTT topic string
new Gson().toJson((Invitation) pair.second)	The Invitation object as JSON
0	The integer "0"
false	The boolean "false"

While it was obvious that the first argument is the MQTT topic the message is being published on and the second argument is the message itself, it was not immediately obvious what the third and fourth arguments were for. Digging down into the source code of the MQTT package being used ([org.eclipse.paho.client.mqttv3](#)) revealed their purpose:

Figure 40: `MqttAsyncClient.publish()`

After passing through a couple `MqttManagerImpl` methods, the four arguments listed above become the first four arguments passed to this internal `publish()` method. The JSON string (second argument), is converted from a string to a byte array in the interim; the rest of the arguments are unchanged.

Knowing this, it was clear that the second argument is indeed the MQTT message, the third argument is the [QoS](#), and the fourth argument is a flag that specifies whether or not the message should be [retained](#). A QoS value of "0" means that the message will be delivered to clients subscribed to the topic at most once. A retained flag of "false" means that new subscribers to the topic will not receive the most recent published message upon subscription.

Intercepting Calls

As we've already established, every temi robot has a unique MQTT client ID and many of the topics it subscribes to contain this ID to indicate that they are intended for that specific robot. If users of the temi phone app can receive calls in addition to making them, it stands to reason that they must also have a unique MQTT client ID – a robot ID equivalent. If there was an easy way to discover the client ID of another phone app user, it might be possible to subscribe to their topics and thus receive calls intended for them, making it worthy of investigation.

If we refer back to [Figure 22](#), we saw that if a call is initiated from the Recent Calls screen, a method called `getMd5PhoneNumber()` is used to obtain the client ID of the callee. While a temi doesn't have a phone number to speak of, we began to suspect that the client ID for users of the temi phone app might just be an MD5 hash of their phone number.

Although we could have followed the code in order to track down exactly where this value comes from, we thought it might be easier to simply verify our suspicions. To do this, we first took the MD5 hash of the temi admin's phone number and then performed a string search for this hash in every temi-related file we had access to.

Figure 41: The Google Voice number used to register with temi

Figure 42: Taking the MD5 hash of the phone number

Sure enough, this hash appeared in two places. First, it appeared in the primary SQLite 3 database file for the temi app running on the robot itself. More specifically, it appears in the "RecentCallModel" table under the "userid" column for the table's only entry. Based on the table's name and sole entry, this is almost certainly used to store temi's recent calls, as its admin was the only user it had called at this point.

Figure 43: The matching string in temi's RecentCallModel table, part of its SQLite 3 Database

The second match was in the log output we saved after running logcat on our temi earlier, the same log output seen in [Figure 30](#):

Figure 44: Matching strings in the logcat output recorded earlier

This log output appears to conclusively confirm our suspicions. With what we knew now, these log messages appeared to be JSON data being sent to/received from the MQTT broker. Moreover, the "topic" string in the first result exactly matched the MQTT topic format string found on line 82 of [Figure 38](#). The only difference is that in that case, the string between "users/" and "/status" was the robot ID; here, it was an MD5 hash of the user's phone number.

Since we now knew that

1. temi robots and phone app users received calls on the topic "client/<CLIENT_ID>/invite", where "<CLIENT_ID>" is the MQTT client ID of the callee,
2. the MQTT client ID for phone app users was simply an MD5 hash of the phone number used to register with the app, and
3. we could alter the existing phone app,

it stood to reason that we could modify the app to subscribe to another user's call invite topic in order to intercept calls intended for that user as long as we knew the user's phone number. In theory, this would allow an attacker to effectively impersonate another user and spy on them by intercepting their calls. The question then became: What code needs to be modified in order to get this to happen? Well, we're looking at a situation where temi initiates a call with its admin and an attacker attempts to intercept that call. In the case of calls initiated by the phone app, we discovered that call invitations are sent via `InvitationManagerImpl.sendInviteMsg()`, which publishes the call invite message on the topic "client/<ROBOT ID>/invite". We suspected a similar approach was being used when a call is initiated from a temi to a phone user and decided to investigate to confirm.

Luckily for us, the exact same `InvitationManagerImpl.sendInviteMsg()` method could be found in the temi robot's code, and it even seemed to function identically. Thus, it was probably safe to assume that the robot initiates calls with phone users in the same way: by publishing a call invitation to the topic "client/<CLIENT ID>/invite", `CLIENT_ID` being the MQTT client ID of the callee.

