Demystifying Android's Scoped Storage Defense

Yu-Tsung Lee | Pennsylvania State University
Haining Chen | Google
Trent Jaeger | Pennsylvania State University

Android recently introduced the scoped storage defense to better protect application use of shared external storage. This article examines the evolution of Android external storage defenses leading to scoped storage and assesses the impact of the scoped storage defense for limiting opportunities for exploitation.

Android has become the most dominant mobile operating system (OS) worldwide, deployed by a large number of vendors across a wide variety of form factors, including phones, tablets, and wearables.[14] As Android devices integrate into people's daily lives, the OS needs to provide sufficient and appropriate security methods to protect applications, services, and itself from compromise. One area of concern for Android is how to balance the ease with which applications can download and share content while protecting against attacks.

Many applications need to perform file downloads to retrieve data (e.g., media files) and perform software updates. Android, with a rich application platform common for smartphone systems, must support such downloads. In the early days, Android relied on external storage, such as removable secure digital (SD) cards, to store downloaded files to avoid exhausting limited on-device storage. With the increase in on-device storage, Android systems, maintaining the same application model, now devote a file system partition for such downloads, which is called the *external storage partition*. For simplicity, this article will refer to all access to files and directories in the external storage partition as *access to external storage*.

However, shared external storage presents problems with protecting access to downloaded files. In recent Android versions, applications can request that a user grant permission to read and write files in external storage. Since sharing is common for processing some types of files, such as media, requesting permission to access files in external storage has become common for applications, inuring users against the threat their authorization could pose. By granting a malicious third-party application full read or write access, a user allows the app to compromise the integrity of, or leak sensitive data from, any file in external storage. Multiple common vulnerabilities and exposures (CVEs) have been reported that exploit such permissions in recent Android systems (for example, see reports for CVE-2020-11990, CVE-2018-6599, CVE-2018-15004, and CVE-2019-12763). In addition, CheckPoint performed a study of this attack vector,[9] revealing that several popular applications were prone to attack, as described in the "Risks in Sharing Files" section. For example, an exploit was demonstrated that leverages this vulnerability against WhatsApp, leading to remote code execution on victim devices.[7] As a result, depending on users to manage access to external storage is problematic.

Android recently introduced a new defense to prevent attacks on shared external storage. In Android 11, Google fully deployed scoped storage,[6] an implementation of external storage that protects files by greatly reducing sharing and limiting the impact of user consents to modifications. (A limited version of scoped storage was introduced in Android 10, but this article focuses on the complete deployment in Android 11.) This article examines the impact that scoped storage

Digital Object Identifier 10.1109/MSEC.2021.3090564
Date of current version: 2 August 2021

 Copublished by the IEEE Computer and Reliability Societies 1540-7993/21©2021IEEE

has on external storage security. After describing prior efforts to protect external storage and their limitations, the article describes the semantics of scoped storage and estimates its effectiveness.

What Can Go Wrong in External Storage?

Some degree of sharing among applications is expected for many media files and documents. However, other files, such as software updates, are not typically expected to be shared. By downloading all these types of files to external storage, Android introduces risks that enable adversaries to gain access that may cause two kinds of security problems.

Sharing in External Storage

In the early days of Android, nearly every device relied on microSD cards for (external) storage, due to the fact that the phones shipped with limited internal storage. In newer devices, there is only one unified partition for applications to store data of all kinds: the */data* partition. However, applications still use external storage for many files, as described. External storage now refers to a virtual file system (i.e., partition), */data/media*, which is mounted on the */data* partition using a bind mount at */storage/emulated*. Figure 1 shows the structure of the file systems Android provides for application storage. The */data* partition provides app-internal storage under the */data/data* directory. External storage is provided in the */data/media* partition through a shared directory (*/storage/emulated*) in which any application can download a file and create a directory (given permission). Applications can then download files into their own directories. In addition, Android provides dedicated folders for shared downloads, videos, photos, and audio files. Any application can download a file into those folders.

The sharing of files among applications is common in external storage, where a user may produce a media file or a document with one application and apply another app to view (read) and modify it. Multiple apps from same developer commonly share data files through external storage. In addition, applications employ external storage to download other large files, such as software updates. Game apps are notable for large downloads (e.g., more than 100 MB). Originally, external storage was specifically provided to enable the downloading and storage of large files (e.g., via external SD cards). Although on-device storage is much greater, the use of external storage to download updates and store large media file continues.

Threat Model for External Storage

In general, the threat from shared external storage is that an untrusted app, in particular, an app from a third-party source, may attack a victim app. Thus, adversaries are third-party apps. Victims may be other third-party apps but also vendor apps (i.e., from original equipment manufacturers) and platform apps (i.e., from the Android distribution). Vendor and platform apps are often more privileged—Android assigns them signature-level permissions[4]—and they perform critical tasks. It is critical for Android to protect the integrity of privileged applications when they use external storage. An adversary may use its access to files and directories in external storage to launch attacks. First, a malicious party may try to exploit file sharing in external storage to leak sensitive data and to modify integrity-critical information. Second, it may leverage the ability to write to shared directories to create files and symbolic links that may lure victims. These attack vectors are examined in more detailed in the following.

