Mystique in the House: The Droid Vulnerability Chain that Owns All Your Applications

Dawn Security Lab

Abstract

The Android Application Sandbox is the cornerstone of the Android Security Model, which protects and isolates each application's process and data from the others. Attackers usually need kernel vulnerabilities to escape the sandbox, which by themselves proved to be quite rare and difficult due to emerging mitigation and attack surfaces tightened.

However, we found a vulnerability in the Android 11 stable that breaks the dam purely from userspace. Combined with other 0days we discovered in major Android vendors forming a chain, a malicious zero permission attacker app can totally bypass the Android Application Sandbox, owning any other applications such as Facebook and WhatsApp, reading application data, injecting code or even trojanize the application (including unprivileged and privileged ones) without user awareness. We named the chain "Mystique" after the famous Marvel Comics character due to the similar ability it possesses.

In this talk we will give a detailed walk through on the whole vulnerability chain and bugs included. On the attack side, we will discuss the bugs in detail and share our exploitation method and framework that enables privilege escalation, transparently process injection/hooking/debugging and data extraction for various target applications based on *Mystique*, which has never been talked about before. On the defense side, we will release a detection SDK/tool for app developers and end users since this new type of attack differs from previous ones, which largely evade traditional analysis.

I. Bugs

This section describes the whole chain of *Mystique*, including a fundamental bug in AOSP which has been assigned CVE-2021-0691, and bugs in other major vendors such as CVE-2021-25450, CVE-2021-25485 and CVE-2021-23243.

A. The AOSP bug

The recent change in AOSP's SELinux policy caught our attention: this change accidentally introduces additional ability for system_app context[1]:

```
index e5d7d18..1432017 100644
--- a/private/system_app.te
+++ b/private/system_app.te
@@ -69,6 +69,9 @@
 # Settings need to access app name and icon from asec
 allow system_app asec_apk_file:file r_file_perms;

+# Allow system_app (adb data loader) to write data to /data/incremental
+allow system_app apk_data_file:file write;
+
 # Allow system apps (like Settings) to interact with statsd
 binder_call(system_app, statsd)
```

The label *apk_data_file* corresponds to files in */data/app*, which are most app's code files.

```
a70q:/data/app/com.unionpay.tsmservice-0bfCG-Zd90PuEwegtR8UzQ== # ls -lR
.:
total 6184
-rw-r--r-- 1 system system  6308538 2021-01-01 01:22 base.apk
drwxr-xr-x 3 system system     4096 2021-01-01 01:22 lib
drwxrwx--x 3 system install    4096 2021-01-01 01:22 oat

./lib:
total 8
drwxr-xr-x 2 system system 4096 2021-01-01 01:22 arm64

./lib/arm64:
total 2092
-rwxr-xr-x 1 system system  185816 1979-11-30 00:00 libentryexpro.so
-rwxr-xr-x 1 system system  723744 1979-11-30 00:00 libuptsmaddon.so
```

```
-rwxr-xr-x 1 system system 1216480 1979-11-30 00:00 libuptsmservice.so

./oat:
total 8
drwxrwx--x 2 system install 4096 2021-01-01 01:22 arm64

./oat/arm64:
total 2588
-rw-r--r-- 1 system all_a192   28672 2021-01-01 01:22 base.art
-rw-r--r-- 1 system all_a192   87712 2021-01-01 01:22 base.odex
-rw-r--r-- 1 system all_a192 2517132 2021-01-01 01:22 base.vdex
```

This policy allows any one with system-app uid file overwrite ability to overwrite any other app's code file, including *so* and *base.apk*, which will lead to code injection, execution or even applications being backdoored. It greatly enlarges the power of a possible system-app bug to nearly owns all the userspace. Before this bug is introduced, even a system-app shell cannot do meaningful things such as accessing other applications' code and data files. However, this bug grants them the privilege.

