Toward Efficient File Vulnerability Testing with Policy Analysis and Program Analysis on Android

Yu-Tsung(Eddy) Lee

Ph.D. Candidate @ Penn State U.

Feb. 29, 20224

The Hidden Threat: File Vulnerabilities in Android

- Deceptive - Exploiting Legitimate Functionality

  Unlike malware, which often behaves suspiciously, file vulnerabilities often exploit legitimate app components

- Silent Leakage - No Immediate Disruptions

  Secrecy related attacks often aims to steal private user data without interruption to service

- Leverage Existing Trust - Abusing Access Control Permissions

  Attackers can exploit permissions granted by users or malicious third-party libraries can abuse permissions granted to trusted applications

Types of File-Related Vulnerabilities in Android

Adversary-controlled Resource Access

File Squatting

App or system process tries to access a file with a specific name, access adversary-controlled file instead of genuine one

Symbolic-Link Attack

App or system process tries to access a file with a specific name, but is provided with an adversary controlled symbolic link.

Adversary-controlled Pathname Access

Confused Deputy

App or system process uses adversary-controlled filepath without sanitization. Leads to unauthorized read/write of victim data

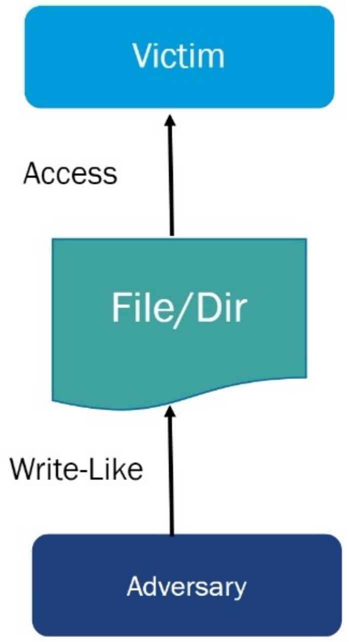
Detecting File Vulnerability

What are the causes of file attack?

- Access control allows victims to access malicious resources
- Access control allows attacker to modify content/binding of resources accessed by victim
- Victim uses malicious resources without proper sanitization

To detect file vulnerabilities...

- **Access control policy analysis** to identify concrete attack surface for processes on host device
- **Program analysis** to verify conditions needed for victims to access risky resources is satisfiable



Threat Model on Android

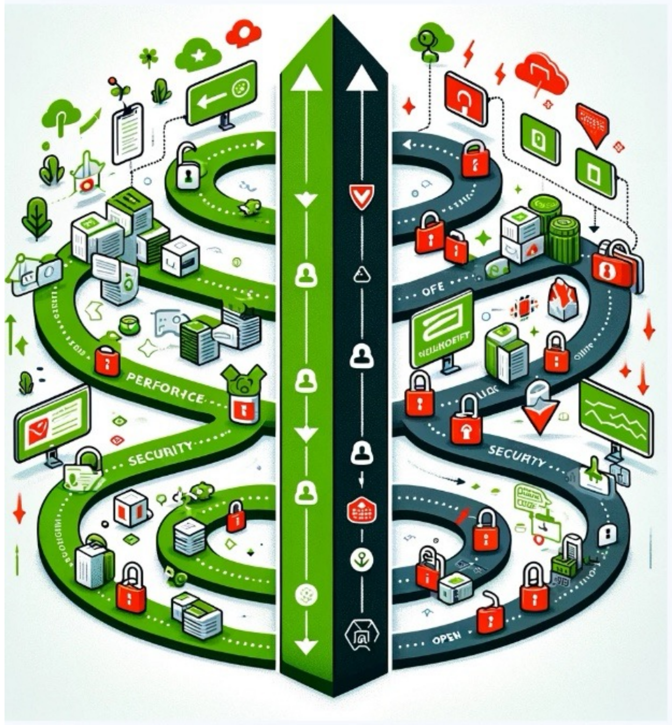
Executing untrusted code on host system isn't ideal...

But, that's the default business model of Android (third-party applications)

Foundation of Android File Security

Access Control

Balancing File Sharing and Security



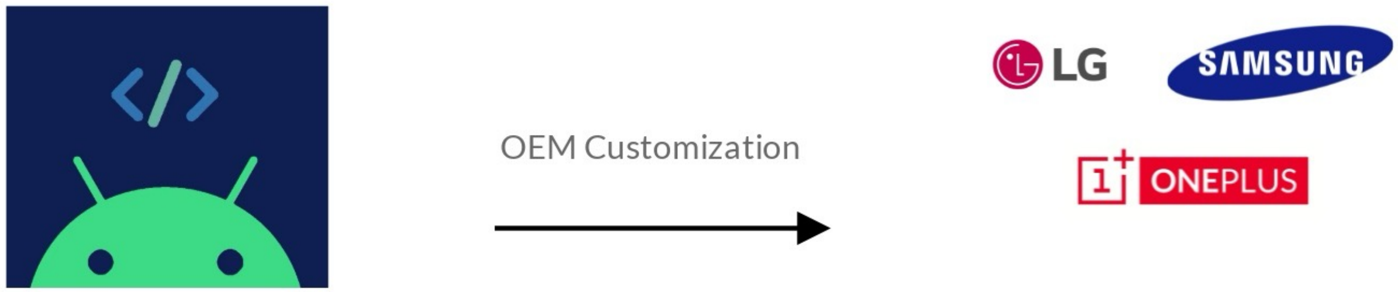
File Sharing

- Sharing media content between social apps
- Document sharing between productivity apps
- File/Data sharing between apps from the same developer