Risks in Sharing Files

Using shared external storage for downloads may enable adversaries to maliciously modify another application's files. For example, Checkpoint performed a study[9] of an attack vector in Android external storage that is shown in Figure 2. The company tested whether malicious applications could replace another app's update files in external storage by swapping a malware file for a library. Researchers found that the attack could compromise popular applications, including Google Translate, Google Voice Typing, and Xiaomi Browser.

As shown in Figure 2, a victim app requests an update from its cloud server that is downloaded to external storage. (While the Google Play Store downloads updates to app-internal storage, it is common for Android applications to update from third-party sites.) An adversary can monitor external storage to determine when a victim application downloads an update. If the adversary

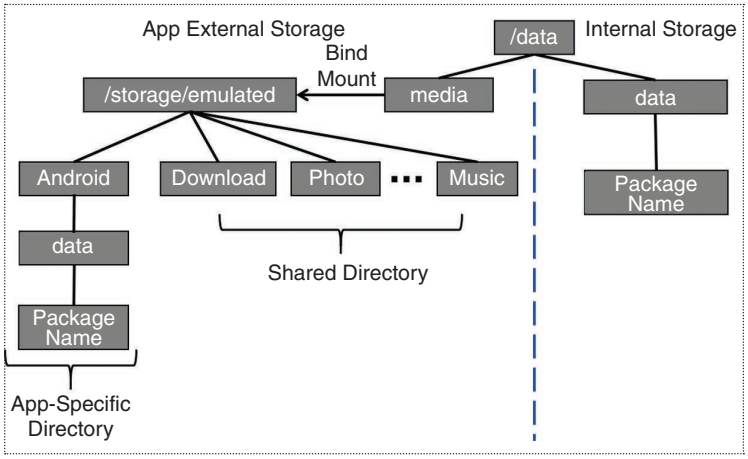

Figure 1. The application storage structure.

has sufficient permissions (e.g., Android permissions to read and write in external storage, as described in the "Prior Android Controls for File Sharing" section), it can replace that file with a malicious one that may be used by the victim app. Although applications could perform integrity checks through checksum and signature verification, leaving this responsibility to developers is not ideal. Furthermore, adversaries could leverage time-of-check-to-time-of-use attacks[1] to bypass signature validation. Finally, bad actors may use this attack vector to leak secrets by reading sensitive information in media files and documents.

Risks in Sharing Directories

Adversaries may abuse access to shared directories in external storage to trick applications into using certain files when performing downloads. Should an adversary be authorized to write to a shared directory, it can create files and symbolic links to lure victims to files it selected. Such attacks have long been recognized by researchers[11] but have been difficult to prevent because of the desire to allow multiple parties to share directories, as in external storage. In one type of attack, called *file squatting attacks*, an adversary predicts the name of a file that a victim will create and makes a file with that name in advance. If the victim does not validate that the file has already been created, it will use the malicious file instead, enabling a third party to read and modify data. Note that the victim must have or be granted permission to create the file (i.e., write access) to access the "squatted" version. Similarly, an adversary may create symbolic links in a shared directory to lure a victim to a file that enables access to unauthorized data, which is known as a *link traversal attack*. Fortunately, the use of symbolic links in external storage is prohibited due to a lack of support in the original file system types, as described in the "Prior Android Controls for Shared Directories" section.

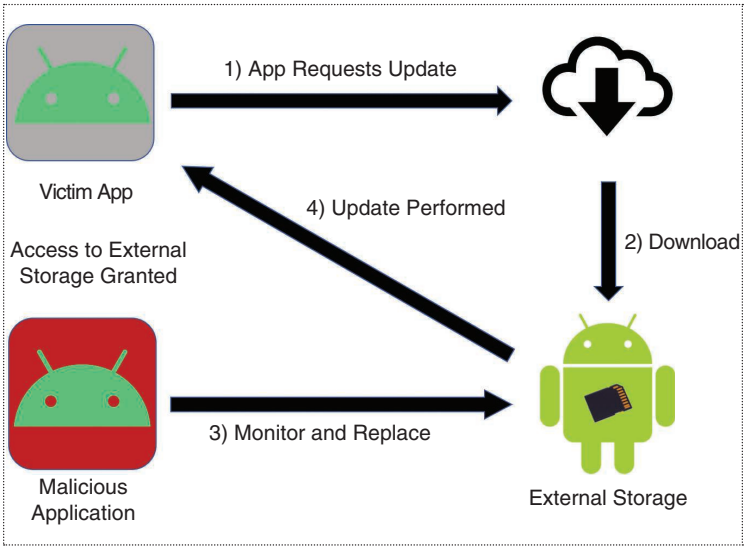

Figure 2. File swapping in external storage. Assuming that both victim and malicious applications are granted access to external storage, 1) a victim app requests an app update, 2) the update file is downloaded to external storage, 3) a malicious application monitors the external storage and replaces the victim's update file, and 4) the victim fails to verify the integrity of the file, and malicious modifications are applied instead of a normal update.