Looking back to the history of relevant commits, this change might be introduced for the feature *incremental install*, which is for large applications and games with size more that 1GB. In the Android R (11) Preview 1, a helper system application *AdbDataLoader* is added to support *incremental install*. Thus, the corresponding policy for system_app to modify apk_data is added. However, while the very user is removed in Preview 4 and never made it to the stable, this permission change is retained. Thus, this single line of change looses the whole Android sandbox.

This bug has been assigned CVE-2021-0691 by Google and fixed in September 2021's Android Security Bulletin.

B. Static Analysis Technique

To make use of the first vulnerability, we need to find bugs in system apps that allow us to write to arbitrary path with controlled content. This can be turned into a classical source-sink style data-flow taint analysis problem for the Java language, solved under the IFDS (Inter-procedure, Finite, Distributive) framework implemented on Soot.

For any taint analysis problem, we must first define sources and sinks. The sources are defined as follows:

1) Intents received from exported Activity, Service, Broadcast Receiver or Content Provider
2) Data received from controllable Input Streams such as Socket
3) Files from controllable locations such as Attacker package data and world-readable/writable locations

The sink definitions are as follows:

1) File OutputStreams
2) Dynamic code loading functions
3) Command Execution functions

The analysis algorithm is largely . To improve the precision of the data flow analysis framework,

We mapped these sources and sinks into our static code analyzer 'Star Watcher', which are built on top of FlowDroid, to major vendors such as Samsung and Oppo, and found multiple problems that can be used in this chain. For each device, we will elaborate on a typical bug used in following sections. Our static code analyzer automatically finds the possible code paths that lead to file overwrite.

Our tool is Context- and flow-, field-, object-sensitive and works on an interprocedure-cfg. To make the analysis as precise as possible, which stands out among other tools and find bugs that are previously unseen, there are several crucial design and implementation points that need to be considered:

1) Building a Complete Call Graph: A complete callgraph is fundamental to all the following analysis such as Data Flow and Taint Analysis. The current state-of-the-art SPARK pointer analysis of Java, occasionally misses allocation nodes, thus leading to incomplete callgraph and missing results. Consider this real-world examples:

Example 1:

```
public class Sample {
    public void target() {
        for(Helper helper: helpers)
            helper.handle();
    }
    Set<Helper> helpers;
    public Sample () {
        this.helpers = new HashSet<>();
        this.helpers.add(new HelperImplA());
    }
}
```

```
interface Helper {
    public void handle();
}
class HelperImplA implements Helper {
    @Override
    public void handle() {
        System.out.println("wtf");
    }
}
class HelperImplB implements Helper {
    @Override
    public void handle() {
    }
```

Spark pointer analysis cannot handle this situation in which elements are put into container and then retrieved out, it cannot decide the type of *helper* at the call site of *handler.handle()*. The call edges of *handler.handle* are thus missing from the call graph, leading to false negative if there are possible data flows in the *handle* function.

Example 2:

```
class AParcelable implements Parcelable {
    void func() {
    }
}
class MainActivity extends Activity {
    public void onResume() {
        AParcelable p = getIntent().getParcelableExtra("p");
        p.func();
    }
}
```

For Example 2, the allocation of the *AParcelable* class is done in the library code when unmarshaling parcelable, which is beyond the analysis scope. Spark in this example also fails to properly detect the allocation node for this variable, thus lead to missing edge for the *fun* call of the target class in the call graph.

To solve this problem, we incorporate CHA (Class Hierachy Analysis) with the original SPARK pointer analysis. For edges missing at invoke expression callsites, we conduct an additional CHA analysis and add the possible edges into the call graph. This approach is effective and lightweight against real-world examples and found bugs that would previously be unseen.

Another important part that may affect the completeness of the call graph is component API entrypoint changes. During the analysis, we build a dummy main method that calls all components' lifecycle methods to simulate the behavior of Android Framework. The component entrypoint methods largely remains unchanged however sometimes they do change. For example, a new variant of the *call* function of content providers was added in API 29.