Security

- Sandboxing through traditional access control (MAC, DAC)
- Fine-grained access control through mechanism like Scoped Storage

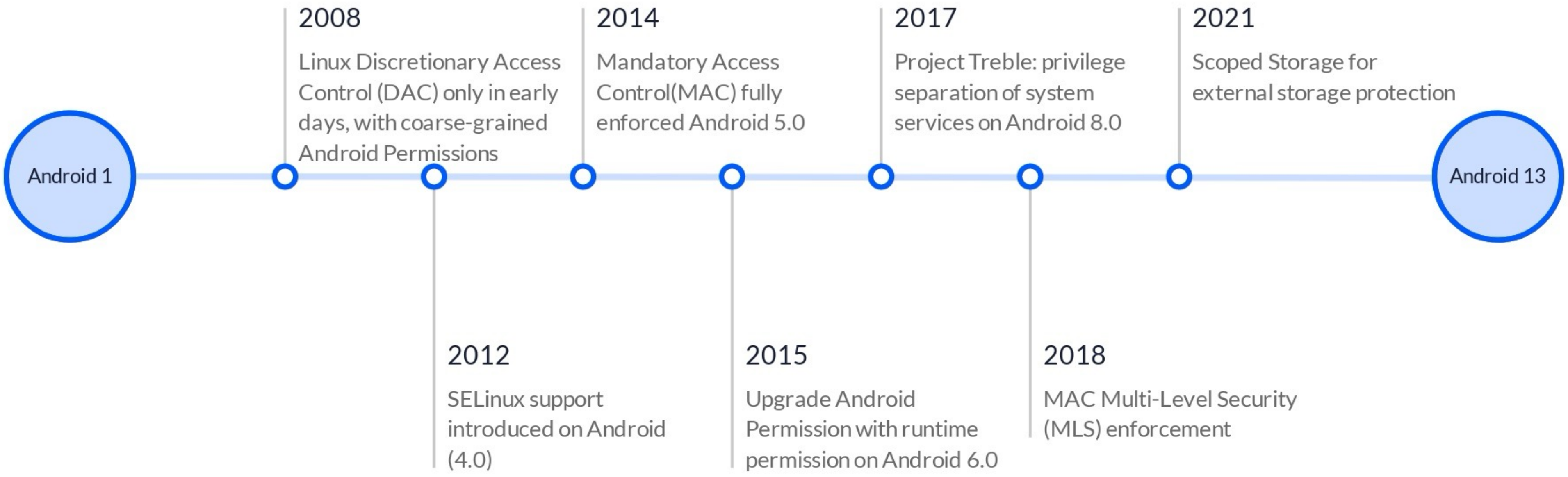
Problem with Android Access Control Policies



- Very complex access control model
- Rapid update and improvements made by Google

- Extensive access control policy customization
- Vendor service pre-installed to improve value
- Slower to adapt to new security practives

Evolution of Android Access Control



Android 1
2008
Linux Discretionary Access Control (DAC) only in early days, with coarse-grained Android Permissions
2012
SELinux support introduced on Android (4.0)
2014
Mandatory Access Control(MAC) fully enforced Android 5.0
2015
Upgrade Android Permission with runtime permission on Android 6.0
2017
Project Treble: privilege separation of system services on Android 8.0
2018
MAC Multi-Level Security (MLS) enforcement
2021
Scoped Storage for external storage protection
Android 13

Complexity of Android Access Control Policy

Multiple defense mechanism

- Mandatory Access Control (MAC)
- Discretionary Access Control (DAC)
- Android Permission
- Multi-Layer Security (MLS)
- Scoped Storage