Prior Defenses for Android External Storage

Android has adopted multiple access control techniques to protect application and system files, such as UNIX discretionary access control (DAC), Security Enhanced Android (SE Android) mandatory access control (MAC), and the Android permission system. These have greatly improved the OS's integrity protection.[10] However, they provide only limited protection to shared folders and files in external storage, as illustrated in Figure 3 (and described in detail in the following). Prior Android versions configured access control to accommodate file sharing, which introduced risks leading to the vulnerabilities described previously.

Prior Android Controls for File Sharing

Before scoped storage, Android applied a combination of UNIX DAC and permissions to govern access to external storage. Although SE Android's MAC enforcement techniques have become a prominent part of the access control enforcement, the dynamic nature of sharing in external storage requires permission changes at runtime. As a result, SE Android grants all applications full access to external storage. Traditionally, DAC is used to control access to files in shared directories in UNIX systems. Android, as a UNIX-based OS, employs DAC but by leveraging permissions. Traditional UNIX DAC marks a file creator as the owner, which can configure and modify permissions to govern sharing (e.g., chmod). Android's DAC differs because instead of expecting apps to regulate permissions, the OS allows users to manage sharing. Android permissions are defined for users to grant apps read access or read–write access to all the directories and files in external storage.

Figure 3(a) shows how Android controlled access to external storage prior to scoped storage (prescoped storage). Without obtaining any permission, applications could access their app-specific directory and the files there, as in Figure 3(a). Two Android permissions, read external storage (REX) and write external storage (WEX), were defined to grant read and read–write access, respectively, to shared directories and all app-specific directories of other apps (the official names for these permissions are READ_EXTERNAL_STORAGE and

WRITE_EXTERNAL_STORAGE). User consent was required to obtain them, but they could be requested by and granted to any application. Prior to Android 6.0, apps had to request these permissions at installation time, but since that version, they can ask for them at runtime. Many media-related applications (e.g., photo and music apps) request permission to access WEX to share access to files. As a result, requesting permission to use access external storage has become very common, so most users do not understand how powerful their authorization is. Should malicious applications obtain write permission to external storage, they could replace another app's files, laying the groundwork for the file swapping attack in Figure 2.

Prior Android Controls for Shared Directories

Researchers have found that adversaries can exploit shared directories via file squatting and link traversal attacks, as described in the "Risks in Sharing Directories" section, if they have write access.[15] However, researchers have also found that preventing such attacks, in general, is not practical without imposing restrictions on application functionality.[2] Prior to scoped storage, Android depended on applications to protect themselves from file squatting attacks and benefited from limitations in the physical file systems for external storage to prevent link traversal attacks.

To prevent file squatting attacks, defenses must perform some action to block adversaries from creating files before victims do or keep victims from using squatted files. Prior to scoped storage, Android depended on applications to provide their own mechanisms for this. One approach was for applications to generate unpredictable names for the files they created. For example, applications could compute randomized names that were impractical for adversaries