2) Optimizing performance by dropping irrelevant instance data flows: The IFDS algorithm is by nature a fix-point algorithm, which need to keep data fact associated with each statement to decide whether we should stop. The IFDS algorithm works on an inter-procedure CFG and categorize the flow functions by four types:

- NormalFlowFuntions
- CallFlowFunctions
- ReturnFlowFunctions
- CallToReturnFlowFunctions

For instance taint facts, the CallFlow function is responsible for whether we should pass the fact into callee function. Normally, if the callee function is an instance invoke (virtual function), the IFDS will pass the instance taint fact into callee and record relevant edges created/cached with the statement, since the taint fact is implicitly associate with the *this* instance. However, if the instance field is never read or written to, we actually do not need to pass in the function anymore. This requires us to do a quick precheck and cache the instance analysis on the icfg.

We implement it as an optimization extension upon the original algorithm and the result is quite promising. Some analyses that previously throw OOM error now can success finish, and the overall speed improves by 20 percent.

C. The Vendor Bugs

1) Privilege Escalation in Camera and FilterProvider: When a package with *com.sec.android.camera.permission.USE_EFFECT_FILTER* is installed, the *FilterProvider* is automatically notified through its registered broadcast receiver and recognize it as a CAMERA EFFECT FILTER. The files in target package is extracted to */data/DownFilters/Lib64/*, in the following function:

```
package com.samsung.android.provider.filterprovider;
class FilterInstaller {
  //...
   private String copyFileFromAssets(AssetManager arg10, String arg11, String arg12,
    ↪  boolean arg13) {
     //...
         private void installLibFilter(Context arg11, String arg12) {
        Resources v11_1;
        Log.d("FilterInstaller", "install filter for Lib");
        try {
            v11_1 = arg11.getPackageManager().getResourcesForApplication(arg12);
        }
        catch(PackageManager.NameNotFoundException v11) {
            Log.e("FilterInstaller", "installLibFilter, resource error, package name :
             ↪  " + arg12 + ", exception : " + v11.toString());
            return;
        }

        AssetManager v12 = v11_1.getAssets();
        String[] v2 = new String[0];
        try {
            v2 = v12.list("so");
        }
        catch(IOException v3) {
            Log.e("FilterInstaller", "installLibFilter, asset lib list exception : " +
             ↪  v3.toString());
        }

        if(v2.length <= 0) {
            Log.e("FilterInstaller", "asset lib list length is small than 0");
            return;
        }

        int v3_1 = v2.length;
        int v4;
        for(v4 = 0; v4 < v3_1; ++v4) {
            String v5 = v2[v4];
            Log.d("FilterInstaller", "filterFile : " + v5);
            if(this.storeLibFilters(v12, v11_1, v5, v5.split("\\.")[0]) == -1L &&
             ↪  (v5.endsWith(".so"))) {
                Log.e("FilterInstaller", "storeLibFilters fail");
            }
        }


         ↪  this.mServiceContext.getContentResolver().notifyChange(Constants.URI_NOTIFY_ADD,
         ↪  null);
    }
     //...
private long storeLibFilters(AssetManager arg18, Resources arg19, String arg20, String
 ↪  arg21) {
  //...
          if((arg20.endsWith(".so")) && !this.checkSoSignature(v6)) {
            Log.e("FilterInstaller", "Signature check failed.");
            return -1L;
        }

        if(!arg20.endsWith(".sig")) {
```

```
            ContentResolver v10 = this.mServiceContext.getContentResolver();

            ContentValues v11 = new ContentValues();

            try {

                int v8_1 = arg19.getIdentifier(arg21 + "_title", "string",
                ↪  this.mPackageName);

                int v9_1 = arg19.getIdentifier(arg21 + "_vendor", "string",
                ↪  this.mPackageName);

                int v4 = arg19.getIdentifier(arg21 + "_version", "string",
                ↪  this.mPackageName);

                Log.e("FilterInstaller", "title = " + arg19.getString(v8_1) + ",
                ↪  vendor = " +
arg19.getString(v9_1) + ", version = " + arg19.getString(v4));

                v11.put("name", arg19.getString(v8_1));

                v11.put("filename", v6);

                v11.put("version", arg19.getString(v4));

                v11.put("vendor", arg19.getString(v9_1));

                v11.put("package_name", this.mPackageName);

                v11.put("title_id", Integer.valueOf(v8_1));

                v11.put("preload_filter", Integer.valueOf(0));

                v11.put("filter_type", "SINGLE");

                v11.put("category", Integer.valueOf(2));

                v11.put("vendor_package_name", this.mPackageName);

                if(v0_2 != null) {

                    v11.put("texture", v13);

                }

            }


              //...
```