10k+

ACCESS CONTROL POLICIES RULES

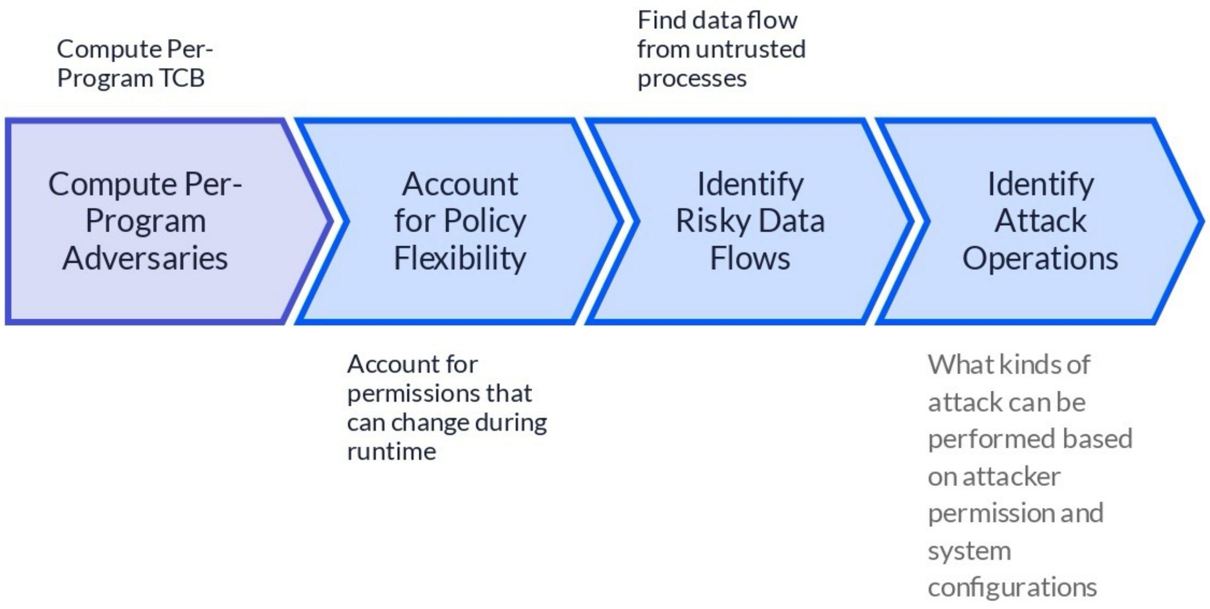
150k+

DATA FLOW ALLOWED BY ACCESS CONTROL

So many rules...
So many policies...
So many data flow...

Impractical to test all data flows!

Triaging Data Flow with PolyScope


Compute Per-Program TCB
Find data flow from untrusted processes
Compute Per-Program Adversaries
Account for Policy Flexibility
Identify Risky Data Flows
Identify Attack Operations
Account for permissions that can change during runtime
What kinds of attack can be performed based on attacker permission and system configurations

150k+

Data Flows Allowed by Access Control

Triage Down to

~1k

Attack Operations

Yu-Tsung Lee, William Enck, Haining Chen, Hayawardh Vijayakumar, Ninghui Li, Daimeng Wang, Zhiyun Qian, Giuseppe Petracca, and Trent Jaeger. Polyscope: Multi-policy access control analysis to triage android systems. USENIX Security 2021

Key Insight of PolyScope

Define Per-Program Adversary

Systematically compute TCB for each process running on the system

Account for Policy Flexibility

First work to include possible permission expansion into analysis

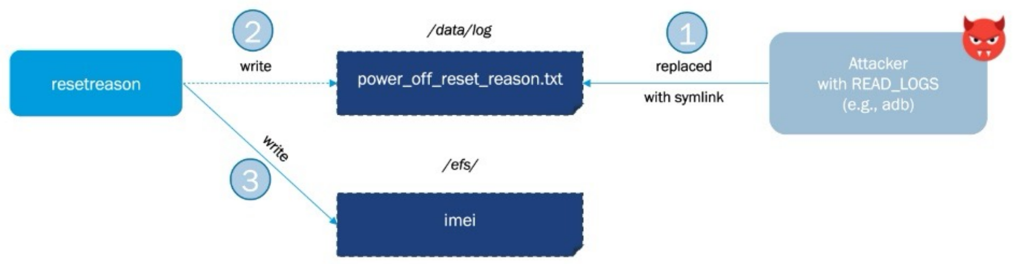
Account for System Configurations

Some parts of the filesystem are read only / does not allow symbolic links

Zero-Day Discovered

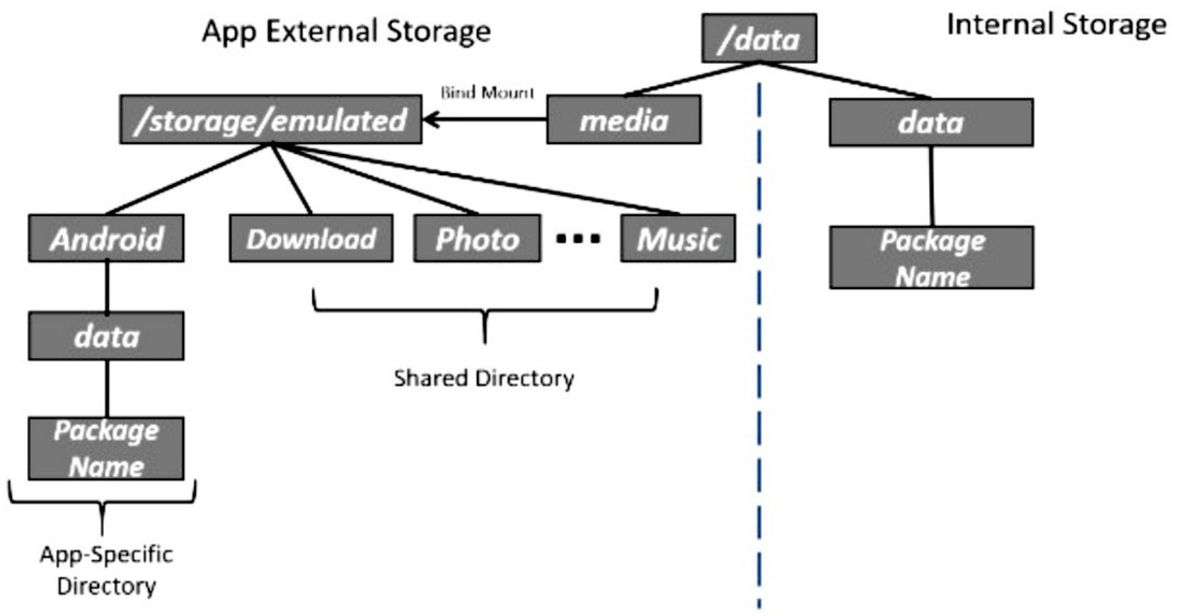
CVE-2020-13833 - Arbitrary Write through Symlink Attack