If the caller publishes their call invites to a certain MQTT topic, it follows that the callee must subscribe to that same topic to receive the invite. Now that we knew the format of the call invite topic, the next step was to track down the code used by the Android app to subscribe to this topic so we could alter it to subscribe to the temi admin's topic instead.

Performing a string search for this pattern in the decompiled phone app's code produced three unique results:

Figure 45: Searching for "client/./invite"

The second result we recognized as being part of the code used to generate an outgoing call invitation. Thus, we were only interested in the first and third results.

We began by looking at the first result, found on line 170 of `InvitationManagerImpl.java`:

Figure 46: `InvitationManagerImpl.sendInviteAbortMsg()`

This reference is part of the method `InvitationManagerImpl.sendInvitationAbortMsg()`. Since we were interested in the code that subscribes to the call invite topic and not the code that publishes messages to it, we moved on.

The third result was found on line 523 of `MqttManagerImpl.java`:

Figure 47: `MqttManagerImpl.buildInviteTopic()`

This didn't tell us anything about how the generated topic is used, so we took a step back and looked at what code invokes this method:

Figure 48: MqttManagerImpl.lambda\$initMqttClient\$13()

The call to buildInviteTopic() can be seen on line 408. There's a lot going on in this method, but at a high level it appears that it is responsible for setting the callback functions for the MqttManager, which are being defined inline. More specifically, the invite topic string generated by buildInviteTopic() is used in the connectComplete() callback function, where it is passed as the first parameter to MqttManagerImpl.subscribe().

As expected, MqttManager's subscribe() method is used to subscribe to a particular MQTT topic, with the first parameter being the topic string:

Figure 49: MqttMangerImpl.subscribe()

Thus, it appeared that we had found the code being used to subscribe to the call invite MQTT topic. Based on these findings, we decided the simplest approach would be to change the call to MqttManagerImpl.subscribe() on line 408 of [Figure 48](#) - instead of passing it the topic string returned by MqttManagerImpl.buildInviteTopic(), we would instead hard-code it to pass the temi admin's call invite MQTT topic string.

Using this approach, we were able to construct a modified temi phone app that would receive all calls intended for another user, as shown in the following video:

Here, the vulnerability was the lack of any authentication when publishing or subscribing to arbitrary topics, denoted by CVE-2020-16167 and having a CVSS score of 8.6. At a minimum, a check should have been made to ensure that clients cannot subscribe to another client's topics.

Problem: temi Won't Answer My Calls

Our next goal was to gain the ability initiate a call with a temi and have it automatically answer. As mentioned previously, this capability is typically reserved for the temi's admin and users explicitly authorized by the admin. Thus, if an attacker wishes to spy on a temi user, they would need to trick temi into thinking the call is originating from one of these authorized users.

With no other leads, we began searching through the robot's codebase for keywords related to user permissions, ownership, admin, etc. until we found a very promising enum class:

Figure 50: Class used for delineating the various roles held by temi's users

This enum is used in com.roboteam.teamy.users.User, the class used to store information about individual temi users, with User.role being an object of the Role enum class:

Figure 51: Class used to describe individual temi users/contacts

references to the Role class, but we'll only be looking at the pertinent ones here.

The first reference we'll be looking at appears on lines 351 and 357 of `ConferencePresenter.handleViewForCallTypes()`:

Figure 52: `ConferencePresenter.handleViewForCallTypes()`

On line 351, a string comparison is performed between `this.resourceLoader.getString(R.string.privacy_support_support_caller_name)` and `this.callerDisplayName`; if the two match, a new User is created with a role of CONTACT. So just what is this string that the 'if' statement is checking against? We decided to take a look at where this constant is defined:

Figure 53: Searching for "privacy_support_support_caller_name"

Taken together, this means that if the caller's display name is exactly "Support", a new User is created with CONTACT privileges. This check likely exists in the event that a member of temi's support staff calls a user. While this was certainly interesting, it is a niche scenario.

What happens if the caller's name is not "Support"? In that case, the else statement on line 352 is hit, which simply sets user to the result of `getUserByPeerId()`:

Figure 54: `ConferencePresenter.getUserByPeerId()`

This method tries to obtain the User associated with the current caller by performing a lookup in temi's `UsersRepository` using the caller's MQTT client ID. If the lookup succeeds, the found User is returned; if it fails, a new User is created with `Role.CONTACT` privileges.

As mentioned previously, [Figure 52](#) contains two references to the Role class. Let's now look at the second reference, found on line 357. Here, the role of the user is checked. If they are either an ADMIN or an OWNER, temi:

- waits 1.5 seconds (line 362)
- checks if the call hasn't already ended (line 365)
- if it hasn't, answers the incoming call (line 370)

Otherwise, the function returns after setting the app's view for an incoming call.