Tex and *so* file will be extracted. However, for *so* files, the FilterProvider has some checks on the package. A *.sig* signature file must be accompanied and verified by prebuilt RSA public key, other wise the content provider will refuse to insert it into *content://com.samsung.android.provider.filterprovider/filters* table . The intended behavior is the *so* file is signed by Samsung private key and verified by public key so that the package can be distributed on Camera App Store (Filter Package Selection in Camera), like the following picture:

However, there's a major bug here: the verification process only checks for *so*, not for *SO*! If we change the packaged library to *so*, it will evade the signature check and successfully make it way into the content provider.

Later time when the Camera application is opened, and effect icon is clicked (the magic wand button) shown in below screen-

shot, all filters will be preloaded in to the Camera process space, by the following code in *libcamera_effect_processor_jni.so*

```
__int64 __fastcall sub_13A3A4(__int64 a1, char *a2, char *s1)
{
   //...
  v28 = 0LL;
  ptr[0] = 0LL;
  if ( strcmp(s1, "com.samsung.android.provider.filterprovider") && *(_DWORD *)(a1 +
  ↪  36) != 2 )
  {
    asprintf(ptr, "%s%s", "/data/DownFilters/Lib64/", a2);
    v7 = "/data/DownFilters/Tex/";
LABEL_11:
    ptr[1] = v7;
    goto LABEL_12;
  }
  v5 = *(_QWORD *)(a1 + 16);
  if ( v5 )
  {
    v6 = sub_57490(v5, a2);
    if ( v6 )
    {
      if ( v6 != 400 )
      {
        if ( (sub_57748(*(_QWORD *)(a1 + 16), (int)ptr, a2) & 1) != 0 )
          sub_575C4(*(_QWORD *)(a1 + 16), (int)&v28, a2);
        else
          asprintf(ptr, "%s%s", "/system/lib64/", a2);
        v7 = "/system/cameradata/preloadfilters/Tex/";
        goto LABEL_11;
      }
    }
  }
LABEL_12:
  __android_log_print(3, "SECIMAGING", "Try to open external effect (%s)", ptr[0]);
  *(_QWORD *)(a1 + 56) = dlopen(ptr[0], 2);
  free(ptr[0]);
  v8 = *(void **)(a1 + 56);
  if ( !v8 )
  {
    v19 = dlerror();
    __android_log_print(6, "SECIMAGING", "Filter %s dlopen failed", v19);
    return 0LL;
  }
  v9 = (__int64 (*)(void))dlsym(v8, "Create");
```