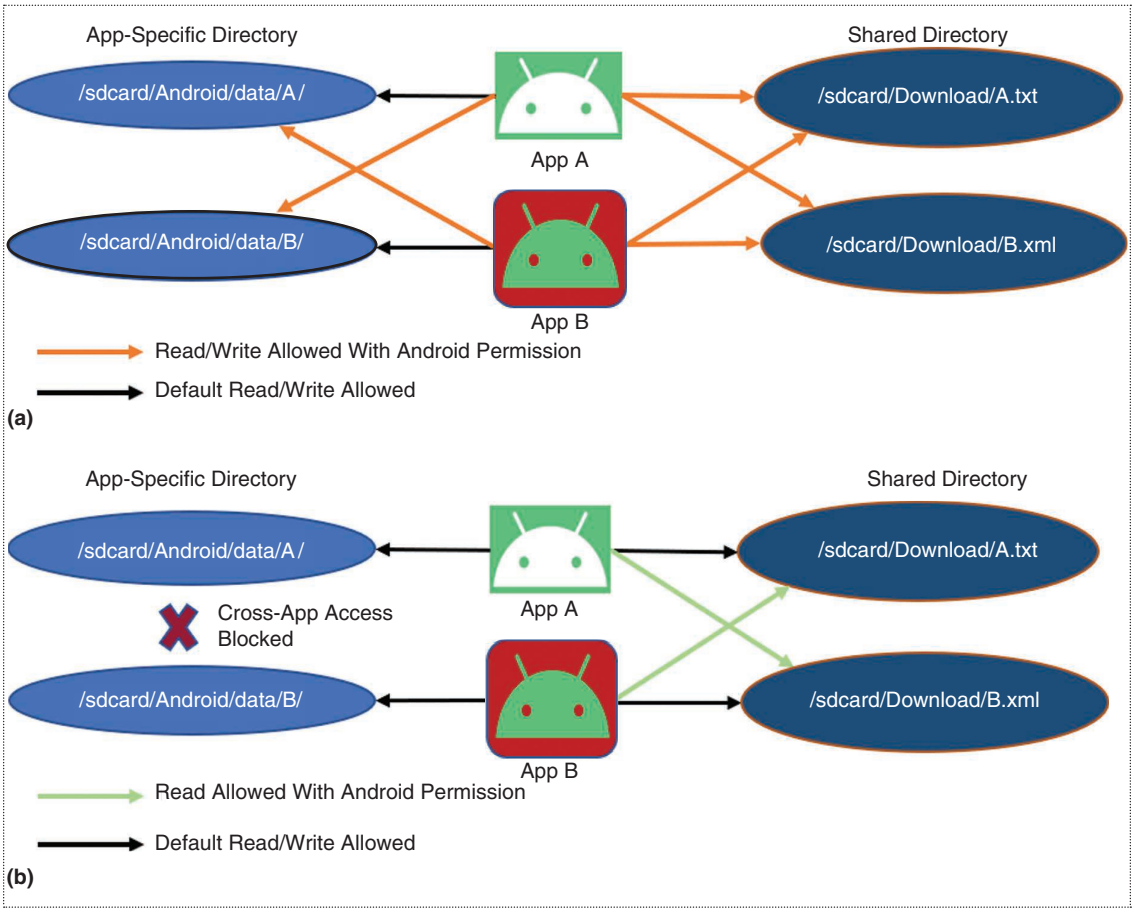

Figure 3. The access pattern authorized by prescoped storage (e.g., Android 9) versus scoped storage (Android 11). (a) Prescoped storage enables apps that obtain Android permissions to read and write files in shared and app-specific directories. Apps can read and write files in their own directories by default. Scoped storage blocks cross-app access to application-specific directories. In addition, scoped storage limits apps to read and write the files they created in shared directories, also by default. (b) In scoped storage, read access to shared directories is granted via Android permissions, as in prescoped storage, but write access may be obtained only via user consent for every file in shared directories.

to guess. However, not all applications would apply name randomization. It is nontrivial to request every developer to deploy such strategy. Furthermore, in the case of media, users prefer readable file names to ease searching the file manager.

On the other hand, Android has employed a simple defense to prevent link traversal attacks by prohibiting the creation of symbolic links in external storage. This has been aided by the fact that the types of file systems used for external storage do not support symbolic links. Originally, external storage was mounted as a virtual file allocation system; the File System in User Space (FUSE) and SD Card File System used later did not support symbolic links either. Fortunately, applications have no need for symbolic links for sharing in external storage. In general, it is now possible to mount a flag in a file system to prevent the use of symbolic links in name resolution (since Linux 5.10).

Understanding Scoped Storage

In Android 10, an experimental implementation of scoped storage[6] was introduced with the aim of providing improved protection to app and user data in external storage. Scoped storage involves significant restrictions of file sharing, as described in more detail in the following. To permit vendors and app developers to migrate to the new storage access pattern, it was still possible to request the legacy access model (by setting the requestLegacyExternalStorage attribute to *true* in the manifest file) described in the "Prior Defenses for Android External Storage" section. Scoped storage was fully implemented in and mandatory for Android 11. This article describes scoped storage in Android 11.

Scoped Storage Policy

The main purpose of scoped storage is to enable applications to download files that will never be shared with other apps, whose ability to modify shared files is restricted, as in Figure 3(b). Specifically, scoped storage makes two key changes: 1) application files downloaded into app-specific directories are treated as private files, and 2) application files downloaded into shared directories are treated as public files, which can be shared for reading via Android permissions and for writing with explicit user consent, with the exceptions of the system gallery and apps that are eligible for all-file access (all-file access requires apps to be vetted before they are published to the Google Play Store). As detailed in Figure 3(b), applications can access only their own files. Thus, app-specific directories are suitable for downloading software updates whose integrity must be protected and sensitive data files.

However, sharing media and documents is still required. To permit controlled sharing, Android provides common directories for public files. The directories are the same as those in prescoped storage, as shown in Figure 1, although permission management has been changed. By default, applications can read and write only their own files. However, as demonstrated in Figure 3(b), users may grant some applications REX permission, but only files in shared directories are readable. WEX permission is deprecated in scoped storage, so there is no authorization that provides blanket write privileges to applications for files in external storage. In scoped storage, applications have to request user consent to modify any file, one file at a time, and only for public files in shared directories.

To manage these permissions, scoped storage stores the relationship between files and their creators. Although this is similar to how UNIX DAC associates a file with its owner, DAC allows apps to modify permissions for any file they own arbitrarily (i.e., without user consent). Scoped storage prevents applications from ever accessing other apps' private files, providing mandatory protection beyond what UNIX DAC can offer. Even for public files, only users can grant permissions to other apps to access files in a shared directory, either via REX to allow read access or by providing explicit consent per file to permit read–write access [directory-level sharing through the storage access framework (SAF) is an exception, as discussed in the "SAF" section]. Unlike UNIX DAC, apps are not entrusted with managing file permissions.

