“Dirty stream” attack: Discovering and mitigating a common vulnerability pattern in Android apps

microsoft.com/en-us/security/blog/2024/05/01/dirty-stream-attack-discovering-and-mitigating-a-common-vulnerability-pattern-in-android-apps/

May 1, 2024

Skip to main content

By

Microsoft discovered a path traversal-affiliated vulnerability pattern in multiple popular Android applications that could enable a malicious application to overwrite files in the vulnerable application’s home directory. The implications of this vulnerability pattern include arbitrary code execution and token theft, depending on an application’s implementation. Arbitrary code execution can provide a threat actor with full control over an application’s behavior. Meanwhile, token theft can provide a threat actor with access to the user’s accounts and sensitive data.

We identified several vulnerable applications in the Google Play Store that represented over four billion installations. We anticipate that the vulnerability pattern could be found in other applications. We're sharing this research so developers and publishers can check their apps for similar issues, fix as appropriate, and prevent introducing such vulnerabilities into new apps or releases. As threats across all platforms continue to evolve, industry collaboration among security researchers, security vendors, and the broader security community is essential in improving security for all. Microsoft remains committed to working with the security community to share vulnerability discoveries and threat intelligence to protect users across platforms.

After discovering this issue, we identified several vulnerable applications. As part of our responsible disclosure policy, we notified application developers through Coordinated Vulnerability Disclosure (CVD) via Microsoft Security Vulnerability Research (MSVR) and worked with them to address the issue. We would like to thank the Xiaomi, Inc. and WPS Office security teams for investigating and fixing the issue. As of February 2024, fixes have been deployed for the aforementioned apps, and users are advised to keep their device and installed applications up to date.

Recognizing that more applications could be affected, we acted to increase developer awareness of the issue by collaborating with Google to publish an article on the Android Developers website, providing guidance in a high-visibility location to help developers avoid introducing this vulnerability pattern into their applications. We also wish to thank Google's Android Application Security Research team for their partnership in resolving this issue.

In this blog post, we continue to raise developer and user awareness by giving a general overview of the vulnerability pattern, and then focusing on Android share targets, as they are the most prone to these types of attacks. We go through an actual code execution case study where we demonstrate impact that extends beyond the mobile device's scope and could even affect a local network. Finally, we provide guidance to users and application developers and illustrate the importance of collaboration to improve security for all.

Overview: Data and file sharing on Android

The Android operating system enforces isolation by assigning each application its own dedicated data and memory space. To facilitate data and file sharing, Android provides a component called a content provider, which acts as an interface for managing and exposing data to the rest of the installed applications in a secure manner. When used correctly, a content provider provides a reliable solution. However, improper implementation can introduce vulnerabilities that could enable bypassing of read/write restrictions within an application's home directory.

The Android software development kit (SDK) includes the *FileProvider* class, a subclass of *ContentProvider* that enables file sharing between installed applications. An application that needs to share its files with other applications can declare a *FileProvider* in its app manifest and declare the specific paths to share.

Every file provider has a property called *authority*, which identifies it system-wide, and can be used by the consumer (the app that wants to access the shared files) as a form of address. This content-based model bears a strong resemblance to the web model, but instead of the *http* scheme, consumers utilize the *content* scheme along with the *authority*, followed by a pseudo-path to the file that they want to access.

For example, assuming that the application *com.example.server* shares some files under the *file:///data/data/com.example.server/files/images* directory that it has previously declared as shared using the name *shared_images,* a consumer can use the *content://[authority]/shared_images/[sub-path]/[filename] URI* to index these files.

Access is given by the data sharing application most commonly using the *grantUriPermissions* attribute of the Android manifest, in combination with special flags that are used to define a read or write mode of operation. The data sharing application creates and sends an intent to the consumer that provides temporary fine-grained access to a file. Finally, when a provider receives a file access request, it resolves the actual file path that corresponds to the incoming URI and returns a file descriptor to it.

Implementation pitfalls

This content provider-based model provides a well-defined file-sharing mechanism, enabling a serving application to share its files with other applications in a secure manner with fine-grained control. However, we have frequently encountered cases where the consuming application doesn't validate the content of the file that it receives and, most concerning, it uses the filename provided by the serving application to cache the received file within the consuming application's internal data directory. If the serving application implements its own malicious version of *FileProvider*, it may be able to cause the consuming application to overwrite critical files.

Share targets

In simple terms, a share target is an Android app that declares itself to handle data and files sent by other apps. Common application categories that can be share targets include mail clients, social networking apps, messaging apps, file editors, browsers, and so on. In a common scenario, when a user clicks on a file, the Android operating system triggers the share-sheet dialog asking the user to select the component that the file should be sent to:

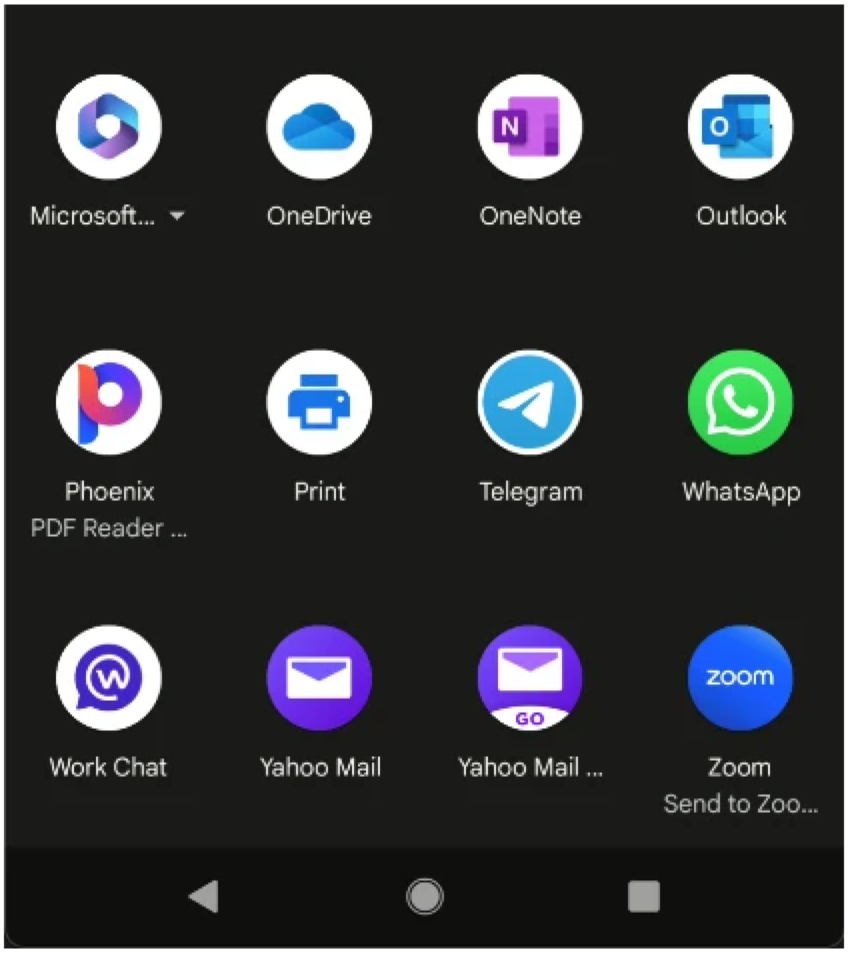


Figure 1. The Android share sheet dialog

While this type of guided file-sharing interaction itself may not trigger a successful attack against a share target, a malicious Android application can create a custom, explicit intent and send a file directly to a share target with a malicious filename and without the user's knowledge or approval. Essentially, the malicious application is substituting its own malicious *FileProvider* implementation and provides a filename that is improperly trusted by the consuming application.

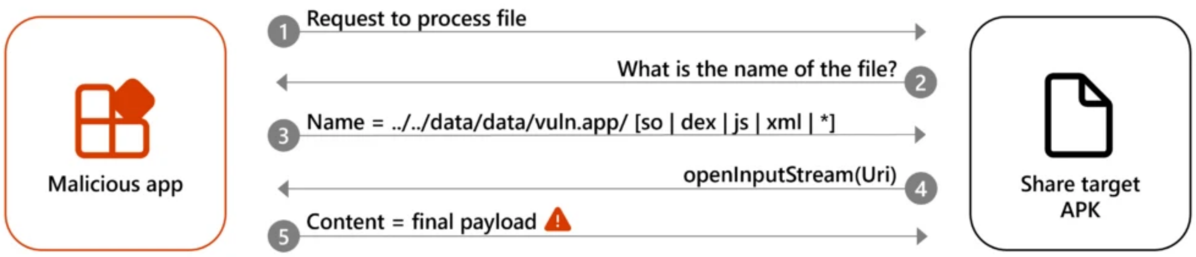

Malicious app
1 Request to process file
What is the name of the file? 2
3 Name = ../../data/data/vuln.app/ [so | dex | js | xml | *]
openInputStream(Uri) 4
5 Content = final payload
Share target APK

Figure 2. Dirty stream attack

In Figure 2, the malicious app, on the left, creates an explicit intent that targets the file processing component of the share target, on the right, and attaches a content URI as an intent's extra. It then sends this intent to the share target using the *startActivity* API call.