Given the Camera process which posesses *WRITE_EFFECT_FILTER* permission, what can we do next? The FilterProvider *MyFilterService* which requires this permission, accepts intent and write them to file:

```
package com.samsung.android.provider.filterprovider;

    void install(Bundle arg15) {

        Log.d("MyFilterInstaller", "install");

        if(arg15 == null) {

            Log.e("MyFilterInstaller", "install : bundle is null, return.");

            return;
```

```
        }


        this.createMyFilterDirectory();

        String v2 = arg15.getString("name");

        String v4 = arg15.getString("filename");

        byte[] v6 = arg15.getByteArray("filter_thumbnail");

        String v7 = v2 + ".bmp";

        String v8 =
↪  this.copySelFileFromIntent(MyFilterInstaller.FILTER_STORAGE_MY_FILTER + "/"
+ v4, arg15);

        if(v8 == null) {

            Log.e("MyFilterInstaller", "install : sel file copy failed = " + v4 + ",
↪  return.");

            return;

        }



//.. /data/xxx/ + "../../../../../"
 private String copySelFileFromIntent(String arg9, Bundle arg10) {
        byte[] v9_1;
        FileOutputStream v10;
        File v5;
        int v4 = 0;
        try {
            v5 = new File(arg9);
            goto label_13;
        }
        catch(Exception unused_ex) {
            v10 = null;
            v5 = null;
            goto label_43;
            try {
            label_13:
                if(v5.exists()) {
                    v5.delete();
                    Log.w("MyFilterInstaller", "The named file already exists : " +
                     ↪  arg9 + ". So remove file.");
                }

                v5.createNewFile();
                v9_1 = arg10.getByteArray("filter_data");
                v10 = new FileOutputStream(v5);
                goto label_33;
            }
            catch(Exception unused_ex) {
```

```
                v10 = null;
                goto label_43;
                try {
                label_33:
                    v10.write(v9_1);
                    if(!v5.setExecutable(true, false)) {
                        Log.e("MyFilterInstaller", "filter.setExecutable when copy
                         ↪  from asset is not complete properly");
                    }

                    if(!v5.setReadable(true, false)) {
                        Log.e("MyFilterInstaller", "filter.setReadable when copy from
                         ↪  asset is not complete properly");
                    }
                }
                catch(Exception unused_ex) {
                label_43:
                    Log.e("MyFilterInstaller", "copySelFileFromIntent : Exception is
                     ↪  occurred.");
                    if(v10 != null) {
                        try {
                            v10.close();
                        }
                        catch(IOException unused_ex) {
                            Log.e("MyFilterInstaller", "copySelFileFromIntent :
                             ↪  Exception is occurred.");
                        }

    //...

        try {
            v10.close();
        }
        catch(IOException unused_ex) {
            Log.e("MyFilterInstaller", "copySelFileFromIntent : Exception is
             ↪  occurred.");
            goto label_62;
        }
```

This service is protected by a signature permission:

```
    <service android:enabled="true" android:exported="true"
android:name="com.samsung.android.provider.filterprovider.MyFilterService"
android:permission="com.samsung.android.provider.filterprovider.permission.WRITE_FILTER">

      <intent-filter>

        <action
 ↪  android:name="com.samsung.android.provider.filterprovider.INSERT_MYFILTER"/>

        <action
android:name="com.samsung.android.provider.filterprovider.INSERT_MYFILTER_LIST"/>

        <action
 ↪  android:name="com.samsung.android.provider.filterprovider.DELETE_MYFILTER"/>

      </intent-filter>

    </service>
```

With this bug together, we can utilize the previous bug to call the service and get a straight forward file write, thus able to chaining with Mystique. Now we can call FilterProvider to overwrite any file as system app. The vulnerabilities have been assigned CVE-2021-25510, CVE-2021-25511 and fixed in Dec 2021's Samsung Security Bulletin.