Solution: Become an OWNER

To recap what we've learned thus far:

- The role of ADMIN appears to be reserved for the user that originally registers temi via QR code scan.
 - If the user calling temi is not recognized, a new User object is created for them with CONTACT privileges.
 - If the user calling temi is recognized, their role is checked:
 - If they are a CONTACT, temi waits for the call to be answered via touchscreen.
 - If they are either an ADMIN or an OWNER, temi answers the call automatically.
- This is the behavior we want.**

Scanning the temi's QR code for the purposes of registration is only possible when temi is first powered on or after performing a factory reset. Thus, the attacker cannot realistically make themselves an ADMIN. And since CONTACT privileges are insufficient to get temi to pick up automatically, our best bet was to figure out how to obtain the role of OWNER.

Adding an OWNER: Phone App's Perspective

Although we knew that it was possible to promote a user to an OWNER using the phone app and we suspected that was achieved via MQTT, we still weren't sure of what goes on "under the hood" to make that happen.

After some searching, we found that `AddOwnersPresenter.addOwners()` is called whenever an admin selects one or more users to be granted OWNER privileges:

Figure 55: `AddOwnersPresenter.addOwners()`, trimmed

Here, `selectedIds` refers to the MQTT client IDs of the users selected to be promoted and `robotId` refers to the MQTT client ID of the temi robot this method is granting permissions for.

This request is sent by first fetching the unique private key used to identify this app's instance (line 101), and then creating a new `AddRemoveOwnersRequest` using the `selectedIds`, `robotId`, and this private key:

Figure 56: `AddRemoveOwnersRequest`

The constructor for `AddRemoveOwnersRequest` creates a timestamp for the request, then creates a new `AddOwnersRequestRequest`, which contains the body of the request, and finally uses the passed in `privateKey` in order to generate a signature for the `AddOwnersRequestRequest`. In other words, `AddRemoveOwnersRequest` is nothing more than a wrapper for the real request and is used to store its signature.

We decided to look at this `AddOwnersRequestRequest` object next. While the `ownerIds`, `robotId`, and `timestamp` members were mostly self-explanatory, `source` and `type` were less so. Looking back at line 8, we saw that `source` was being set to a hardcoded value of "ADMIN", which seemed to imply that this was the origin of the request. Looking back at line 6, we saw that `type` is simply the passed in `OwnersRequestType` enum (in our case, `OWNERS_ADD_TYPE`) converted to a string. This enum, defined near the bottom of the class, can take on two values: `OWNERS_ADD_TYPE` and `OWNERS_REMOVE_TYPE`. This implied that this same structure was recycled for requests meant to demote OWNERS back to CONTACTs.

Thus, we determined that `AddRemoveOwnersRequests` had the following structure:

Figure 57: Anatomy of an `AddRemoveOwnersRequest`

Now that we knew the structure of these requests, we next wanted to know how they were being sent and where. To this end, we decided to look at `OwnersApi.addOwners()`, which, according to [Figure 55](#), is actually sending the `AddRemoveOwnersRequest`.

Figure 58: `OwnersApi.addOwners()`

The body of this method didn't tell us much, but the imports for the `OwnersApi` class gave us a clue: this was using the [Retrofit 2 HTTP Client for Android](#), which is typically used to send HTTP requests to a REST API. According to [this Retrofit tutorial](#), the `@POST` annotation "indicates that we want to execute a POST request when this method is called" and "the argument value for the `@POST` annotation is the endpoint." The `@Body` annotation, on the other hand, indicates that we want the `AddRemoveOwnersRequest` to serve as the body of the POST request.

Okay, so this method is simply sending the request to add an OWNER via POST to some REST server at the endpoint "ownership/admin/add/owners". Our next question became: Where is this REST server located?

Part 5 of that same Retrofit tutorial told us that the `getClient()` method is typically used to obtain/create a Retrofit instance, and the only argument it takes is a string containing the URL for the REST server. Searching for "`getClient()`" in the phone app's code led us to `ApiClient.getClient()`:

Figure 59: `ApiClient.getClient()`

Working backwards from this method, we were able to track down the server's URL:

Figure 60: URLs for the MQTT broker and REST server

This URL confirmed that the recipient of this request was *not* the temi robot and it was *not* being sent via MQTT, contrary to our initial assumptions. This begged the question: if the phone app wasn't sending these requests to temi, how was temi being notified of any updates to the privileges of its users? We hypothesized that this REST server was simply a middleman whose job was to authenticate all requests to add/remove OWNERS

request's signature was valid.