1. Attacker gain DAC GID log through Android Permission (adv-expansion)
2. Replaced power_off_reset_reason.txt with symlink to encrypted file system
3. resetreason becomes victim of confused deputy attack when writing reboot logs
4. Could corrupt file in /efs and brick the device



Adapting PolyScope to Scoped Storage

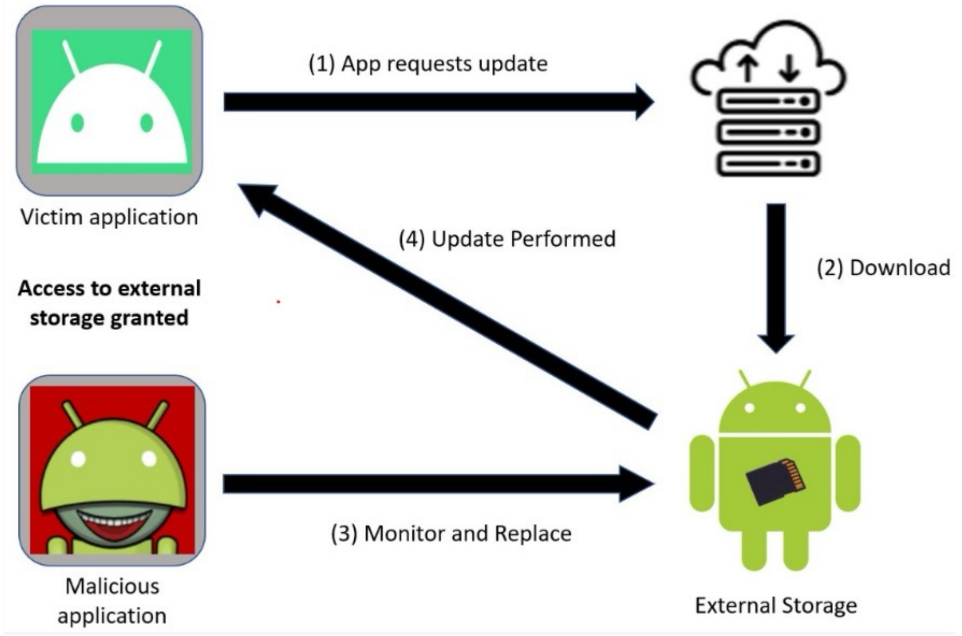
The most chaotic and problematic place on Android



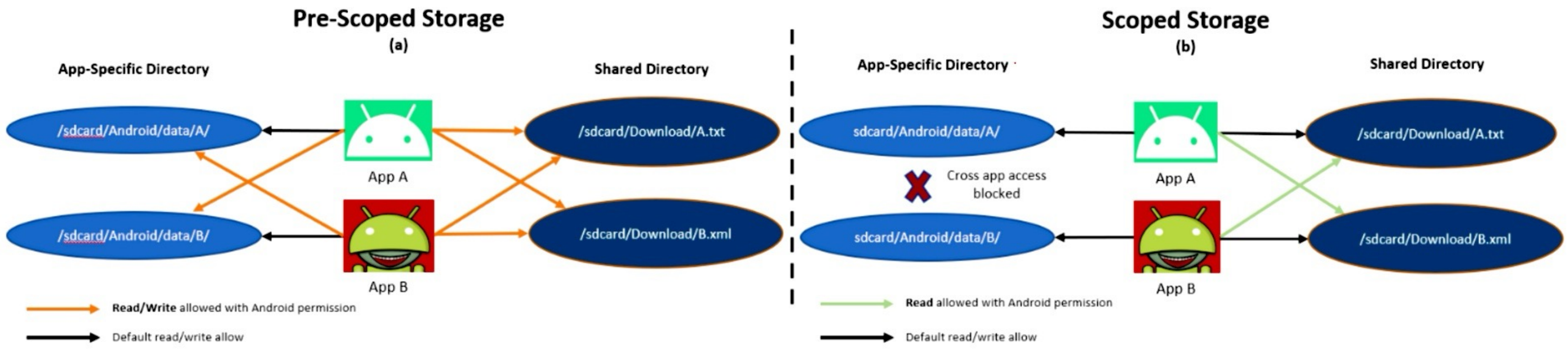
Example Attack on External Storage

Assume both attacker and victim are granted access to external storage

1. A victim app requests an app update
2. Update file is downloaded to external storage
3. A malicious application monitors external storage and replaces the victim's update file
4. Victim fails to verify the integrity of the file and malicious modifications are applied instead of normal update