2) Path traversal in Samsung FactoryAirCommandManager: The vulnerability is in *BluetoothSocketModule*, which has two subclasses *BluetoothClient* and *BluetoothServer*. A common function in this class allows attacker to send file and specify arbitrary path, from bluetooth socket, which can be force connected by attacker without user consent.

```
public void startSocket(Context context) {
        byte[] buffer = new byte[0x2000];
        byte[] newBuffer = new byte[0x2000];
        int maxLength = 0;
        ACUtil.log_i("BluetoothSocketModule", "startSocket");
        this.readSocket = new BufferedReader(new InputStreamReader(this.is,
        ↪  Charset.forName("UTF-8")));
        ACUtil.log_i("BluetoothSocketModule", "readSocket", "readSocket: " +
        ↪  this.readSocket);
        this.isRunning = true;
        try {
            while(true) {
            label_28:
                int read = this.is.read(buffer);
                if(read == -1) {
                    break;
                }
//...
                  if(this.isFileMode) {
                    this.getFileSize += read;
                    //...
                    this.fos.write(buffer, 0, read);
                    this.mHandler.sendEmptyMessage(1);
                    if(this.getFileSize < this.fileLength) {
                        goto label_28;
                    }

                    ACUtil.log_i("BluetoothSocketModule", "Socket", "end file mode");
                    this.mHandler.sendEmptyMessage(2);
                    this.isFileMode = false;
                    this.fos.close();
                    this.getFileSize = 0;
                    goto label_28;
                }
//...

                System.arraycopy(((Object)buffer), 0, ((Object)newBuffer), maxLength,
                ↪  read);
                maxLength += read;
                //...

                ACUtil.log_i("BluetoothSocketModule", "Socket", "string");
                byte[] v4_1 = new byte[maxLength];
                System.arraycopy(((Object)newBuffer), 0, ((Object)v4_1), 0,
                ↪  maxLength);
                String v8_2 = new String(v4_1, 0, maxLength, "UTF-8");
                if(v8_2.toUpperCase().contains("AT+FACMFILE")) {
                    this.isFileMode = true;
                    String subStr = v8_2.substring(v8_2.indexOf("=") + 1,
                    ↪  v8_2.length() - 2);
                    this.fileName = subStr.split(",")[0];
                    this.fileLength = Integer.parseInt(subStr.split(",")[1]);
```

```
ACUtil.log_i("BluetoothSocketModule", "Socket", "name: " +
↪  this.fileName + " length: " + this.fileLength);
File file = new File(BluetoothSocketModule.savePath);
if(!file.exists()) {
    file.mkdirs();
}

File v3_1 = new File(BluetoothSocketModule.savePath +
↪  this.fileName);
ACUtil.log_d("BluetoothSocketModule", "Socket", "Save: " +
↪  v3_1.getPath());
if(this.fileLength != 0) {
    this.fos = new FileOutputStream(v3_1);
    this.mHandler.sendEmptyMessage(0);
}
```

The file path is specified from bluetooth socket stream input with no check on path, so attacker can use this to overwrite arbitrary file in the context of *FactoryAirCommandManager*. What makes things worse is that this application owns the permission to force-accept any bluetooth pairing request, as the code in *BluetoothService* automatically retrieved and set the pairing pin. This enables an local zero-permission attacker to silently trigger the path traversal bug, without the need to trick user into manually confirming the bluetooth pairing.

The vulnerability has been assigned CVE-2021-25450 and fixed in September 2021's Samsung Security Bulletin.[5]

3) Similar bugs in other major vendors: We also applied static analysis on other major vendors such as Oppo, Xiaomi and got some promising results. Due to the disclosure process, we cannot write about the details for the found bugs, but it's not very hard to find one. For example, Google's research team disclosed a bug in Xiaomi's system application MIUI Powerkeeper App that also serves our purpose[7]. We will not elaborate again here.

II. Post-Exploitation

As described in the previous sections, those vulnerabilities combined can be used to replace the application code files under */data/app/*. To fully unleash the power of this chain, there are still several remaining crucial problems need to be discussed. We have two possible type of targets to operate on:

1) so library under /data/app/A/lib

2) base.apk in /data/app/A/

For the first type of targets, the malicious code can be injected to *so* directly. As there is no verification on the *so* files after an application is installed (*PackageManagerService* won't do that again), the injected code can remain intact until the target application is uninstalled or fully removed and updated. We use Frida-gadget for so injection so that we can control the behavior in payload js files dynamically and more easily.

The second approach is to replace the *base.apk* file with repackage target. The advantage of this approach is it's more flexible and can bypass some applications' custom verification in dynamic library loading and also works on application without dynamic library code. It's also easier to integrate payload directly into the application's dex file using smali and baksmali by writing smali and java code. However it has a significant drawback: *PackageManagerService* will do a re-verification on the *base.apk* upon each reboot, the repackaged file will fail in this process. In one word, this approach of injecting code cannot survive reboot, unlike the former one.