Accessing Files Protected by Scoped Storage

Applications have three ways to access files in scoped storage: the file application programming interface (API), which invokes the Posix API; the MediaStore API; and the SAF API. The impact scoped storage has on file and MediaStore API access is described in Table 1. Using the SAF API is not directly governed by scoped storage, so that interface is separately considered in the following.

File and MediaStore APIs. As shown in Table 1, these APIs provided similar functionality through Android permissions to control access in prescoped storage, but scoped storage enforces access to limit sharing differently, particularly for writes. We include scoped storage in Android 10 and 11 in Table 1 to show the transition. Since apps do not require permission to access their private files, and others cannot access those files via these APIs, we consider only public files in Table 1. There are several key differences between prescoped and scoped storage for accessing public files. Most importantly, WEX permission is deprecated in scoped storage, as described in the "Scoped Storage Policy" section. Thus, typical third-party apps can no longer get users to grant them permission to write files owned by other

Table 1. The scoped storage access policy for public files.

	File API read		File API write		MediaStore API read		MediaStore API write	
Prescoped storage	Own file	REX 1	Own file	WEX	Own file	REX	Own file	WEX
	Other file	REX	Other file	WEX	Other file	REX	Other file	WEX
Android 10	Own file	Yes*	Own file	Yes	Own file	Yes	Own file	Yes
	Other file	REX, RLS	Other file	WEX, RLS	Other file	REX	Other file	User†
Android 11	Own file	Yes	Own file	Yes	Own file	Yes	Own file	Yes
	Other file	REX	Other file	AFA	Other file	REX	Other file	User, AFA

**Authorized without Android permission.*
†Denied with the exception of explicit user consent.
RLS: request legacy storage, AFA: all-file access (privileged).

applications in shared directories via the file API. Apps with all-file access can use the file API to write public files. To write a public file owned by another application in external storage, an app must use the new MediaStore API and obtain user consent, as illustrated in Figure 4. However, the MediaStore API has two restrictions: 1) it cannot be used for app-specific directories, and 2) it cannot be employed for nonmedia files (e.g., .txt and .pdf files).

SAF. Since Android 4.4, the SAF has provided access to files and directories that are explicitly selected by a user.[5] Apps can choose to share local and cloud files by implementing the DocumentsProvider, which handles requests from other apps. Users must choose the files to be shared. In prescoped storage, the SAF allows file- and directory-level sharing from external storage, which is not governed by Android permissions. To comply with the intent of scoped storage, limitations were placed external storage sharing through the SAF in Android 11. For example, apps can no longer request access to the download directory in external storage. In addition, app-specific directories in external storage are no longer visible through the SAF by default.[6] However, apps can still actively share files from app-specific directories by implementing their own custom DocumentsProvider. In addition, apps can share their own files in app-specific directories by using the FileProvider, through which they can define their own access control logic. The FileProvider has the ability to limit sharing to prevent some attacks.[3]

How Is Scoped Storage Implemented?

To support scoped storage, there are multiple implementation changes to external storage. The most significant is the adoption of a FUSE system for external storage to enforce the scoped storage policy semantics. As illustrated in Figure 5, FUSE has two main components: the in-kernel driver and the user space daemon. When an app tries to access external storage, the Linux kernel invokes the driver, which passes the file request to the daemon. The daemon retrieves the file owner from the MediaProvider service to perform a permission check. If access is allowed, the daemon performs the operation on the ext4 file system (e.g, */data/media*) and returns the result to the requesting application.

The process for permission enforcement differs based on whether a resource is in an app-specific directory or a shared directory and whether the resource is in primary

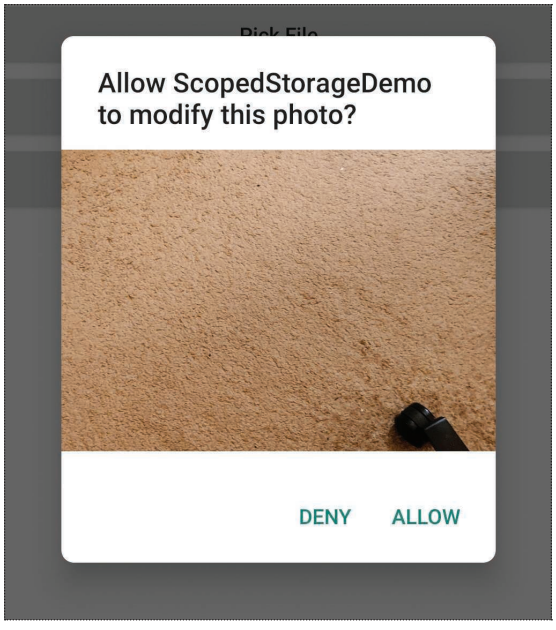

Figure 4. A request for user consent to allow another app (ScopedStorageDemo) to modify a photo.

or secondary external storage. To facilitate more efficient access to files in app-specific directories in primary external volumes, a bind mount is provided for individual services and apps, rather than a mount on the FUSE file system. The bind mount is protected by UNIX DAC permissions that strictly block access by other apps. For app-specific directories in secondary external storage, the FUSE daemon invokes a Java method in the MediaProvider to check by the package name whether the file operation request comes from the directory's owner. Authorization is granted only if the directory owner matches the requesting app's package name.