We took some time trying to send these requests from a user that wasn't temi's admin, but they failed, lending some credence to our theory. This was looking like a dead end.

Adding an OWNER: temi's Perspective

Well, if we couldn't successfully spoof the request the phone app sends to the REST server, perhaps we could instead spoof the request the server sends to temi, bypassing the authentication mechanism altogether. We started looking for the code temi used to handle these requests.

Searching specifically for "Role.OWNER" in the temi's decompiled code led us to `OwnersController$getUsersRepository$2.apply()`:

Figure 61: `OwnersController$getUsersRepository$2.apply()`

Starting with `OwnersController$getUsersRepository$2`, we moved up the call chain in an attempt to discover how temi processes requests to add OWNERS. More concretely, this was accomplished through a liberal use of the "Find Usage" feature in JADX, which can be done by simply right-clicking virtually any symbol. Although convenient, "Find Usage" would often fail when the method in question was not invoked directly, such as when an implementation of an abstract class/interface served as the middleman. In such cases, we would perform a string search for instances where the method was invoked in the small code. To help separate invocations from declarations, we took advantage of the small syntax for method calls, which consisted of the name of the class the method belonged to, followed by a right arrow (`->`), followed by the method's signature.

As an example, "Find Usage" failed for the `accept()` method of the class `UsersAdminTopicListener$listenToUserAdminTopic$1`, so to find where it's invoked, we ran:

Figure 62: Searching for `UsersAdminTopicListener$listenToUserAdminTopic$1.accept()` in the small code

For especially indirect invocations, like dependency injection, more creative means had to be used, but a combination of "Find Usage" and string searches such as these got us 99% of the way there.

Using this approach, we found that the mechanism begins with the `UsersAdminTopicListener` class, which, as the name suggests, handles listening on temi's "users admin" topic. Moving down the call chain from here, we found out how messages received on this topic are processed by temi and ultimately used to alter the Role, or privilege level, of certain contacts. Based on our earlier analysis, it was likely that the REST server would forward the request sent by the phone app to this MQTT topic.

We found that the bulk of the work is performed by a method called `listenToUserAdminTopic()`:

Figure 63: `UsersAdminTopicListener.listenToUserAdminTopic()`

This function does several things. First, on line 50, it creates a [Flowable](#) object by calling `UsersAdminTopicListener.getOwnerRequestFlowable()`. Next, on lines 51-56, it subscribes to this Flowable and for each emitted `OwnersRequest`, it calls `OwnersController.handle$app_usaDemoRelease()` upon success or simply logs upon failure.

We decided to first look at the code for `getOwnerRequestFlowable()`:

Figure 64: `UsersAdminTopicListener.getOwnerRequestFlowable()`

- It begins by first converting an Observable obtained via `mqttPipeline.observe()` into a Flowable.
- Next, it throws out all emitted `MqttMessages` whose topic doesn't match the regex `"users/.+/admin"` via [RxJava's filter\(\) method](#).
- [RxJava's map\(\) method](#) is then used to convert the body of the `MqttMessage` from JSON to an object of the `OwnersMessage`
- Finally, `map()` is used a second time to extract and return the `OwnersRequest` from each `OwnersMessage`.

At this point, we decided it would be useful to understand the structure of `OwnersRequests` and `OwnersMessages`, since these seem to be key to temi's privilege management mechanisms:

CONTACT to an OWNER (OWNERS_ADD) or demote an OWNER back to a CONTACT (OWNERS_REMOVE).

Comparing this to [Figure 57](#), an OwnersMessage appears to be an AddRemoveOwnersRequest without the signature. Similarly, an OwnersRequest appears to be a stripped-down AddOwnersRequestRequest, with the robotid, source, and timestamp omitted. This meshes well with our earlier hypothesis that the REST server's job is to authenticate and forward AddRemoveOwnersRequests to the temi robot. The signature, timestamp, and source would be omitted since they've already been verified by the server; the robotid, while needed by the server to know where to forward the request, becomes redundant once the request reaches temi.

Our next step was to figure out what `handle$app_usaDemoRelease()` does. Since it's quite the rabbit hole, however, we will summarize its effects in lieu of venturing down it:

1. It queries the Users table in temi's local database for all users with IDs matching any of the ones in the OwnersRequest's ownerids
2. It replaces the Role for each of these users with one corresponding to the OwnersRequest's type: OWNERS_ADD → OWNER, OWNERS_REMOVE → CONTACT.
3. It updates temi's Users table with the updated user information.