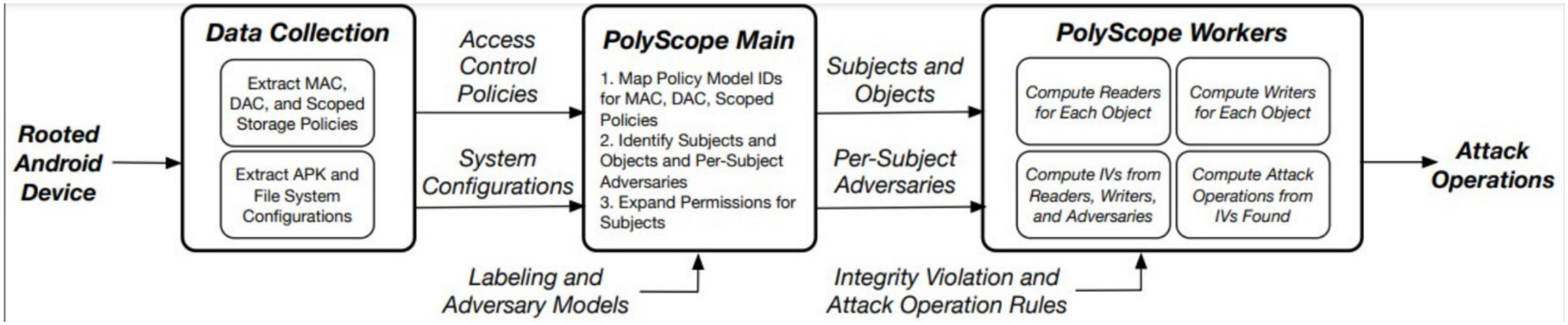


Introducing Scoped Storage



- Implemented by using filesystem in userspace (FUSE)
- File ownership information stored by Android service MediaProvider
- Provides per-app protection in external storage
- Significantly different from other kernel based access control mechanisms (e.g., MAC, DAC). Non-trivial effort to adapt to Scoped Storage.

Final Implementation


Rooted Android Device
Data Collection
Extract MAC, DAC, and Scoped Storage Policies
Extract APK and File System Configurations
Access Control Policies
System Configurations
PolyScope Main
1. Map Policy Model IDs for MAC, DAC, Scoped Policies
2. Identify Subjects and Objects and Per-Subject Adversaries
3. Expand Permissions for Subjects
Labeling and Adversary Models
Subjects and Objects
Per-Subject Adversaries
PolyScope Workers
Compute Readers for Each Object
Compute Writers for Each Object
Compute IVs from Readers, Writers, and Adversaries
Compute Attack Operations from IVs Found
Integrity Violation and Attack Operation Rules
Attack Operations

Input: Access Control Policy, application manifest files, DB files from MediaProvider, xml files from PackageManager

Output: Integrity Violations and Attack Operations

- Significantly improved subjects mapping rate by parsing permission related xml files
- Significant performance improvement by load balancing and multi-threading

Triaging Scoped Storage with PolyScope

By extending PolyScope for Scoped Storage, we demonstrate PolyScope can reason about newly added access control mechanisms independently . We then evaluate effectiveness of Scoped Storage quantitatively

Evaluation Summary

- Scoped Storage reduced attack operations in external storage up to **54%**
- Legacy applications are over-privileged, able to modify twice the number of resources compared to scoped storage compliant apps. We also discovered a file squatting vulnerability on one legacy app on OnePlus device (Android 12)
- Legacy applications can potentially attack twice as many subjects as scoped apps on average
- By emulating fully-enforced Scoped Storage, fully-enforced Scoped Storage should further reduce attack operations in external storage by 12%-28%

PolyScope Project Summary

- Compute accurate attack surface for each process on host Android device
- Can be extended to adapt to new access control mechanism like Scoped Storage
- Helped discover multiple vulnerabilities across different OEMs
- PolyScope + Attack Graph technique used by Army Research Lab to triage potential file attacks on military Android devices (proven to find vulnerabilities)
- Useful for OEMs to verify that there isn't any access control policy misconfiguration before release