Permission enforcement for shared directories is more complicated and requires additional information to be stored. Whenever an app creates a new file in a shared directory in external storage, the MediaProvider inserts an entry in its database to store the owner and Multipurpose Internet Mail Extension (MIME) type. If the MIME type does not match the intended one for the directory (e.g., an audio file for a music folder), the create operation will be denied. Later, when an app wants to access its file, the MediaProvider simply checks whether the app created the file. However, if another application requests access to a file in a shared directory, it must have REX permission (for read only) or gain user consent (for read and write).

How Effective Is Scoped Storage?

To examine scoped storage's effectiveness, we wrote a "test app" that creates an image file in its app-specific directory and in a shared directory. (The image file is used to enable testing on the MediaStore API. The same results would apply to any other files created in an app-specific or shared directory.) We examined conditions under which a malicious app could access (read or write) the files in prescoped and scoped storage. We ran the test app on two systems on one Google Pixel 3a: 1) Android 9 (prescoped storage) and 2) Android 11 (scoped storage). The Pixel 3a implements scoped storage as described in this article. For each Android version, the device was flashed with stock firmware images to ensure a clean file system prior to running the test app. Then, we wrote a simple "attack app," which was assumed to be a third-party application that did not have all-file access. The test app could be a third-party app or a privileged app that used external storage. Privileged apps are deployed by Google and Android vendors to provide critical system functionality. There are many of them on modern Android devices, including 173 on our Pixel 3a. Therefore, protecting them is critical to system integrity.

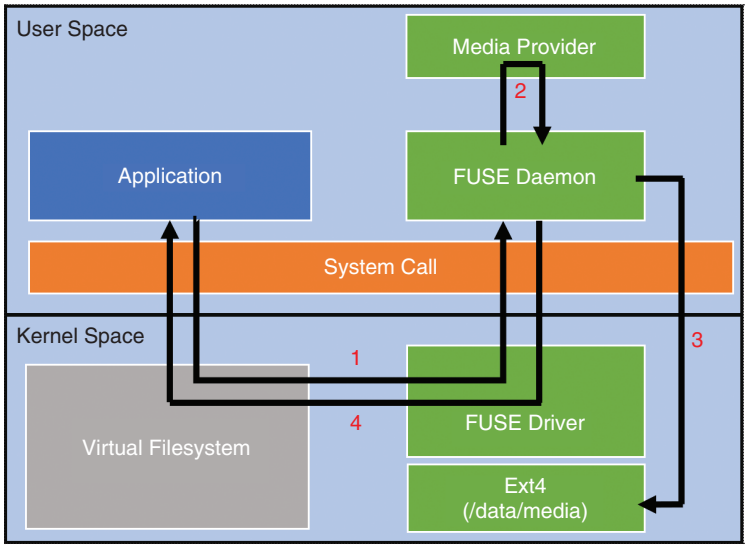

Figure 5. The FUSE implementation of scoped storage: 1) an app issues an operation on a file in the external storage partition, 2) FUSE performs access control using the MediaProvider, 3) the FUSE daemon performs the authorized operation on the ext4 file system, and 4) the operation result is returned to the app.

Protecting Apps From File Attacks

The first question is whether external storage files created by the test app are prone to attacks from a third-party app that may leak contents (i.e., secrecy attacks) or modify information (i.e., integrity attacks). Table 2 lists the operations the attack app was authorized to perform on any file in the test app's application-specific directory and on any file created by the test app in any shared directory for the two Android systems. For prescoped storage (Android 9), the attack app can read the files in both directories once it obtains REX permission and write (as well as read) files in both directories after it receives WEX permission. There is no further user consent required to gain permissions in prescoped storage systems.

On the other hand, under scoped storage, the app-specific folder is private, regardless of the permissions the attack app can obtain. (There is the exception of active sharing by an app for user-selected files through the SAF, which is not governed by scoped storage; see the "Accessing Files Protected by Scoped Storage" section.) Thus, the attack app cannot access files in the test app's application-specific directory, as shown in Table 2. With scoped storage, developers can safely store sensitive and integrity-critical data in app-specific folders. Scoped storage still permits read access to all files in shared directories once the attack app obtains REX permission, so this situation is unchanged for shared directories. However, WEX permission is deprecated, so the attack app cannot modify any files in shared storage when it has just one permission. Instead, it needs to obtain explicit user consent for each file created by another app to perform a write operation through the MediaStore API (for media files only) or the SAF.