After this point, most of the share targets that we have reviewed seem to follow a specific code pattern that includes the following steps:

1. Request the actual filename from the remote file provider
2. Use this filename to initialize a file that is subsequently used to initialize a file output stream
3. Create an input stream using the incoming content URI
4. Copy the input stream to the output stream

Since the rogue app controls the name as well as the content of the file, by blindly trusting this input, a share target may overwrite critical files in its private data space, which may lead to serious consequences.

Impact

We identified this vulnerability pattern in the then-current versions of several Android applications published on the Google Play Store, including at least four with more than 500 million installations each. In each case, we responsibly disclosed to the vendor. Two example vulnerable applications that we identified are Xiaomi Inc.’s File Manager (1B+ installs) and WPS Office (500M+ installs).

In Xiaomi Inc.’s File Manager, we were able to obtain arbitrary code execution in version V1-210567. After our disclosure, Xiaomi published version V1-210593, and we verified that the vulnerability has been addressed. In WPS Office, we were able to obtain arbitrary code execution in version 16.8.1. After our disclosure, WPS published and informed us that the vulnerability has been addressed as of version 17.0.0.

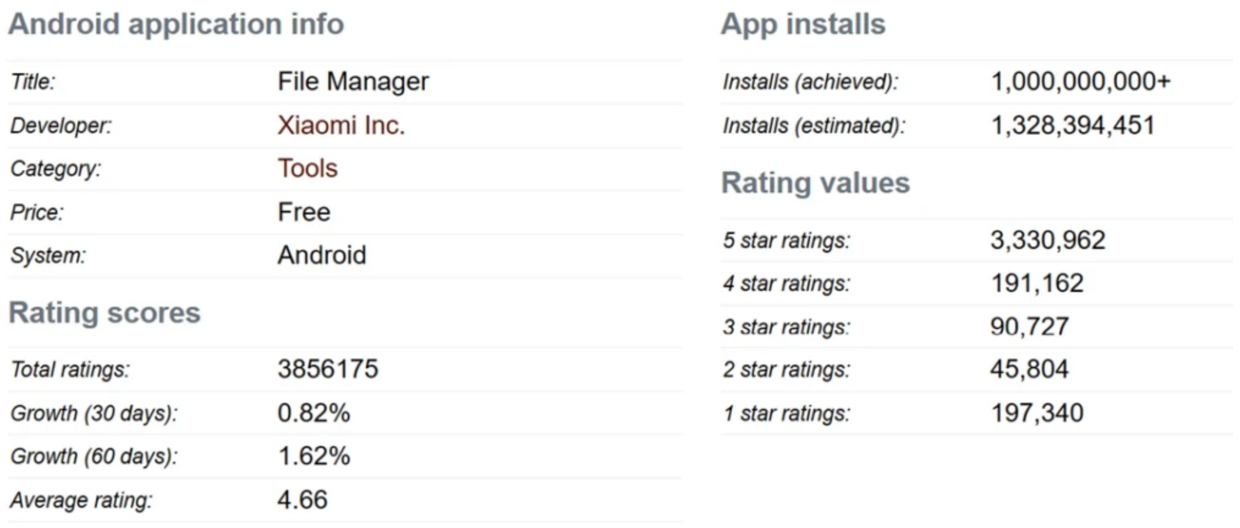
The potential impact varies depending on implementation specifics. For example, it’s very common for Android applications to read their server settings from the *shared_prefs* directory. In such cases, the malicious app can overwrite these settings, causing the vulnerable app to communicate with an attacker-controlled server and send the user’s authentication tokens or other sensitive information.

In a worst-case (and not so uncommon) scenario, the vulnerable application might load native libraries from its data directory (as opposed to the more secure */data/app-lib* directory, where the libraries are protected from modification). In this case, the malicious application can overwrite a native library with malicious code that gets executed when the library is loaded. In the following section, we use Xiaomi Inc.’s File Manager to illustrate this case. We demonstrated the ability for a malicious application to overwrite the application’s shared preferences, write a native library to the application’s internal storage, and cause the application to load the library. These actions provided arbitrary code execution with the file manager’s user ID and permissions.

In the following sections, we focus on this case and delve into the technical details of this vulnerability pattern.

Case study: Xiaomi Inc.’s File Manager

Xiaomi Inc.’s File Manager is the default file manager application for Xiaomi devices and is published under the package name *com.mi.android.globalFileexplorer* on the Google Play Store, where it has been installed over one billion times.



Android application info

Title:	File Manager
Developer:	Xiaomi Inc.
Category:	Tools
Price:	Free
System:	Android

Rating scores

Total ratings:	3856175
Growth (30 days):	0.82%
Growth (60 days):	1.62%
Average rating:	4.66

App installs

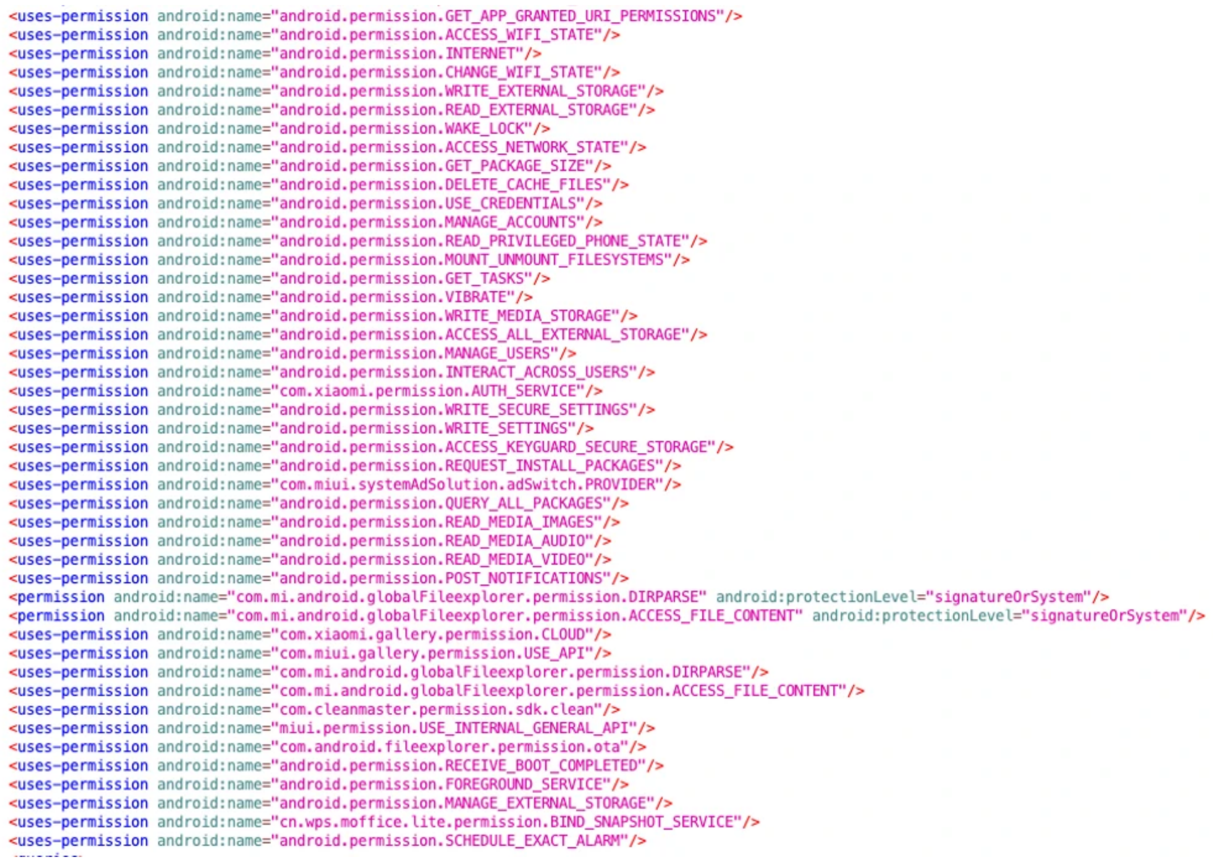
Installs (achieved):	1,000,000,000+
Installs (estimated):	1,328,394,451

Rating values

5 star ratings:	3,330,962
4 star ratings:	191,162
3 star ratings:	90,727
2 star ratings:	45,804
1 star ratings:	197,340

Figure 3. Xiaomi’s File Manager profile according to Android rank (source: File Manager)