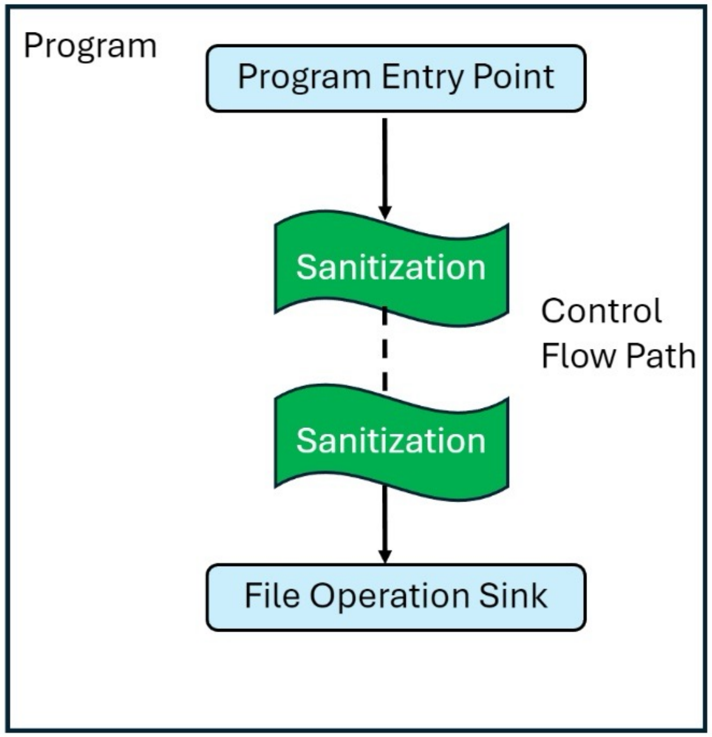
Toward Automatic Test Generation through Program Analysis

Reason Programs through Program Analysis

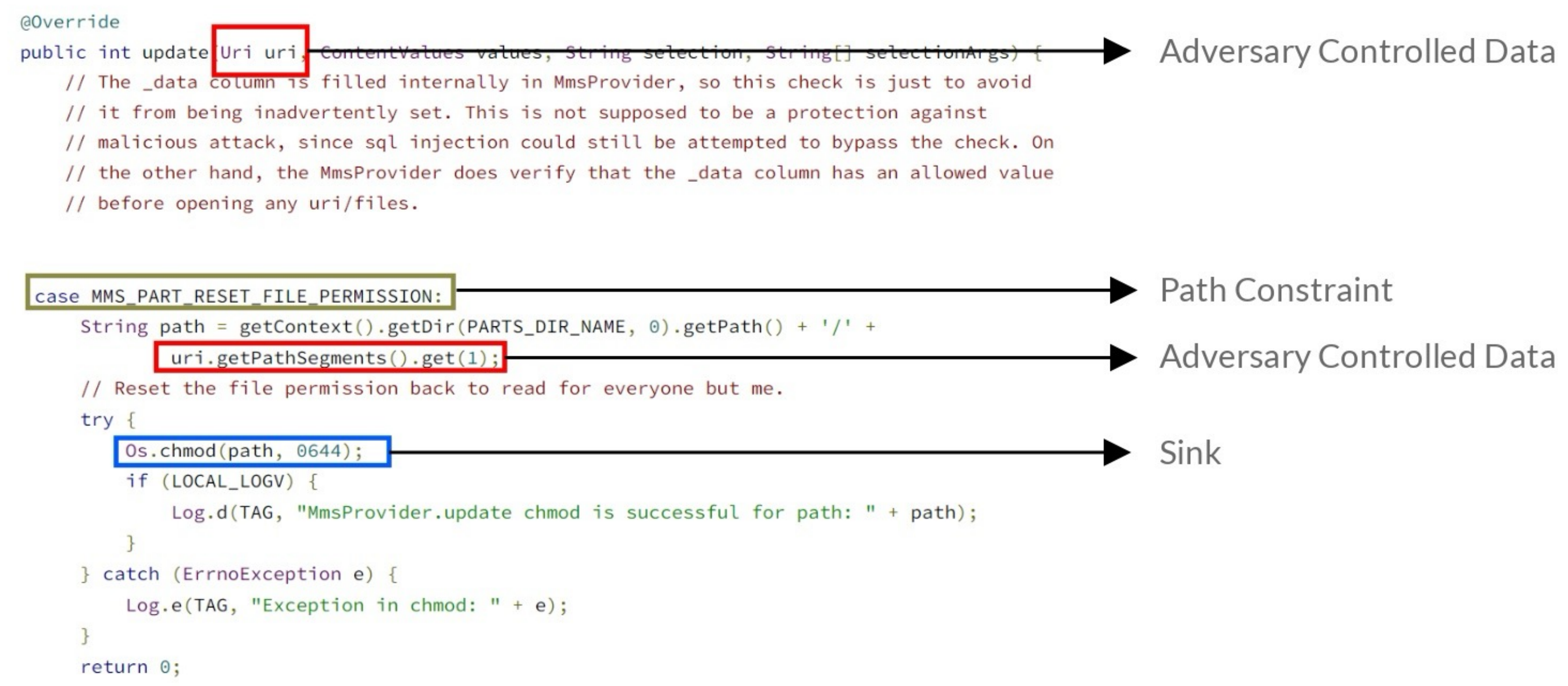
A program is threatened by filesystem attacks when:

- Performing file operation (e.g., read/write/open)
- Resources used controlled by adversary
- Pathname used controlled by adversary
- Lack of sanitization/filtering