This was promising, since it meant that we could potentially trick temi into promoting an arbitrary user/contact to an OWNER simply by crafting a custom OwnersRequest and publishing it on temi's `users/<ROBOT_ID>/admin` topic, thereby bypassing the authentication server entirely.

Unfortunately, part 1 above reveals a potential obstacle for this plan: since temi will only process OwnersRequests for users already present in its local Users table, we must first add ourselves to this table for this strategy to succeed. Recalling our earlier analysis of how temi handles unrecognized callers, one way of accomplishing this was to simply cold call temi, which would cause it to automatically add the caller to its contacts list. This was far from ideal, however, since cold calling temi without already having the role of OWNER or ADMIN would cause it to ring and display the caller's username on its screen, potentially alerting temi's users that something weird is going on.

Detour: Sneaking Onto temi's Contact List

Before continuing on, we decided to take a brief detour to find a better way for an attacker to add themselves to temi's contact list.

From our prior investigation into how temi implements its various privilege levels through the use of Roles, we discovered that temi uses the User class to define its various users. Thus, it follows that any code used to add a new user to temi's local contact list would first create a new User object, so that's exactly what we searched for.

Figure 66: Searching temi's code for `"new User(",` trimmed

[Figure 66](#) shows the result that we followed up on, the others have been omitted. The name of the containing class, `SyncContactsController`, sounded promising on its own since syncing contacts from temi's ADMIN would likely involve adding new contacts without needing to start a call, which was exactly what we were trying to do.

Using largely the same strategy we employed for tracing the code flow for adding OWNERS (JADX's "Find Usage" feature + grepping the small code), we were able to trace the code flow all the way back to the app's entry point. With a more holistic understanding of the mechanism used to sync contacts, we realized that the process is ultimately kicked off by `SyncContactsTopicListener.subscribeMqttPipeline()`:

Figure 67: `SyncContactsTopicListener.subscribeMqttPipeline()`

The first thing this method does is take this `mqttPipeline` and turns it first into an [Observable](#) (using `MqttPipeline.observe()`) and then into a [Flowable](#) (using `RxJavaInterop.toV2Flowable()`), as seen on lines 29 and 30.

Essentially, this `mqttPipeline` acts as a relay. Incoming MQTT messages are pushed to the relay using its `push()` method, which it then relays to all its observers.

The output of this relay is then filtered on lines 31-38 to only return MQTT messages received on topics matching the regex `"syncontacts/.+"`. Based on the other MQTT topic strings we've seen up to this point –

- `"users/<CLIENT_ID>/status"`
- `"users/<CLIENT_ID>/admin"`
- `"client/<CLIENT_ID>/invite"`

– we were fairly certain temi's client ID goes after the forward slash. Thus, temi appeared to be listening on the MQTT topic `"syncontacts/<CLIENT_ID>"` for messages regarding contact synchronization.

On lines 39-43, the now-filtered MQTT messages emitted by the relay are passed to a call to RxJava's [map\(\)](#) function, which converts each incoming MQTT message from JSON to an object of the `SyncContactsMessage` class. We quickly determined that `SyncContactsMessages` had the following structure:

Figure 68: Anatomy of a `SyncContactsMessage`

Put briefly, each `SyncContactsMessage` consisted of a `senderClientid`, a string holding the MQTT client ID of the request's sender, and `contacts`, a list of `ContactEntry` objects. Each `ContactEntry` object in the list corresponded to a contact to be synced to temi's contact list, containing both their `clientid` and their name.

Figure 69: SyncContactsController.save()

This method is doing a lot, but we'll focus on only the most pertinent side-effects:

- On lines 53-62, **all messages where the senderClientId does not match the temi admin's client ID are discarded**. This will become important later.
- On lines 63-75, the list of contacts is extracted from the SyncContactsMessage and is used to build a list of User objects – one per contact. The Users produced by this method are initialized in the following manner:

Member	Assigned Value
User.id	ContactEntry.clientId
User.name	ContactEntry.name
User.picUrl	""
User.role	Role.CONTACT
User.userId	SyncContactsMessage.senderClientId

- On lines 81-85, our newly-minted list of User objects is passed to insertOrUpdateContact(). This method writes the list of Users to the Users table in temi's SQLite 3 database, completely overwriting temi's old contacts list in the process.