Protecting Against File Squatting

The second question concerns the extent to which scoped storage protects the test app from file squatting attacks (see the "Risks in Sharing Directories" section). Recall that to complete a file squatting attack, 1) an adversary (e.g., the attack app) must be able to create a file of the same name as the one to be created by the victim (e.g., the test app), and 2) the victim must have the permission to create (i.e., modify) the file created by the adversary. (Recall from the "Risks in Sharing Directories" section that although a victim may open a squatted file that has only read permissions, it expects to be able to create the file, which requires write privileges.) Table 3 presents the permissions required for the attack app to create files that may be used in file squatting attacks and those required for the test app to write the squatted file created by the attack. Thus, the conditions under which a file squatting attack is possible are indicated in Table 3. *App specific* in Table 3 refers to any files in the test app's application-specific directory. The attack app is assumed to be able to predict the names of the files created by the test app in this experiment.

For prescoped storage, the attack app needs WEX permission to write to shared directories and app-specific directories to conduct file squatting attacks. Once the attack app has that permission, it can launch file squatting attacks in the app-specific and shared directories, as shown in Table 3. For an attack to succeed, the test app needs to be authorized to access the squatted file. The test app can access any file in its app-specific directory without permissions, but it needs to be granted WEX permission to access squatted files in shared directories. Thus, apps with greater privilege are more prone to file squatting attacks in prescoped storage, as demonstrated in results from Android policy studies.[8]

For scoped storage, the attack app is denied the ability to write to other app's application-specific directories under any circumstances, but it is granted permission to write to shared directories by default (i.e., without requiring the deprecated WEX permission). However, as Table 3 shows, it is more difficult for the test app to obtain write permissions to squatted files, as the app must request and be granted user consent. Since the test app would be unlikely to request user consent to write a file that it expects to create (i.e., the test app would obtain write permission by virtue of creating the file), it is unlikely to request user consent.

Are There Issues to Consider for Scoped Storage?

Although scoped storage improves app security, it does present some concerns that require consideration.

User Consent May Allow Exploits

Under scoped storage, an app may modify a file in a shared external storage folder that is owned by another application, but a user must explicitly consent. In Android 10, users need to issue consent for every file. This could create a large number of requests when an app requests write access to multiple files (i.e., modifying the brightness for multiple photos requires numerous consents). In Android 11, the MediaStore API allows users to consent to write access for multiple media files at once. However, a potential problem is that users may not carefully check each file in bulk operation requests, which could result in authorizing malicious writes. Ideally, scoped storage will balance clarity in user consent requests with functional needs. Researchers have explored this problem for runtime authorization of Android permissions in general.[12,13] Another problem is that a user may grant all-file access to a malicious app. Although most users

Table 2. Attack app permissions to target test app files.

	Prescoped storage		Scoped storage	
	App specific	Shared	App specific	Shared
No Android permissions	—	—	—	—
REX (secrecy)	R	R	—	R
WEX (integrity)	RW	RW	N/A†	N/A†
User consent (integrity)	N/A*	N/A*	—	RW

R: read only; RW: read–write.
**User consent is not used in prescoped storage.*
†WEX permission is deprecated in scoped storage.

Table 3. The necessary permissions for file squatting.

	Prescoped storage		Scoped storage	
	App specific	Shared	App specific	Shared
No Android permissions	—	—	—	C
REX	—	—	—	C
WEX	CW	CW	N/A†	N/A†
User consent	N/A*	N/A*	—	CW

C: an attack app can create a file; CW: a victim app can write a squatted file (i.e., a file squatting attack is possible).
**User consent is not used in prescoped storage.*
†WEX permission is deprecated in scoped storage.

obtain applications through the Google Play Store, which carefully vets apps before allowing all-file access, some people install apps from third-party sites. Therefore, there is still a concern that malicious apps may obtain all-file access. We do not know how this situation will be resolved.

Threats to Secrecy Remain

Although broad write access to external storage has been removed, significant read access to shared directories in external storage still exists. Although Google recommends that developers store private data in app-specific directories, developers might make mistakes. For example, performance logs that might include sensitive hardware information could be stored in shared directories. In this case, another app may leverage the coarse-grained REX permission to leak sensitive data. In the future, scoped storage may need to be extended to reduce the threats presented by programmer mistakes in reading data produced by other apps.

Will Apps Operate Correctly Under Scoped Storage?

Application developers are used to having complete access to external storage, as this has been allowed since the beginning of Android. Therefore, requesting that developers adapt to the new access model may introduce problems. In fact, the original plan to require fully enforced scoped storage in Android 11 has already been relaxed, and the legacy external storage model has been restored for apps targeting API levels below 29.[6] In addition, compatibility issues for apps that require all-file access to function (e.g., file management apps)[6] have yet to be fully resolved. Currently, these apps cannot obtain this permission because Google has paused app vetting.