For attack on the first type of targets, we divide the full exploitation chain into preparation phase and execution phase. The preparation phrase includes shared library injection and adjusting based on the actual location of target app. When target APP is started, it will load the injected shared library, which will then start frida-gadget and run attack script.

The purpose of shared library injection is to construct an injected one with malicious code based on target app's chosen shared library(e.g. libtmessages.38.so of Telegram). Here we use LIEF project[2] to inject dynamic dependency(NEEDED tag in dynamic section of elf) on original library. Figure 1 shows the difference between original and injected dynamic library file.

One thing to be mentioned when using LIEF is that you need to use a long enough directory path to reserve enough space in injected elf file, as Android adds app name random suffix in installation path.

A. Attacker app

We implement an attacker app to automate the preparation phase, which includes run-time path adaption and shared library replacement.

1) Attacker app resource files contain injected shared library, frida gadget files (library and its configuration) and attack scripts. Figure 2 shows the details of these files and their location. Due to Android system's restriction, we need to put the injected library and frida configuration file, which need be update later, into a world readable directory.

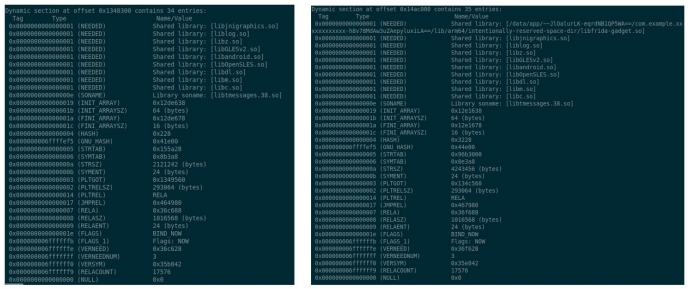

Fig. 1. SO Injection

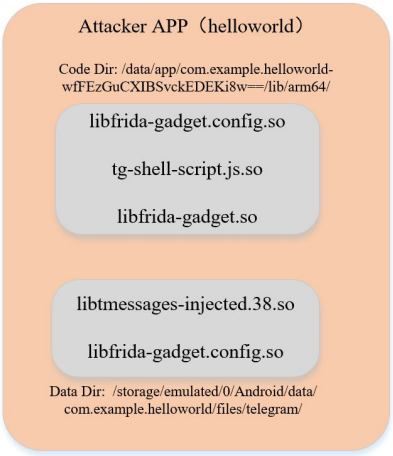

Fig. 2. Attacker APP Resources

2) App user interface is shown in Figure 3, supporting one-click installation and run-time field path adaption. As Android adds random suffix to app name, the attacker app needs to first update the dynamic dependency section in injected library, making it point to frida gadget library under *nativeLibraryDir*. Then the attacker app updates the script path in frida configuration file to make it point back to script file under *nativeLibraryDir*.

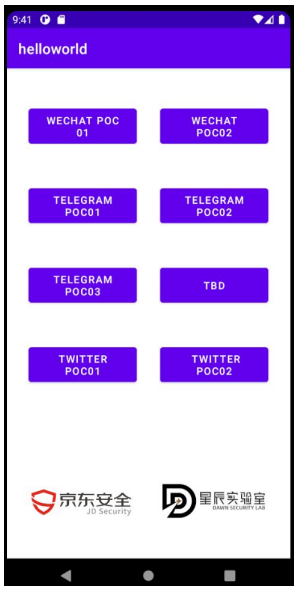

Fig. 3. Attacker APP UI

3) The final attack step is shown in the Figure 4. With the power of the *Mystique* chain, we first replace the target APP's original library with the injected one, then we replace the old frida configuration file under code directory to the updated one.

Finally after the attack is conducted, when target APP is started, injected library and frida gadget will be loaded and malicious script will be executed.