So now that we knew how an ADMIN's contacts are synced to their temi, how could we leverage that knowledge to add ourselves to temi's contact list in a discrete fashion? Well, if we could publish a message to temi's synccontacts MQTT topic, we could add ourselves as a contact. Although temi does perform a check to make sure that the sender of the message is its ADMIN, it makes the mistake of trusting that the contents of the message accurately reflect the actual sender. In theory, there's nothing stopping us from publishing a SyncContactsMessage from one client ID and setting the senderClientId field in the message to a completely different ID – the ADMIN's client ID, for example.

Based on our understanding of how the temi robot parses MQTT requests to sync contacts, we crafted a JSON message that should decode into a valid SyncContactsMessage object and would add our "attack" phone to temi's contacts:

Figure 70: Our custom SyncContactsMessage in JSON format

This message was crafted using the following rationale:

1. Objects in JSON are indicated via curly braces (`{}`). Since the entire message contents are being converted into a single SyncContactsMessage object, it follows that the contents should be contained within a pair of curly braces, representing the SyncContactsMessage object itself.
2. A SyncContactsMessage contains a senderClientId, a string indicating the client ID of the "sender" of the message. Thus, we added an element to our object with the key "senderClientId" and the string "060f62296e1ab8d0272b623f2f08f915" – the client ID of the temi's admin – as its value.
3. A SyncContactsMessage also contains contacts, a list of ContactEntry. Thus, we added an element with the key "contacts" and a new list as its value. In JSON, lists are indicated via square brackets (`[]`).
4. In our case, the contacts list contains only a single ContactEntry – one corresponding to the "attack" phone, so we added a single pair of curly braces to the list to indicate the contents of this single ContactEntry.
5. Each ContactEntry contains a clientId. Thus, we added an element to the object with the key "clientId" and the string "fe5d7af42433f0b6fb6875b6d640931b" – the client ID of the "attack" phone – as its value.
6. Each ContactEntry also contains a name. Thus, we added a second element to the object with the key "name" and the string "Test" as its value. This simply represents the display name for the contact, so we could set it to basically whatever we liked.
7. As for spacing and newlines, we referred to the [Gson User Guide](#), since that's the library temi was using to perform JSON serialization/deserialization. The guide states:

"The default JSON output that is provided by Gson is a compact JSON format. This means that there will not be any whitespace in the output JSON structure. Therefore, there will be no whitespace between field names and its value, object fields, and objects within arrays in the JSON output."

As a result, whitespace and newlines have been omitted.

Now that we understood where to publish the request and what the request should look like, the next step was figuring out how to alter the temi phone app to send this request. Since we were aiming for a simple proof-of-concept, we prioritized ease and speed of implementation over robustness or elegance when considering which code to change. To this end, it made sense to leverage the code already present in the app that was dedicated to publishing MQTT messages since it would minimize the amount of code we needed to change and thus reduce the risk of introducing unintended bugs into our altered app – an ever-present risk when directly modifying an app's bytecode. While the phone app publishes many MQTT messages to various topics during its runtime, we decided to try using the phone app's video calling functionality. This had the obvious advantage of giving us clear control over when and how often the MQTT message is published, since calls are always initiated by hitting the "Connect" or "Call" buttons. Ultimately, we decided to leverage InvitationManagerImpl.sendInviteMsg() for this

video calls. This meant that it would only get executed unless when we manually initiated a call from the app.

Running our altered app resulted in our attack phone being added to temi's contact list, as shown in the following video:

Gaining OWNER Privileges

All that was left was for us to craft and publish a custom OwnersMessage in order to gain OWNER privileges on our temi. As before, we began by crafting the JSON for the message itself.

Figure 71: Our custom OwnersMessage in JSON format

This message was crafted using the following rationale:

1. Since the entire message contents are being converted into a single OwnersMessage object, it follows that the contents should be contained within a pair of curly braces, representing the OwnersMessage object itself.
2. An OwnersMessage contains a request, an object of the OwnersRequest class, and nothing else. Thus, we added an element with the key "request" and a new object as its value. The contents of this inner object will become our OwnersRequest.
3. An OwnersRequest contains a type, an enum specifying the type of request. In our case we want to add an OWNER, so we added an element with the key "type" and the string "OWNERS_ADD" as its value. As for why we're expressing this enum value as a string, it's because [this article](#) shows that Gson's default behavior is to express enum values as strings corresponding to the name of the enum value.
4. An OwnersRequest also contains ownerids, a list of strings enumerating the temi contacts the request should apply to. In our case, the list contains only a single ID - the ID of the "attack" phone, which is just the MD5 hash of its phone number.
5. As before, spaces and newlines have been omitted, per the Gson User Guide.

Fortunately, we were able to leverage the code modifications we used to publish our SyncContactsMessage, since the only things changing were the MQTT topic we're publishing to and the contents of the message.