Android has introduced the scoped storage defense to improve file system integrity in external storage. In prior systems, Android applications were prone to attacks on files in shared directories in external storage, causing several vulnerabilities. Scoped storage provides applications with true app-specific directories that are inaccessible to other applications and removes permissions that grant broad write access to shared files to limit the ability of applications to write objects owned by others. An evaluation of the file protections offered by prescoped and scoped storage shows that scoped storage reduces the opportunities adversaries have to attack files in external storage. Thus, scoped storage appears to improve file system integrity, but issues remain, such as the potential for excessive user consent and the need for existing applications to be modified to operate effectively. ■

Acknowledgments

This research was sponsored by the Combat Capabilities Development Command Army Research Laboratory (ARL) under cooperative agreement W911 NF-13-2-0045 (ARL Cyber Security Collaborative Research Alliance), National Science Foundation grant CNS-1801534, and Google's 2020 Android Security and Privacy Research Award. The views and conclusions in this article are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory, the U.S. government, or Google. The U.S. government is authorized to reproduce and distribute reprints for government purposes, not withstanding any copyright notation here on.

References

1. M. Bishop and M. Digler, "Checking for race conditions in file accesses," *Comput. Syst.*, vol. 9, no. 2, pp. 131–152, Spring 1996.
2. X. Cai, Y. Gui, and R. Johnson, "Exploiting Unix file-system races via algorithmic complexity attacks," in *Proc. 30th IEEE Symp. Security Privacy*, 2009, pp. 27–41.
3. "FileProvider URL." Google. https://developer.android.com/reference/android/support/v4/content/FileProvider (accessed May 2021)
4. "Permissions overview URL." Google. https://developer.android.com/guide/topics/permissions/overview (accessed May 2021)
5. "Storage access framework URL." Google. https://developer.android.com/guide/topics/providers/document-provider (accessed May 2021)
6. "Storage updates in Android 11 URL." Google. https://developer.android.com/preview/privacy/storage (accessed May 2021)
7. C. Karamitas, "Remote exploitation of a man-in-the-disk vulnerability in WhatsApp (CVE-2021-24027)," Census Lab, Athens, Apr. 2021. [Online]. Available: https://census-labs.com/news/2021/04/14/whatsapp-mitd-remote-exploitation-CVE-2021-24027/
8. Y.-T. Lee et al., "PolyScope: Multi-policy access control analysis to triage android systems," in *Proc. 30th USENIX Security Symp.*, to be published.
9. S. Makkaveev, "Man-in-the-disk: Android apps exposed via external storage," Check Point Research, Aug. 2018. https://research.checkpoint.com/2018/androids-man-in-thedisk/ (accessed Apr. 2021).
10. R. Mayrhofer, J. Vander Stoep, C. Brubaker, and N. Kralevich, "The Android platform security model," 2019. [Online]. Available: https://arxiv.org/abs/1904.05572

11. W. S. McPhee, "Operating system integrity in OS/VS2," *IBM System J.*, vol. 13, pp. 230–252, no. 3, Sept. 1974. doi: 10.1147/sj.133.0230.
12. G. Petracca, A.-A. Reineh, Y. Sun, J. Grossklags, and T. Jaeger. "Aware: Reventing abuse of privacy-sensitive sensors via operation bindings," in *Proc. 26th USENIX Security Symp.*, Aug. 2017.
13. T. Ringer, D. Grossman, and F. Roesner, "Audacious: User-driven access control with unmodified operating systems," in *Proc. ACM Conf. Comput. Commun. Security (ACM CCS)*, 2016, pp. 204–216.
14. "OS Market Share," StatCounter. Mar. 2020. https://gs.statcounter.com/os-market-share (accessed Apr. 2021).
15. H. Vijayakumar, J. Schiffman, and T. Jaeger, "STING: Finding name resolution vulnerabilities in programs," in *Proc. 21st USENIX Security Symp.*, 2012, pp. 585–599.

Yu-Tsung Lee is a Ph.D. student at Pennsylvania State University, State College, Pennsylvania, 16801, USA. His research interests include system and software security, particularly mobile platform security and access control. Lee received a B.E. from the University of Michigan, Ann Arbor. Contact him at yxl74@psu.edu.

Haining Chen is a software engineer in Android security at Google, Mountain View, California, 94043, USA. Her research interests include mobile security, in general, and particularly device authentication, biometrics security, and access control. Chen received a Ph.D. from the Department of Computer Science, Purdue University, West Lafayette, Indiana. Contact her at hainingc@google.com.

Trent Jaeger is a professor in the Department of Computer Science and Engineering, Pennsylvania State University, State College, Pennsylvania, 16801, USA. His research interests include systems and software security, particularly for operating systems. Jaeger received a Ph.D. from the University of Michigan, Ann Arbor. He has served as chair of the Association for Computing Machinery (ACM) Special Interest Group on Security, Audit, and Control and as the Steering Committee chair for the Internet Society Network and Distributed Systems Symposium. He is an associate editor in chief of *IEEE Security & Privacy* and an editorial board member of *Communications of the ACM*. Contact him at trj1@pu.edu.

Digital Object Identifier 10.1109/MSEC.2021.3103619