Identified above criteria through program analysis

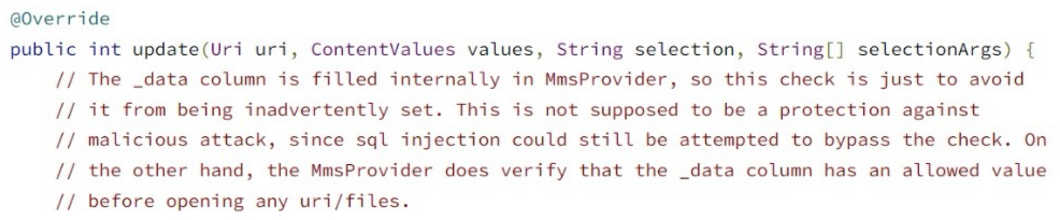


A Closer Look

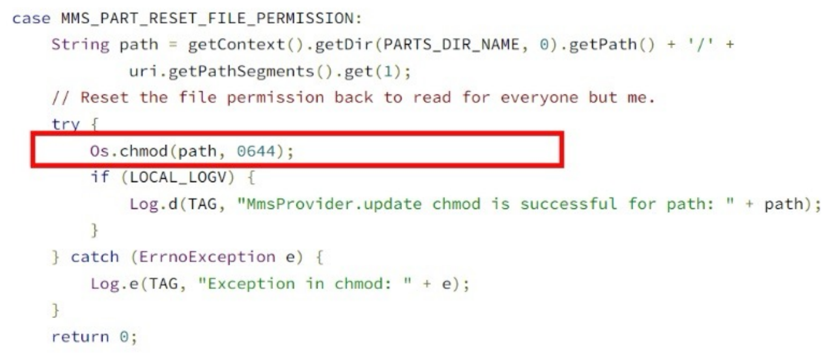

```
@Override
public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {
    // The _data column is filled internally in MmsProvider, so this check is just to avoid
    // it from being inadvertently set. This is not supposed to be a protection against
    // malicious attack, since sql injection could still be attempted to bypass the check. On
    // the other hand, the MmsProvider does verify that the _data column has an allowed value
    // before opening any uri/files.

 case MMS_PART_RESET_FILE_PERMISSION:
     String path = getContext().getDir(PARTS_DIR_NAME, 0).getPath() + '/' +
             uri.getPathSegments().get(1);
     // Reset the file permission back to read for everyone but me.
     try {
         Os.chmod(path, 0644);
         if (LOCAL_LOGV) {
             Log.d(TAG, "MmsProvider.update chmod is successful for path: " + path);
         }
     } catch (ErrnoException e) {
         Log.e(TAG, "Exception in chmod: " + e);
     }
     return 0;
```

Example CVE-2023-21268

MmsProvider DoS attack in Telephony - Chmod of adversary controlled filepath leads to DoS attack



```java
@Override
public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {
    // The _data column is filled internally in MmsProvider, so this check is just to avoid
    // it from being inadvertently set. This is not supposed to be a protection against
    // malicious attack, since sql injection could still be attempted to bypass the check. On
    // the other hand, the MmsProvider does verify that the _data column has an allowed value
    // before opening any uri/files.
```

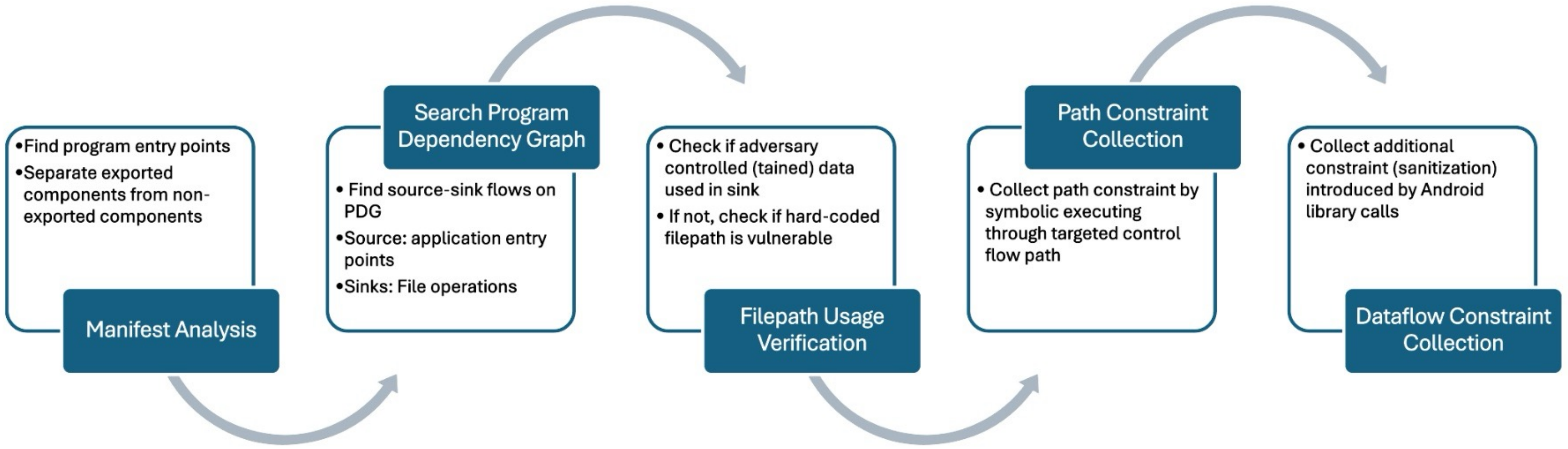


```java
case MMS_PART_RESET_FILE_PERMISSION:
    String path = getContext().getDir(PARTS_DIR_NAME, 0).getPath() + '/' +
            uri.getPathSegments().get(1);
    // Reset the file permission back to read for everyone but me.
    try {
        Os.chmod(path, 0644);
        if (LOCAL_LOGV) {
            Log.d(TAG, "MmsProvider.update chmod is successful for path: " + path);
        }
    } catch (ErrnoException e) {
        Log.e(TAG, "Exception in chmod: " + e);
    }
    return 0;
```