Our full test consisted of running our first modified app in order to add ourselves to temi's contact list, followed by our second modified app in order to perform privilege escalation, as shown in the following video:

The call appeared to be fully functional, and we were able to drive temi around, navigate to its saved locations, and receive its audio/video feeds. All an attacker needed to make this possible was the phone number of one of temi's contacts.

This authentication bypass was the final and most impactful vulnerability we discovered in the temi robot, denoted by CVE-2020-16169 and having a CVSS score of 9.4. To better understand how this an auth bypass, let's compare temi's privilege management during normal operation:

Figure 72: temi's privilege management during normal operation

to how it looks using our modified app:

Figure 73: temi's privilege management with auth bypass

Refining the Exploits

Once our exploits had the capabilities we initially set out to obtain, we got to work refining them. We created a single modified app that combined all of the MQTT attack vectors we previously outlined: it could intercept calls and send both MQTT messages necessary to obtain OWNER privileges, all with a single button press. Furthermore, we added some additional logic to automatically extract the client IDs required by our custom MQTT messages from the app's contact list, meaning our app would work on any temi robot. To see all these features in action, please refer to the video below:

Impact

At this point it would be prudent to take a step back, review the combined capabilities of the exploits we've outlined, and consider what impact these might have in a real-world setting.

At the time of discovery, the vulnerabilities in the temi robot meant that an attacker could join any ongoing temi call simply by using a custom Agora app initialized with temi's hardcoded App ID and iterating over all 900,000 possible channel names – certainly feasible with modern computing power. This becomes more plausible if one considers that our testing revealed that joining a temi call this way does not notify either of the existing call participants that another user is present, since the apps were only designed with 1-on-1 calling in mind. The fact that the attacker needs no information on the victims makes this attack vector worrisome, but it also means that the attacker has no control over who he or she spies on using this method. Realistically, a malicious actor might use this exploit as a means of reconnaissance – by collecting data on arbitrary temi users, they could follow up on the more "promising" ones later with one of the targeted attack vectors. Given temi's limited adoption by consumers but ramping adoption in industries such as healthcare, the odds are in the attacker's favor that they stumble upon sensitive information.

Furthermore, if an attacker has a specific victim in mind, they could leverage the lack of per-topic authentication in temi's MQTT implementation in order to intercept calls between a user and their temi. The plausibility of this particular attack vector is difficult to evaluate. On the one hand, the only information the attacker needs is the victim's phone number, and telemarketers consistently remind us that this is a very low bar. On the other hand, this exploit's behavior during our testing proved somewhat inconsistent. At times, the user running the exploit could listen in on the call as a third party with the call participants being none the wiser. At other times, the user running the exploit would disconnect the original caller and take their place. It is easy to imagine scenarios where both of these outcomes are desirable to a malicious actor, and neither bodes well for the privacy of the user being targeted. It is also important to note that although we focused on using this vulnerability to intercept calls, it can also be used to intercept all sorts of notifications intended for another user: contacts being added, when temi goes online/offline, its current location, etc. While less flashy than intercepting a call, this approach is more discreet and might allow a dedicated attacker to learn a user's routine, only showing their hand to listen in when it would be most impactful.

Of the possible attack vectors, we believe that the ability to call and control a temi remotely, leveraging the authentication bypass present in temi's privilege management mechanism, is the most impactful. The bar for this attack vector is arguably even lower than the previous, as the attacker would only need the phone number of *any* of temi's contacts – it need not be its admin. In our testing, none of the steps involved in leveraging this exploit notify temi's admin in any way that something is amiss; they are not notified that the attacker has added themselves to the robot's contact list nor that they have gained raised privileges. Since this method does not cause temi to ring, an observer would have to see the attacker move temi or have a good look at its touchscreen during the attack to know something nefarious was going on. This level of control and subtlety becomes especially problematic when we consider some of temi's applications in industry. In April of this year, Robotemi Global Ltd. stepped up production to 1,000 units per month in response to [Israel's Ministries of Defense and Health choosing temi to assist with patients in their COVID-19 wards](#). In South Korea, [temi sees use in both public places and nursing homes](#), helping to facilitate social distancing. Besides the obvious impact of compromising patients' sensitive medical information during remote doctor's visits, the ability to have eyes and ears into a hospital is worrying in its own right. It isn't difficult to imagine what a malicious agent might do with an overheard network password, access code to a sensitive area, or the location and condition of a person of interest.