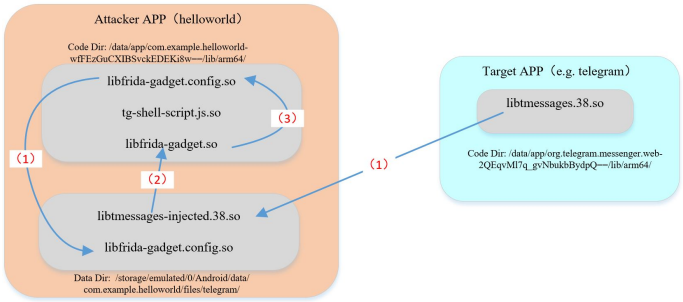

Fig. 4. Exploit Chain

B. Real-world Targets

This paper selects the popular chat software in the market to demonstrate the ability of this chain, including malicious pop-up window, hijacking chat messages and reverse shell. We use Telegram as an example here. First, we launch an attack on Telegram[6], using frida-gadget to call our javascript. To demonstrate malicious pop-up window, we hook an activity on Telegram and use *Toast.makeText().show()* function to pop-up messages; For hijacking chat messages, we reverse and analyse the Telegram software, and find the key function *processSendingText()*. Then we hook this function and modify the chat content; We can also hook the target software to create a */system/bin/sh* reverse shell. These attacks also applies to other chat software.

III. Detection

While this bug chain achieves similar ability with a universal rooting one, it does not have the trace of rooting exploits, which makes it hard to detect with existing tools. To tackle this problem, we have developed a universal inspection tool, which is used to determine whether the user's mobile phone is attacked by this or similar vulnerability in the wild. The detection process does not involve specific POC, and the result of the tool can be used as a reference.The detection logic is shown as below:

1)Enumerate the list of apps on the phone.

2)For each application, recursively traverse all files in its directory, and record the last modification timestamp of the file. After categorizing all files under each application according to the timestamp, if the number of categorized results exceeds two, it is determined that the application may be exploited by this bug chain.

3) For the application that is suspected of being attacked in step 2), we need to make further judgements. For example: we will check if the timestamp of the file is later than the installation time or the last modification time of the the application.

4) Repeat step 2) and 3) to summarize all potentially tampered application, and finally display it to the front-end activity.

IV. Conclusions

This paper presented the detail of the *Mystique* bug chain on Android 11, which allows a zero-permission local malicious application to gain almost all the other applications' privilege and read/write their private files. Due to the pure logical nature of the bugs (no memory corruption is needed), it's very stable on major vendors and poses a real threat for end users if it's exploited by malicious attacker. Due to the behavior of this chain differs from traditional privilege escalation bugs, this paper also discussed a potential mechanism to detect if a particular device has been attacked.

A demo video of this chain can be viewed at [4] and [3], showing attack from a local zero-privileged app toward Telegram, recorded on a fully patched Samsung device (at the time of report). The website for this chain is at https://dawnslab.jd.com/mystique-en/.

References

[1] Commit lead to the mystique bug. URL: https://android.googlesource.com/platform/system/sepolicy/+/020e3ab0356e21a0013689cdb4899bb6ff13d398.
[2] Library to instrument executable formats. URL: https://lief-project.github.io/doc/latest/tutorials/09_frida_lief.html.
[3] Mystique demo video for hijacking. URL: https://drive.google.com/file/d/1qxU2tGQV1onkaAK36Irt9MpkA0C8_PJo/view?usp=sharing.
[4] Mystique demo video for reverse shell. URL: https://drive.google.com/file/d/1kv8yNBfb_aknM3NzlGty4hAcEbO3XQpX/view?usp=sharing.
[5] Samsung security bulletin sep, 2021. URL: https://security.samsungmobile.com/securityUpdate.smsb.
[6] Telegram a new era of messaging. URL: https://telegram.org/.
[7] Xiaomi powerkeeper arbitrary file write. URL: https://bugs.chromium.org/p/apvi/issues/detail?id=50&q=&can=1.html.