Besides having full access to the device’s external storage, the application requests many permissions, including the ability to install other applications:



```
<uses-permission android:name="android.permission.GET_APP_GRANTED_URI_PERMISSIONS"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.GET_PACKAGE_SIZE"/>
<uses-permission android:name="android.permission.DELETE_CACHE_FILES"/>
<uses-permission android:name="android.permission.USE_CREDENTIALS"/>
<uses-permission android:name="android.permission.MANAGE_ACCOUNTS"/>
<uses-permission android:name="android.permission.READ_PRIVILEGED_PHONE_STATE"/>
<uses-permission android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS"/>
<uses-permission android:name="android.permission.GET_TASKS"/>
<uses-permission android:name="android.permission.VIBRATE"/>
<uses-permission android:name="android.permission.WRITE_MEDIA_STORAGE"/>
<uses-permission android:name="android.permission.ACCESS_ALL_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.MANAGE_USERS"/>
<uses-permission android:name="android.permission.INTERACT_ACROSS_USERS"/>
<uses-permission android:name="com.xiaomi.permission.AUTH_SERVICE"/>
<uses-permission android:name="android.permission.WRITE_SECURE_SETTINGS"/>
<uses-permission android:name="android.permission.WRITE_SETTINGS"/>
<uses-permission android:name="android.permission.ACCESS_KEYGUARD_SECURE_STORAGE"/>
<uses-permission android:name="android.permission.REQUEST_INSTALL_PACKAGES"/>
<uses-permission android:name="com.miui.systemAdSolution.adSwitch.PROVIDER"/>
<uses-permission android:name="android.permission.QUERY_ALL_PACKAGES"/>
<uses-permission android:name="android.permission.READ_MEDIA_IMAGES"/>
<uses-permission android:name="android.permission.READ_MEDIA_AUDIO"/>
<uses-permission android:name="android.permission.READ_MEDIA_VIDEO"/>
<uses-permission android:name="android.permission.POST_NOTIFICATIONS"/>
<permission android:name="com.mi.android.globalFileexplorer.permission.DIRPARSE" android:protectionLevel="signatureOrSystem"/>
<permission android:name="com.mi.android.globalFileexplorer.permission.ACCESS_FILE_CONTENT" android:protectionLevel="signatureOrSystem"/>
<uses-permission android:name="com.xiaomi.gallery.permission.CLOUD"/>
<uses-permission android:name="com.miui.gallery.permission.USE_API"/>
<uses-permission android:name="com.mi.android.globalFileexplorer.permission.DIRPARSE"/>
<uses-permission android:name="com.mi.android.globalFileexplorer.permission.ACCESS_FILE_CONTENT"/>
<uses-permission android:name="com.cleanmaster.permission.sdk.clean"/>
<uses-permission android:name="miui.permission.USE_INTERNAL_GENERAL_API"/>
<uses-permission android:name="com.android.fileexplorer.permission.ota"/>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
<uses-permission android:name="android.permission.MANAGE_EXTERNAL_STORAGE"/>
<uses-permission android:name="cn.wps.moffice.lite.permission.BIND_SNAPSHOT_SERVICE"/>
<uses-permission android:name="android.permission.SCHEDULE_EXACT_ALARM"/>
```

Figure 4. A snapshot of the application's permissions

Further, it offers a junk files cleaner plugin as well as the ability to connect to remote FTP and SMB shares:

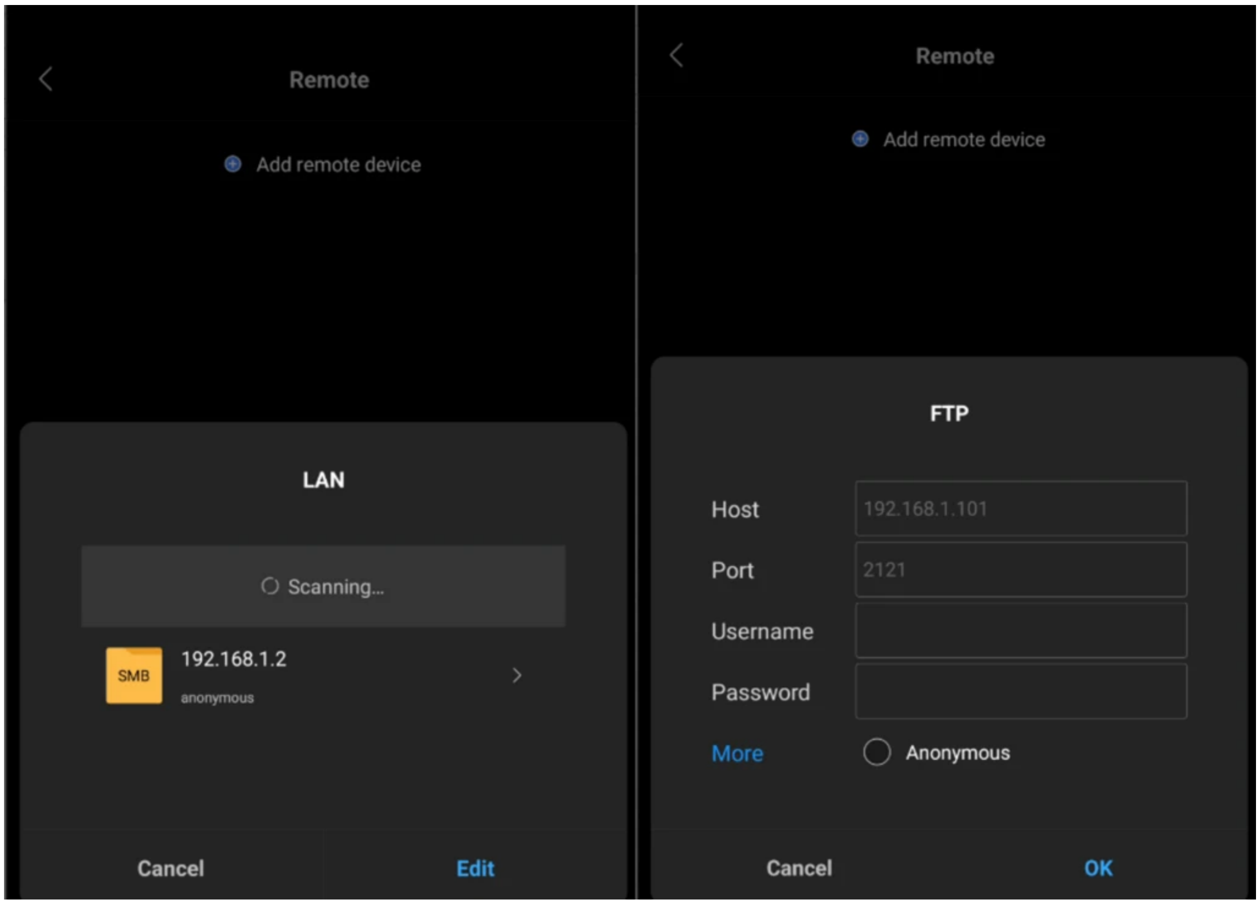


Figure 5. Connecting to remote shares using the file manager

Vulnerability assessment findings

During our investigation, we identified that the application exports the *CopyFileActivity*, an activity alias of the *com.android.fileexplorer.activity.FileActivity*, which is used to handle copy-from-to file operations:

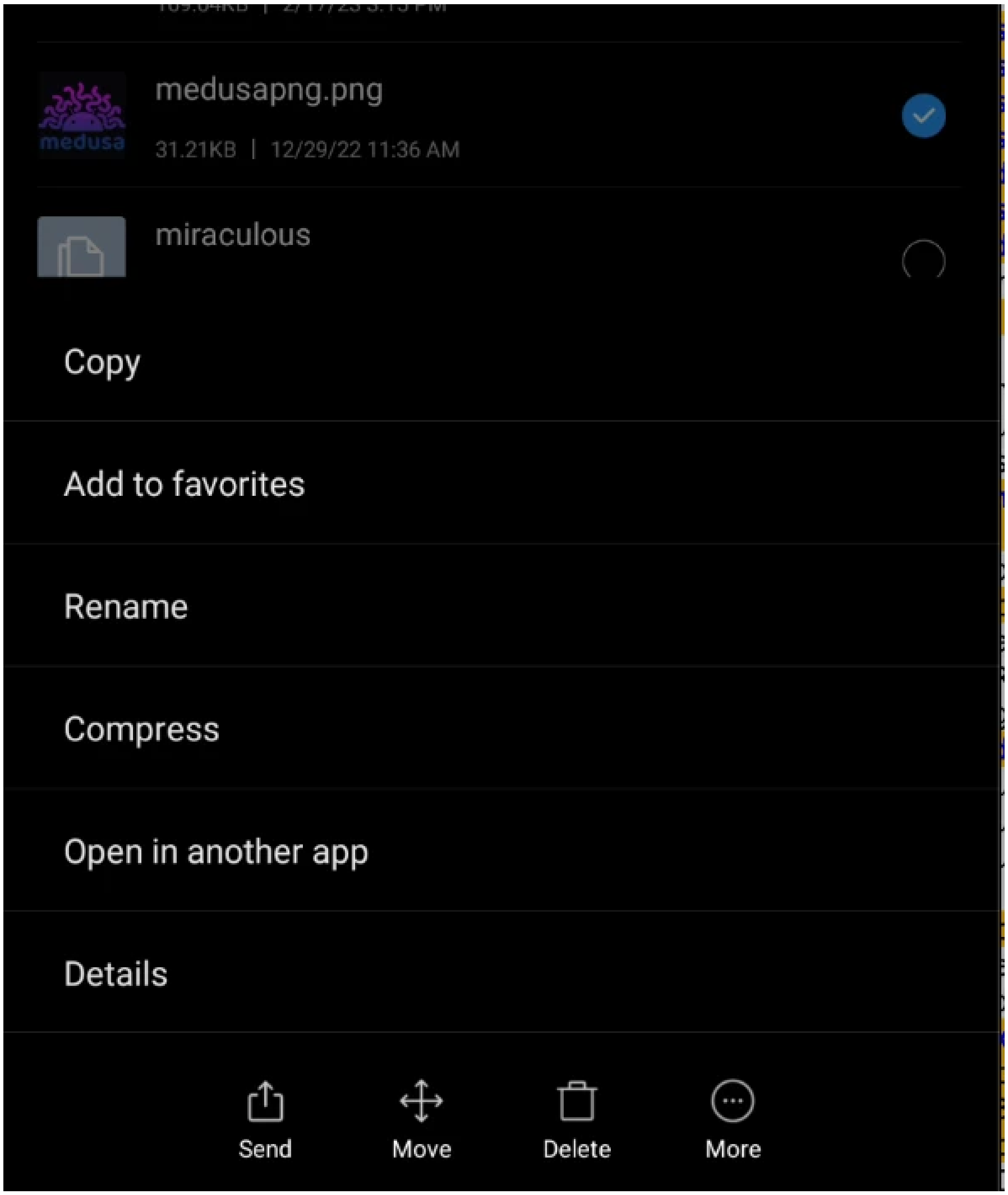
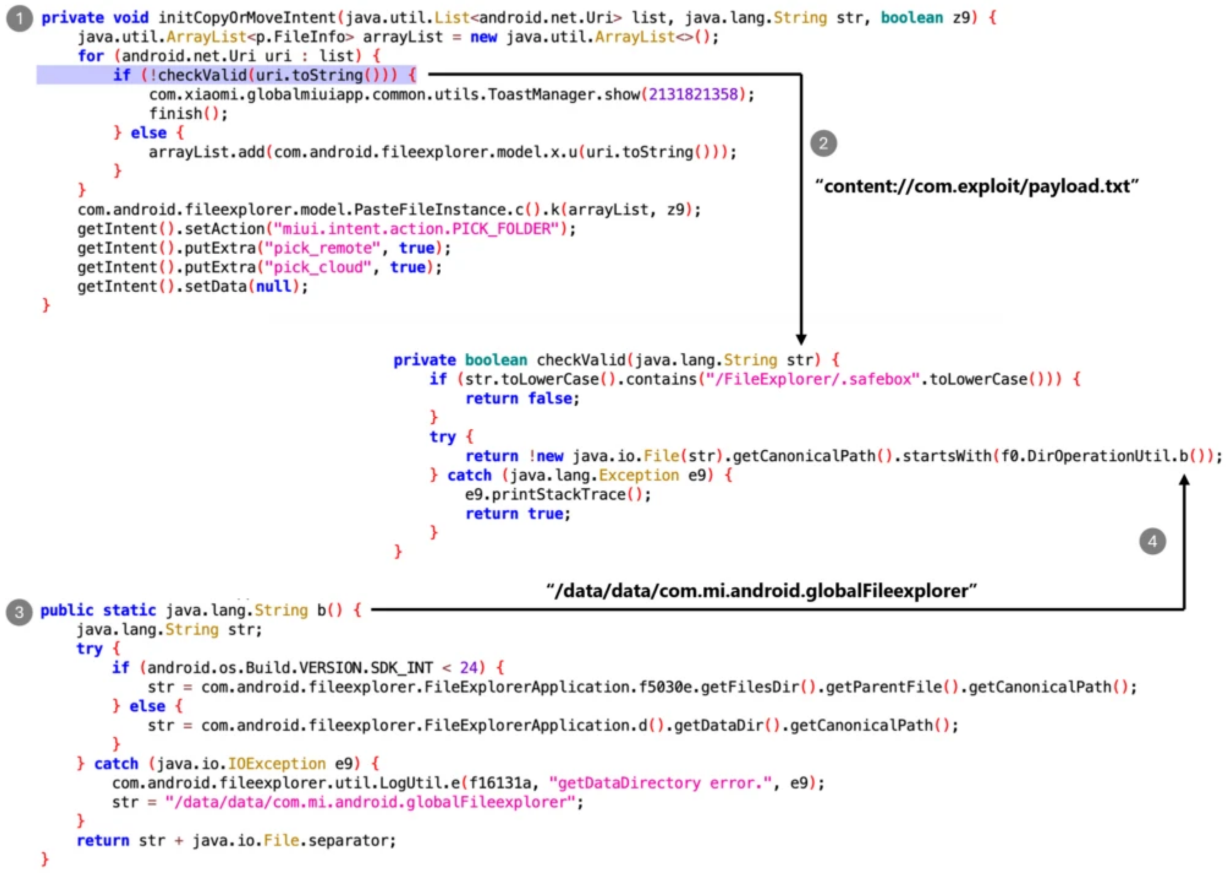


Figure 6. Triggering the copy to CopyFileActivity

Since this activity is exported, it can be triggered by any application installed on the same device by using an explicit intent of action *SEND* or *SEND_MULTIPLE* and attaching a content URI corresponding to a file stream.

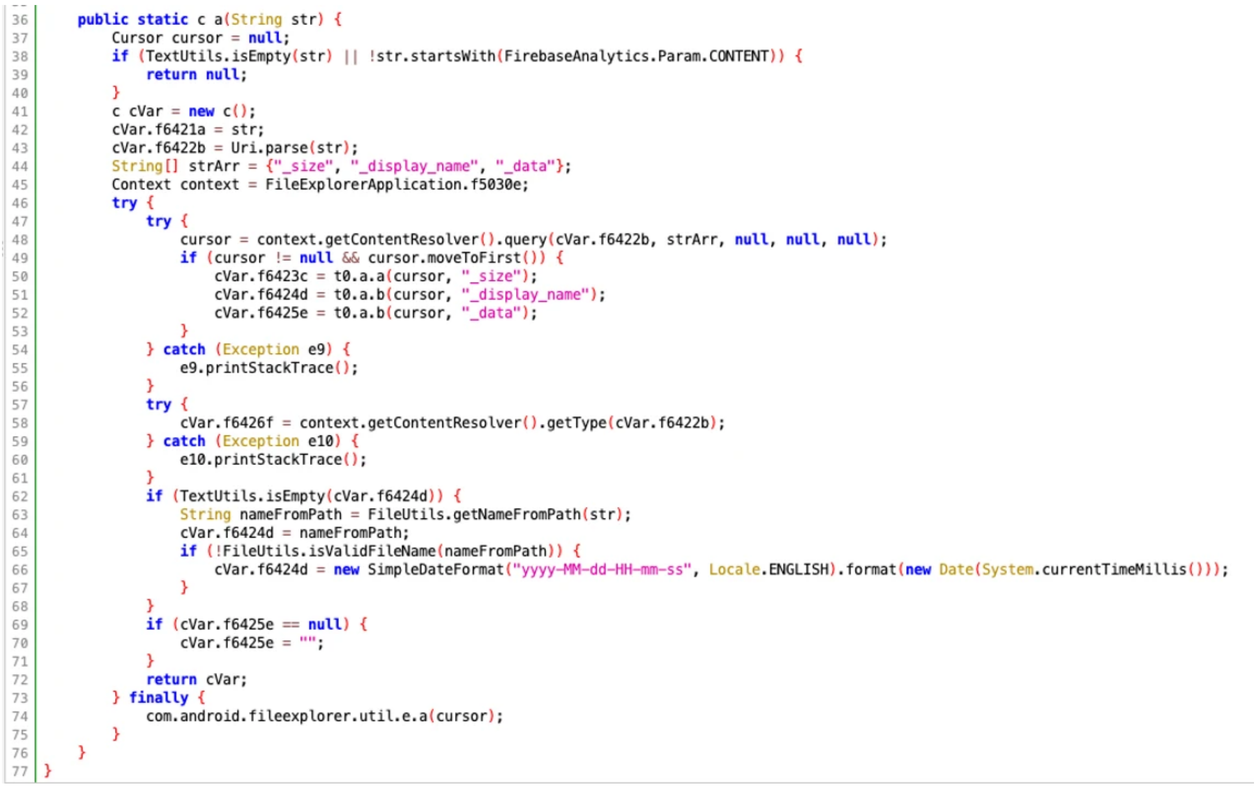
Upon receiving such an intent, the browser performs a validity check, which we found to be insufficient:


1
2
"content://com.exploit/payload.txt"
3
4
"/data/data/com.mi.android.globalFileexplorer"

```
private void initCopyOrMoveIntent(java.util.List<android.net.Uri> list, java.lang.String str, boolean z9) {
    java.util.ArrayList<p.FileInfo> arrayList = new java.util.ArrayList<>();
    for (android.net.Uri uri : list) {
        if (!checkValid(uri.toString())) {
            com.xiaomi.globalmiuiapp.common.utils.ToastManager.show(2131821358);
            finish();
        } else {
            arrayList.add(com.android.fileexplorer.model.x.u(uri.toString()));
        }
    }
    com.android.fileexplorer.model.PasteFileInstance.c().k(arrayList, z9);
    getIntent().setAction("miui.intent.action.PICK_FOLDER");
    getIntent().putExtra("pick_remote", true);
    getIntent().putExtra("pick_cloud", true);
    getIntent().setData(null);
}

private boolean checkValid(java.lang.String str) {
    if (str.toLowerCase().contains("/FileExplorer/.safebox".toLowerCase())) {
        return false;
    }
    try {
        return !new java.io.File(str).getCanonicalPath().startsWith(f0.DirOperationUtil.b());
    } catch (java.lang.Exception e9) {
        e9.printStackTrace();
        return true;
    }
}

public static java.lang.String b() {
    java.lang.String str;
    try {
        if (android.os.Build.VERSION.SDK_INT < 24) {
            str = com.android.fileexplorer.FileExplorerApplication.f5030e.getFilesDir().getParentFile().getCanonicalPath();
        } else {
            str = com.android.fileexplorer.FileExplorerApplication.d().getDataDir().getCanonicalPath();
        }
    } catch (java.io.IOException e9) {
        com.android.fileexplorer.util.LogUtil.e(f16131a, "getDataDirectory error.", e9);
        str = "/data/data/com.mi.android.globalFileexplorer";
    }
    return str + java.io.File.separator;
}
```

Figure 7. Validating an incoming copy file request

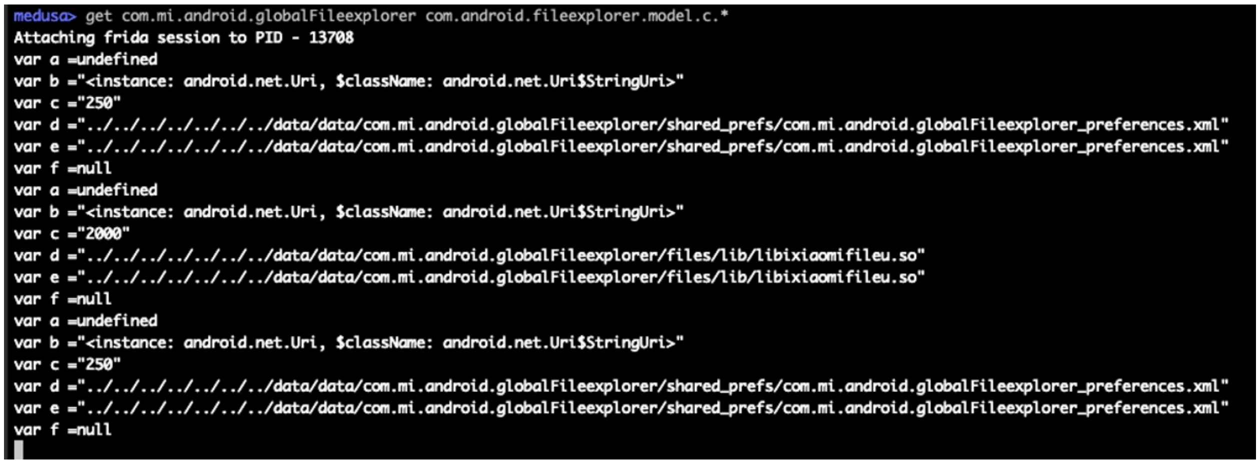
As depicted above, the *initCopyOrMoveIntent* method calls the *checkValid* method passing as an argument a content URI (steps 1 and 2). However, the *checkValid* method is designed to handle a file path, not a content URI. It always returns true for a content URI. Instead, a safer practice is to parse the string as a URI, including ensuring the scheme is the expected value (in this case, *file*, not *content*).The *checkValid* method verifies that the copy or move operation doesn't affect the private directory of the app, by initializing a file object using the incoming string as an argument to the *File* class constructor and comparing its canonical path with the path that corresponds to the home directory of the application (steps 3 and 4). Given a content URI as a path, the *File* constructor normalizes it (following a Unix file system normalization), thus the *getCanonicalPath* method returns a string starting with "*/content:/*", which will always pass the validity check. More specifically, the app performs a query to the remote content provider for the *_size*, *_display_name* and *_data* columns (see line 48 below). Then it uses the values returned by these rows to initialize the fields of an object of the *com.android.fileexplorer.mode.c* class:



```java
    public static c a(String str) {
        Cursor cursor = null;
        if (TextUtils.isEmpty(str) || !str.startsWith(FirebaseAnalytics.Param.CONTENT)) {
            return null;
        }
        c cVar = new c();
        cVar.f6421a = str;
        cVar.f6422b = Uri.parse(str);
        String[] strArr = {"_size", "_display_name", "_data"};
        Context context = FileExplorerApplication.f5030e;
        try {
            try {
                cursor = context.getContentResolver().query(cVar.f6422b, strArr, null, null, null);
                if (cursor != null && cursor.moveToFirst()) {
                    cVar.f6423c = t0.a.a(cursor, "_size");
                    cVar.f6424d = t0.a.b(cursor, "_display_name");
                    cVar.f6425e = t0.a.b(cursor, "_data");
                }
            } catch (Exception e9) {
                e9.printStackTrace();
            }
            try {
                cVar.f6426f = context.getContentResolver().getType(cVar.f6422b);
            } catch (Exception e10) {
                e10.printStackTrace();
            }
            if (TextUtils.isEmpty(cVar.f6424d)) {
                String nameFromPath = FileUtils.getNameFromPath(str);
                cVar.f6424d = nameFromPath;
                if (!FileUtils.isValidFileName(nameFromPath)) {
                    cVar.f6424d = new SimpleDateFormat("yyyy-MM-dd-HH-mm-ss", Locale.ENGLISH).format(new Date(System.currentTimeMillis()));
                }
            }
            if (cVar.f6425e == null) {
                cVar.f6425e = "";
            }
            return cVar;
        } finally {
            com.android.fileexplorer.util.e.a(cursor);
        }
    }
}
```

Figure 8. Getting file metadata from the remote content provider

Given the case that the *_display_name* and *_data* values, returned from the external file provider, are relative paths to the destination directory, after exiting from the method above, these class fields will contain values like the ones depicted below:



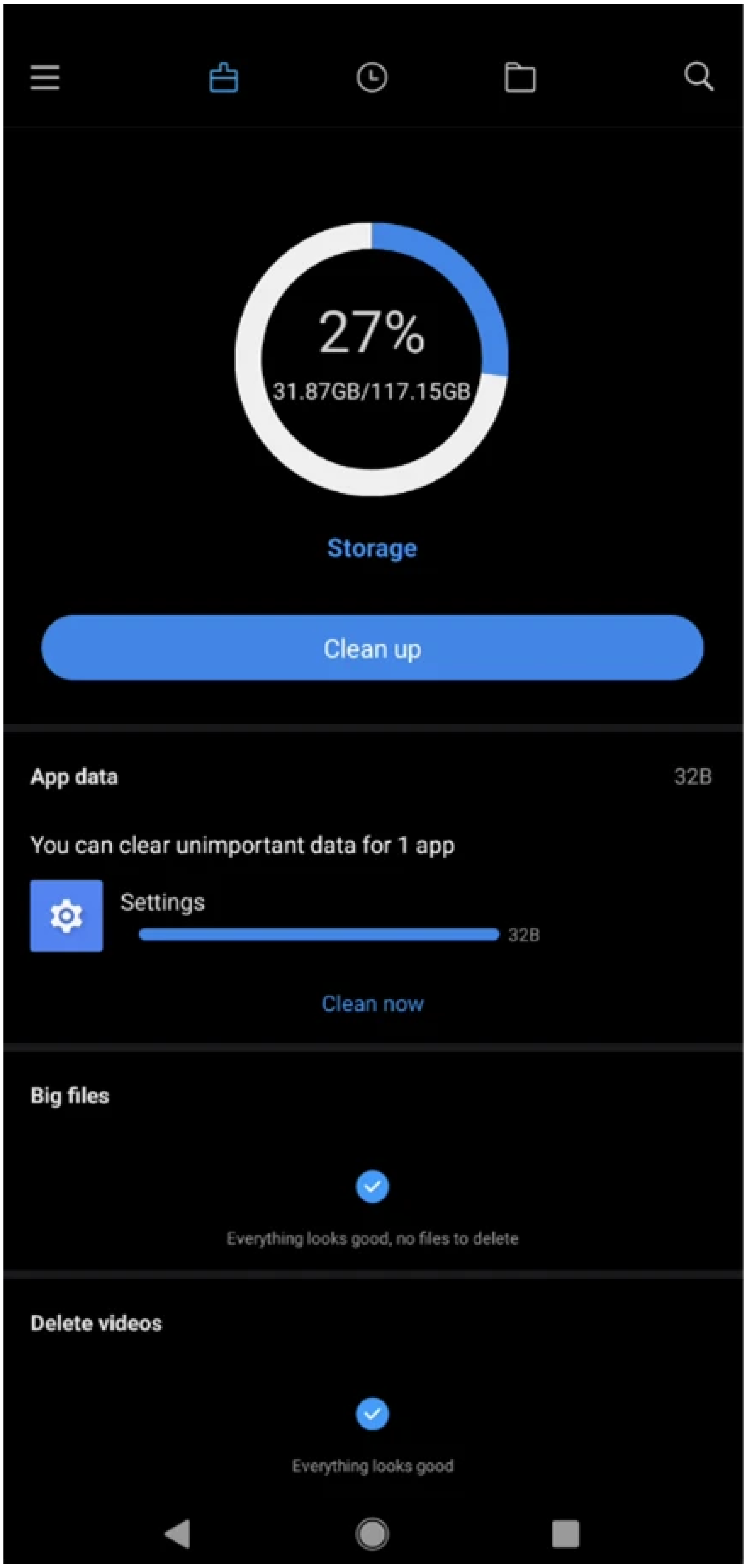
```
medusa> get com.mi.android.globalFileexplorer com.android.fileexplorer.model.c.*
Attaching frida session to PID - 13708
var a =undefined
var b ="<instance: android.net.Uri, $className: android.net.Uri$StringUri>"
var c ="250"
var d ="../../../../../../../data/data/com.mi.android.globalFileexplorer/shared_prefs/com.mi.android.globalFileexplorer_preferences.xml"
var e ="../../../../../../../data/data/com.mi.android.globalFileexplorer/shared_prefs/com.mi.android.globalFileexplorer_preferences.xml"
var f =null
var a =undefined
var b ="<instance: android.net.Uri, $className: android.net.Uri$StringUri>"
var c ="2000"
var d ="../../../../../../../data/data/com.mi.android.globalFileexplorer/files/lib/libixiaomifileu.so"
var e ="../../../../../../../data/data/com.mi.android.globalFileexplorer/files/lib/libixiaomifileu.so"
var f =null
var a =undefined
var b ="<instance: android.net.Uri, $className: android.net.Uri$StringUri>"
var c ="250"
var d ="../../../../../../../data/data/com.mi.android.globalFileexplorer/shared_prefs/com.mi.android.globalFileexplorer_preferences.xml"
var e ="../../../../../../../data/data/com.mi.android.globalFileexplorer/shared_prefs/com.mi.android.globalFileexplorer_preferences.xml"
var f =null
```

Figure 9. The file model initialized after calling the method a

As shown above, the paths (variables *d* and *e*) of this file-model point to files within the home directory of the application, thus the file streams attached to the incoming intent are going to be written under the specific locations.

Getting code execution

As previously mentioned, the application uses a plugin to clean the device's junk files:



App data 32B

You can clear unimportant data for 1 app

Settings 32B

Big files

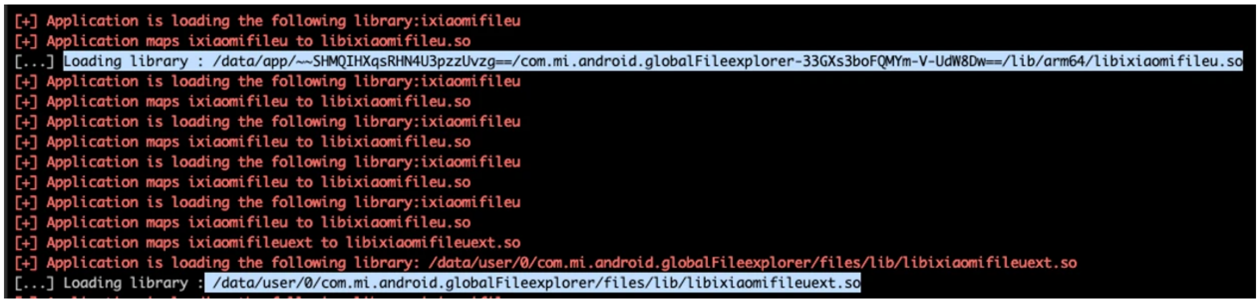
Everything looks good, no files to delete

Delete videos

Everything looks good

Figure 10. The junk files cleaner plugin user interface

When the application loads this plugin, it makes use of two native libraries: *libixiaomifileu.so*, which fetches from the */data/app* directory, and *libixiaomifileuext.so* from the home directory:



```
[+] Application is loading the following library:ixiaomifileu
[+] Application maps ixiaomifileu to libixiaomifileu.so
[...] Loading library : /data/app/~~SHMQIHXqsRHN4U3pzzUvzg==/com.mi.android.globalFileexplorer-33GXs3boFQMYm-V-UdW8Dw==/lib/arm64/libixiaomifileu.so
[+] Application is loading the following library:ixiaomifileu
[+] Application maps ixiaomifileu to libixiaomifileu.so
[+] Application is loading the following library:ixiaomifileu
[+] Application maps ixiaomifileu to libixiaomifileu.so
[+] Application is loading the following library:ixiaomifileu
[+] Application maps ixiaomifileu to libixiaomifileu.so
[+] Application is loading the following library:ixiaomifileu
[+] Application maps ixiaomifileu to libixiaomifileu.so
[+] Application maps ixiaomifileuext to libixiaomifileuext.so
[+] Application is loading the following library: /data/user/0/com.mi.android.globalFileexplorer/files/lib/libixiaomifileuext.so
[...] Loading library : /data/user/0/com.mi.android.globalFileexplorer/files/lib/libixiaomifileuext.so
```

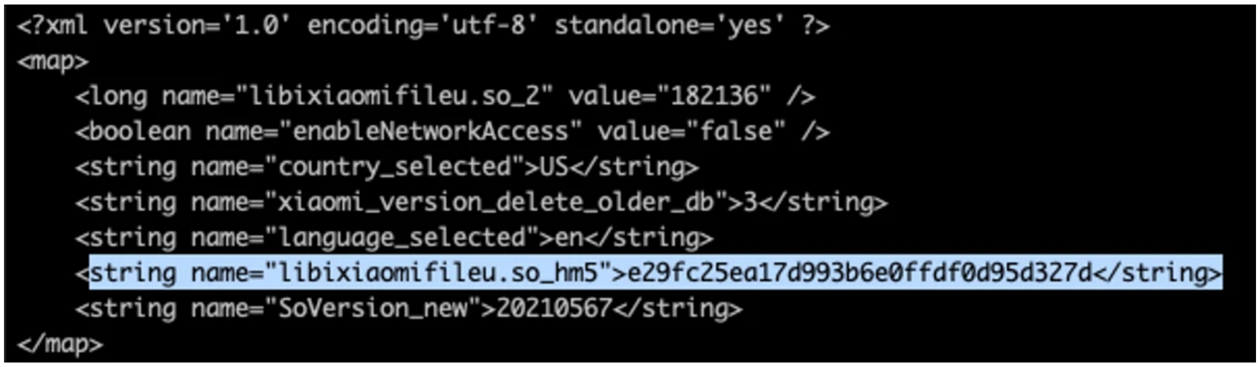
Figure 11. Tracing the loaded native libraries using medusa

As apps don't have write access to the */data/app* folder, the *libixiaomifileu.so* file stored there cannot be replaced. The easiest way to get code execution is to replace the *libixiaomifileuext.so* with a malicious one*.* However, an attempt to do so would fail since **in this particular case, the vulnerability that we described can only be used to write new files within the home directory, not overwrite existing files.** Our next inquiry was to determine how the application loads the *libixiaomifileu.so.*

Our assessment showed that before the application loads this library, it follows the following steps:

1. Calculate the hash of the file *libixiaomifileu.so,* located in the */data/app* directory

2. Compare this hash with the value assigned to the "*libixiaomifileu.so_hm5*" string, fetched from the *com.mi.android.globalFileexprorer_preferences.xml* file



```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <long name="libixiaomifileu.so_2" value="182136" />
    <boolean name="enableNetworkAccess" value="false" />
    <string name="country_selected">US</string>
    <string name="xiaomi_version_delete_older_db">3</string>
    <string name="language_selected">en</string>
    <string name="libixiaomifileu.so_hm5">e29fc25ea17d993b6e0ffdf0d95d327d</string>
    <string name="SoVersion_new">20210567</string>
</map>
```

Figure 12. the com.mi.android.globalFileexprorer_preferences.xml

3. If the values don't match, search for the *libixiaomifileu.so* file in the */files/lib* path in the home directory

4. If the file is found there, calculate its hash and compare it again with the value from the *shared_preferences* folder

5. If the hashes match, load the file under the */files/lib* using the *System.load* method

Given this behavior, in order to get code execution with the file manager's user ID, an attacker must take the following steps:

1. Use the path traversal vulnerability to save a malicious library as */files/lib/libixiaomifileu.so* (the file does not already exist in that directory, so overwriting is not an issue)

2. Calculate the hash of this library to replace the value of the *libixiaomifileu.so_hm5* string

3. Trigger the junk cleaner plugin with an explicit intent, since the activity that loads the native libraries is exported

An acute reader might have noticed that the second step requires the attacker to force the browser to overwrite the *com.mi.android.globalFileexprorer_preferences.xml,* which, as we already mentioned, was not possible.

To overcome this restriction, we referred to the actual implementation of the *SharedPreferences* class, where we found that when an Android application uses the *getSharedPreferences* API method to retrieve an instance of the *SharedPreferences* class, giving the name of the shared preferences file as an argument, then the constructor of the *SharedPreferencesImpl* class performs the following steps:

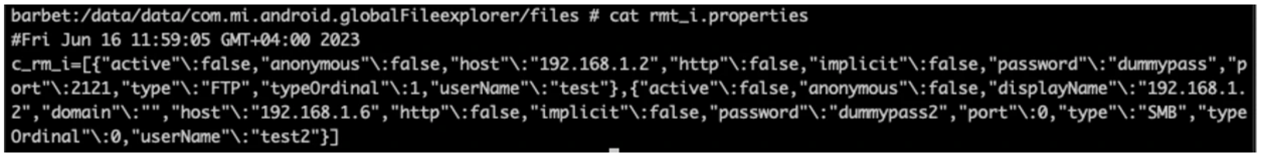
1. Create a new file object using the name provided to the *getSharedPreferences* method, followed by the .xml extension, followed by the .bak extension

2. Check if this file exists, and in case it does, delete the original xml file and replace it with the one created in the first step

Through this behavior, we were able to save the *com.mi.android.globalFileexprorer_preferences.xml.bak* under the shared preferences folder (as during the application's runtime it is unlikely to exist), so when the app tried to verify the hash, the original xml file was already replaced by our own copy. After this point, by using a single intent to start the junk cleaner plugin, we were able to trick the application to load the malicious library instead of the one under the */data/app* folder and get code execution with the browser's user ID.

Impact

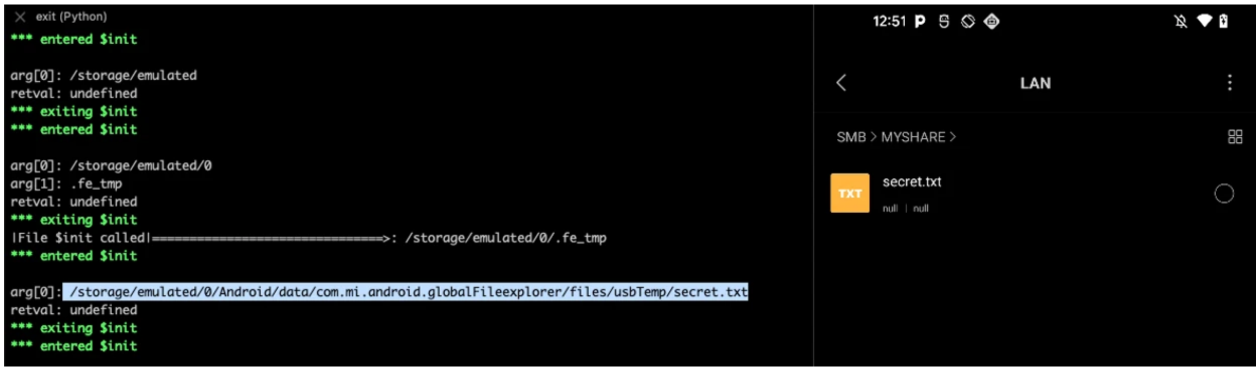
One reason we chose to use this app as a showcase is because the impact extends beyond the user's mobile device. The application gives the option to connect to remote file shares using the FTP and SMB protocols and the user credentials are saved in clear text in the */data/data/com.mi.android.globalFileexplorer/files/rmt_i.properties* file:



```
barbet:/data/data/com.mi.android.globalFileexplorer/files # cat rmt_i.properties
#Fri Jun 16 11:59:05 GMT+04:00 2023
c_rm_i=[{"active"\:false,"anonymous"\:false,"host"\:"192.168.1.2","http"\:false,"implicit"\:false,"password"\:"dummypass","p
ort"\:2121,"type"\:"FTP","typeOrdinal"\:1,"userName"\:"test"},{"active"\:false,"anonymous"\:false,"displayName"\:"192.168.1.
2","domain"\:"","host"\:"192.168.1.6","http"\:false,"implicit"\:false,"password"\:"dummypass2","port"\:0,"type"\:"SMB","type
Ordinal"\:0,"userName"\:"test2"}]
```

Figure 13. SMB/FTP credentials saved in clear text

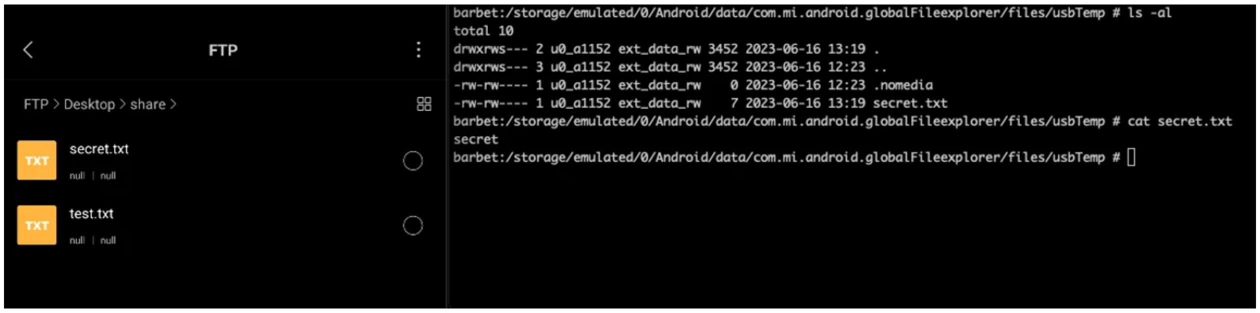
If a third party app was able to exploit this vulnerability and obtain code execution, an attacker could retrieve these credentials. The impact would then extend even further, since by the time that a user requests to open a remote share, the browser creates the directory */sdcard/Android/data/com.mi.android.globalFileexplorer/files/usbTemp/* where it saves the files that the user retrieves:



```
exit (Python)
*** entered $init

arg[0]: /storage/emulated
retval: undefined
*** exiting $init
*** entered $init

arg[0]: /storage/emulated/0
arg[1]: .fe_tmp
retval: undefined
*** exiting $init
|File $init called|=================================>: /storage/emulated/0/.fe_tmp
*** entered $init

arg[0]: /storage/emulated/0/Android/data/com.mi.android.globalFileexplorer/files/usbTemp/secret.txt
retval: undefined
*** exiting $init
*** entered $init
```

Figure 14. SMB shared files, saved in the external storage

This means that a remote attacker would be able to read or write files to SMB shares of a local network, assuming that the device was connected to it. The same stands for FTP shares as they are handled exactly in the same way:



```
barbet:/storage/emulated/0/Android/data/com.mi.android.globalFileexplorer/files/usbTemp # ls -al
total 10
drwxrws--- 2 u0_a1152 ext_data_rw 3452 2023-06-16 13:19 .
drwxrws--- 3 u0_a1152 ext_data_rw 3452 2023-06-16 12:23 ..
-rw-rw---- 1 u0_a1152 ext_data_rw    0 2023-06-16 12:23 .nomedia
-rw-rw---- 1 u0_a1152 ext_data_rw    7 2023-06-16 13:19 secret.txt
barbet:/storage/emulated/0/Android/data/com.mi.android.globalFileexplorer/files/usbTemp # cat secret.txt
secret
barbet:/storage/emulated/0/Android/data/com.mi.android.globalFileexplorer/files/usbTemp #
```