Conclusion

The findings outlined in this paper highlight the importance of secure coding practices and security auditing for cutting edge technologies, particularly in the IoT space. When an IoT camera evolves into one that can also drive around your home or business and even assist medical patients, the need to properly secure who can access it only becomes more important. While history has demonstrated that any sufficiently complex software is bound to have vulnerabilities, there are many steps we can take to substantially raise the bar for bad actors. In our view, these responsibilities are shared between vendors, consumers, and the security industry as a whole.

First and foremost, vendors can ensure they are employing proper security hygiene when designing products. Often, best practices consist not of novel, out-of-the-box thinking, but rather adopting time-tested guidelines like the *principle of least privilege*. In the case of temi, application of this principle likely would have addressed CVE-2020-16167, which allows clients to subscribe/publish to topics they have no business accessing. Considerations for security should also extend beyond the development phase, with third-party security auditing being a powerful tool that allows vendors to discover and patch vulnerabilities before the bad guys get ahold of them. Taking vulnerability disclosure seriously and cooperating with the bodies that report them can often net similar benefits, and this is an area Robotemi excelled in.

Individuals, while typically having fewer resources, can also take certain precautions. Placing IoT devices on a VLAN, a virtually segregated subnetwork, reduces the risk that a vulnerability in one device will compromise your entire network. Staying up to date on important vulnerabilities can also help inform consumers' purchasing decisions, although the presence of vulnerabilities is often less illuminating than if/how a company chooses to address them.

Finally, we in the security industry also have an obligation towards moving the needle in the realm of security, one that we hope research such as this embodies. One of our goals here at McAfee ATR is to identify and illuminate a broad spectrum of threats in today's complex and constantly evolving landscape. While we take seriously our obligation to inform vendors of our findings in a timely and responsible fashion, it is only through cooperation that the best results are possible. Our partnership with Robotemi on addressing these vulnerabilities was a perfect example of this. They responded quickly to our private disclosure report, outlined to us their plans for mitigations and an associated timeline, and maintained a dialogue with us throughout the whole process. We even received feedback that they have further emphasized security in their products by approaching all development discussions with a security-first mindset as a result of this disclosure. The ultimate result is a product that is more secure for all who use it.



Stay Updated

Follow us to stay updated on all things McAfee and on top of the latest consumer and mobile security threats.

Mark Bereza

Mark Bereza is a Security Researcher and recent addition to McAfee's Advanced Threat Research team. An alumnus of Oregon State University's CS systems program, Mark's work has focused on vulnerability...

More from McAfee Labs



[Fake Security App Found Abuses Japanese Payment System](#)

Authored by SangRyol Ryu and Yukihiro Okutomi McAfee's Mobile Research team recently analyzed new malware targeting mobile...

NOV 30, 2022 | 5 MIN READ



[Threat Actors Taking Advantage of FTX Bankruptcy](#)

Authored by Oliver Devane It hasn't taken malicious actors long to take advantage of the recent bankruptcy...

NOV 15, 2022 | 2 MIN READ



[Microsoft's Edge over Popups \(and Google Chrome\)](#)

Following up on our previous blog, How to Stop the Popups, McAfee Labs saw a sharp decrease...

NOV 15, 2022 | 2 MIN READ



[Don't Get Caught Offsides with These World Cup Scams](#)

Authored by: Christy Crimmins and Oliver Devane Football (or Soccer as we call it in the U.S.)...

NOV 09, 2022 | 6 MIN READ



[New Malicious Clicker found in apps installed by 20M+ users](#)

Authored by SangRyol Ryu Cybercriminals are always after illegal advertising revenue. As we have previously reported, we...

OCT 19, 2022 | 6 MIN READ



[Malicious Cookie Scripts Chrome Extensions Million Users](#)

Authored by Oliver Devane Chole September 9, 2022 the original publication of...

AUG 29, 2022 | 8 MIN READ



Products

[McAfee® Total Protection](#)
[McAfee® Gamer Security](#)
[McAfee® Identity Monitoring Service](#)
[McAfee® Security Scan Plus](#)
[McAfee® WebAdvisor](#)
[McAfee® Techmaster Concierge](#)
[McAfee® Virus Removal Service](#)

Resources

[Antivirus](#)
[Free Downloads](#)
[Parental Controls](#)
[Malware](#)
[Firewall](#)
[Blogs](#)
[Activate Retail Card](#)
[Threat Center](#)
[McAfee Enterprise](#)

Support

[Consumer Support](#)
[FAQs](#)
[Renewals](#)
[Support Community](#)

About

[About McAfee](#)
[Careers](#)
[Contact Us](#)
[Newsroom](#)