Figure 15. FTP shared files, saved in the external storage

In summary, the exploitation flow is depicted in the figure below:

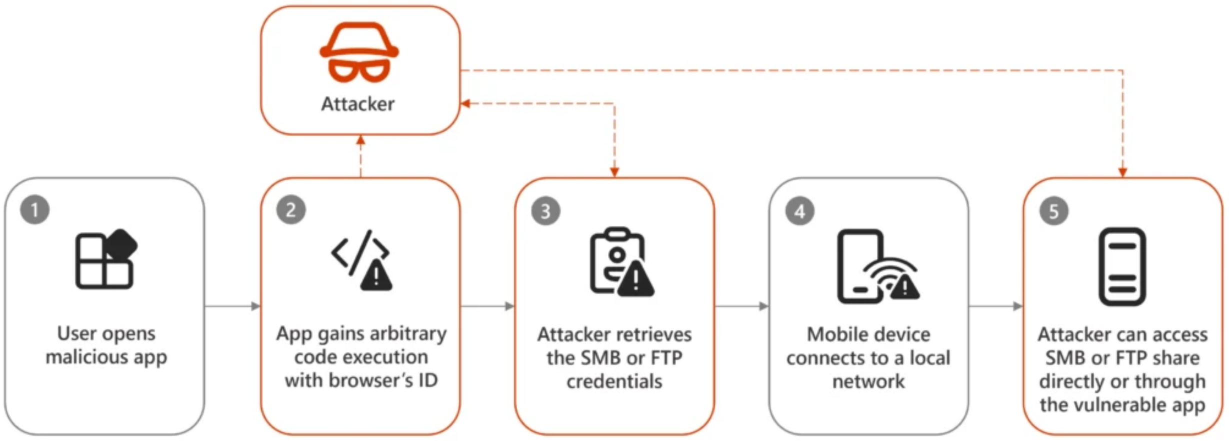

Attacker
1 User opens malicious app
2 App gains arbitrary code execution with browser's ID
3 Attacker retrieves the SMB or FTP credentials
4 Mobile device connects to a local network
5 Attacker can access SMB or FTP share directly or through the vulnerable app

Figure 16. Getting remote access to local shares

In **step 1**, the user opens a malicious app that may pose as a file editor, messaging app, mail client, or any app in general and request the user to save a file. By the time that the user attempts to save such a file, no matter what destination path they choose to save it, the malicious app forces the file browser app to write it under its internal */files/lib* folder. Then, the malicious app can start the junk cleaner using an explicit intent (no user interaction is required) and this will lead to code execution with the browser's ID (**step 2**).

In **step 3,** the attacker uses the arbitrary code execution capability to retrieve the SMB and FTP credentials from the *rmt_i.properties* file. Subsequently, the attacker can now jump to **step 5** and access the shares directly using the stolen credentials. Alternatively, after retrieving the share credentials, the mobile device can connect to a local network (**step 4**) and access an SMB or FTP share**,** allowing the attacker to access the shared files through the */sdcard/Android/data/com.mi.android.globalFileexplorer/files/usbTemp/* folder (**step 5**).

Recommendations

Recognizing that this vulnerability pattern may be widespread, we shared our findings with Google's Android Application Security Research team. We collaborated with Google to author guidance for Android application developers to help them recognize and avoid this pattern. We recommend developers and security analysts familiarize themselves with the excellent Android application security guidance provided by Google as well as make use of the Android Lint tool included with the Android SDK and integrated with Android Studio (supplemented with Google's additional security-focused checks) to identify and avoid potential vulnerabilities. GitHub's CodeQL also provides capabilities to identify vulnerabilities.

To prevent these issues, when handling file streams sent by other applications, the safest solution is to completely ignore the name returned by the remote file provider when caching the received content. Some of the most robust approaches we encountered use randomly generated names, so even in the case that the content of an incoming stream is malformed, it won't tamper with the application.

In cases where such an approach is not feasible, developers need to take extra steps to ascertain that the cached file is written to a dedicated directory. As an incoming file stream is usually identified by a content URI, the first step is to reliably identify and sanitize the corresponding filename. Besides filtering characters that may lead to a path traversal and before performing any write operation, developers must verify that the cached file is within the dedicated directory by performing a call to the *File.getCanonicalPath* and validating the prefix of the returned value.

Another area to safeguard is in the way developers try to extract a filename from a content URI. Developers often use *Uri.getLastPathSegment(),* which returns the (URL) *decoded* value of the last path URI segment. An attacker can craft a URI with URL encoded characters within this segment, including characters used for path traversal. Using the returned value to cache a file can again render the application vulnerable to this type of attack.

For end users, we recommend keeping mobile applications up to date through the Google Play Store (or other appropriate trusted source) to ensure that updates addressing known vulnerabilities are installed. Users should only install applications from trusted sources to avoid potentially malicious applications. We recommend users who accessed SMB or FTP shares through the Xiaomi app before updates to reset credentials and to investigate for any anomalous behavior. Microsoft Defender for Endpoint on Android can alert users and enterprises to malicious applications, and Microsoft Defender Vulnerability Management can identify installed applications with known vulnerabilities.

Dimitrios Valsamaras

Microsoft Threat Intelligence

References

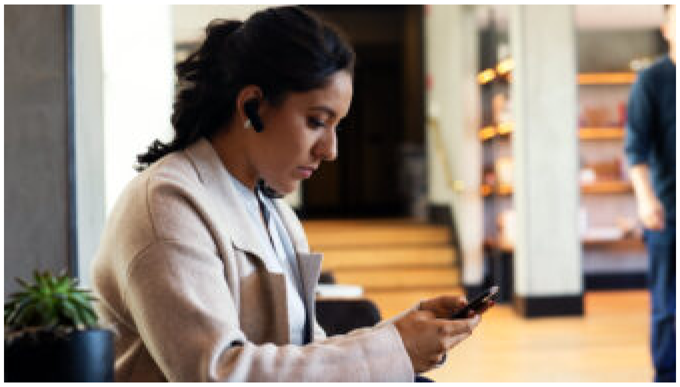
Learn more

For the latest security research from the Microsoft Threat Intelligence community, check out the Microsoft Threat Intelligence Blog: https://aka.ms/threatintelblog.

To get notified about new publications and to join discussions on social media, follow us on LinkedIn at https://www.linkedin.com/showcase/microsoft-threat-intelligence, and on X (formerly Twitter) at https://twitter.com/MsftSecIntel.

To hear stories and insights from the Microsoft Threat Intelligence community about the ever-evolving threat landscape, listen to the Microsoft Threat Intelligence podcast: https://thecyberwire.com/podcasts/microsoft-threat-intelligence.

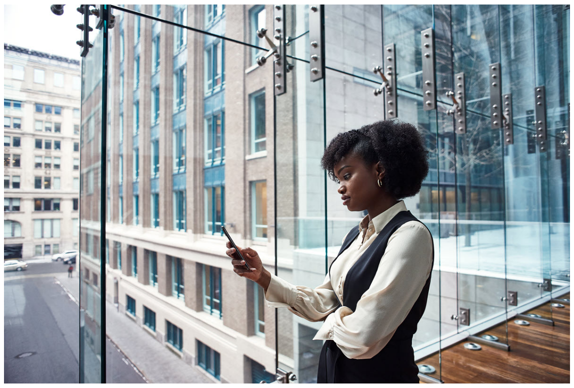
Related Posts



Research
Threat intelligence
Microsoft Defender
Mobile threats
Nov 20, 20239 min read

Social engineering attacks lure Indian users to install Android banking trojans

Microsoft has observed ongoing activity from mobile banking trojan campaigns targeting users in India with social media messages and malicious applications designed to impersonate legitimate organizations and steal users' information for financial fraud scams.



Protecting Android clipboard content from unintended exposure

Microsoft discovered that the SHEIN Android application periodically read the contents of the Android device clipboard and, if a particular pattern was present, sent the contents of the clipboard to a remote server.

DEV-0196: QuaDream's "KingsPawn" malware used to target civil society in Europe, North America, the Middle East, and Southeast Asia

Microsoft analyzes a threat group tracked as DEV-0196, the actor's iOS malware "KingsPawn", and their link to an Israel-based private sector offensive actor (PSOA) known as QuaDream, which reportedly sells a suite of exploits, malware, and infrastructure called REIGN, that's designed to exfiltrate data from mobile devices.

Vulnerability in TikTok Android app could lead to one-click account hijacking

Microsoft discovered a high-severity vulnerability in the TikTok Android application, now identified as CVE-2022-28799 and fixed by TikTok, which could have allowed attackers to compromise users' accounts with a single click.