How do we detect this vulnerability through program analysis?

- We need to identify control flow path from program entry point to chmod
  - A: Construct CFG and search for path from source to sink
- We need to confirm that adversary controlled data reached chmod sink
  - A: Construct data flow graph and perform taint analysis
- How do we know what input we need to drive program down targeted execution path?
  - A: Constraint collection through symbolic execution + constraint solving

Program Analysis Approach


Manifest Analysis
•Find program entry points
•Separate exported components from non-exported components
Search Program Dependency Graph
• Find source-sink flows on PDG
•Source: application entry points
•Sinks: File operations
Filepath Usage Verification
• Check if adversary controlled (tained) data used in sink
• If not, check if hard-coded filepath is vulnerable
Path Constraint Collection
• Collect path constraint by symbolic executing through targeted control flow path
Dataflow Constraint Collection
• Collect additional constraint (sanitization) introduced by Android library calls

3-Types of Targeted Path (TP)

Inherent TP

- No path constraint (natural path of execution)
- No adversary controlled input that reach file operation sink

Attack Condition

- Use vulnerable filepath (identified by PolyScope)

Example

- Resetreason vulnerability shown previously

Contrained TP

- Adversary need to provide correct input to reach the sink
- No adversary controlled input that reach file operation sink

Attack Condition

- Use vulnerable filepath (identified by PolyScope)
- Satisfiable path constraint (determine using SMT solver)

Adversarial TP

- Adversary need to provide correct input to reach the sink
- Adversary controlled data reached file operation sink

Attack Condition

- Adversary controlled data reached sink
- Satisfiable path constraint
- No sanitization

Example

- MmsProvider vulnerability shown previously

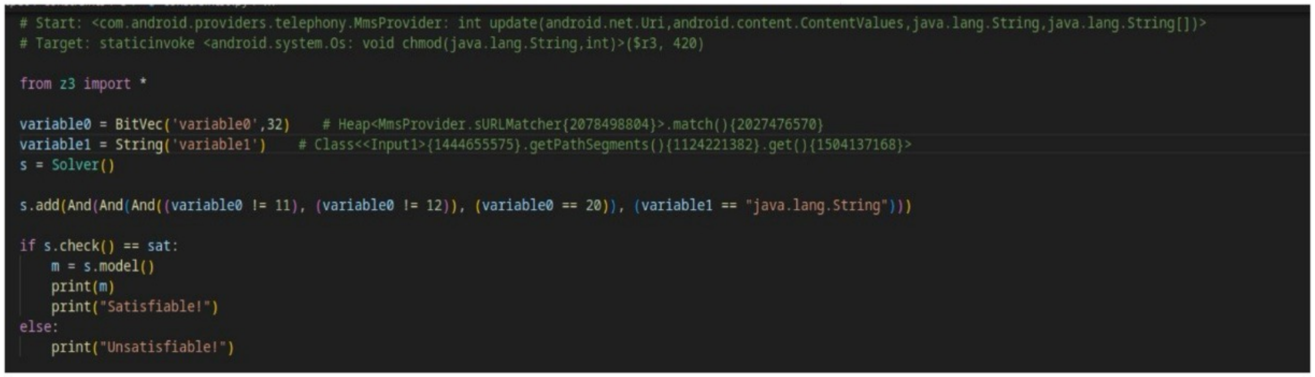
Driving Program through Targeted Path

To direct execution down targeted path, we leverage path constraint collected and Z3 solver

```
case MMS_PART_RESET_FILE_PERMISSION:
    String path = getContext().getDir(PARTS_DIR_NAME, 0).getPath() + '/' +
            uri.getPathSegments().get(1);
    // Reset the file permission back to read for everyone but me.
    try {
        Os.chmod(path, 0644);
        if (LOCAL_LOGV) {
            Log.d(TAG, "MmsProvider.update chmod is successful for path: " + path);
        }
    } catch (ErrnoException e) {
        Log.e(TAG, "Exception in chmod: " + e);
    }
    return 0;
```

Path Constraint

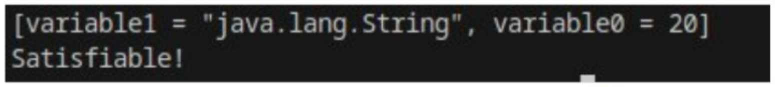
Generate python constraint file



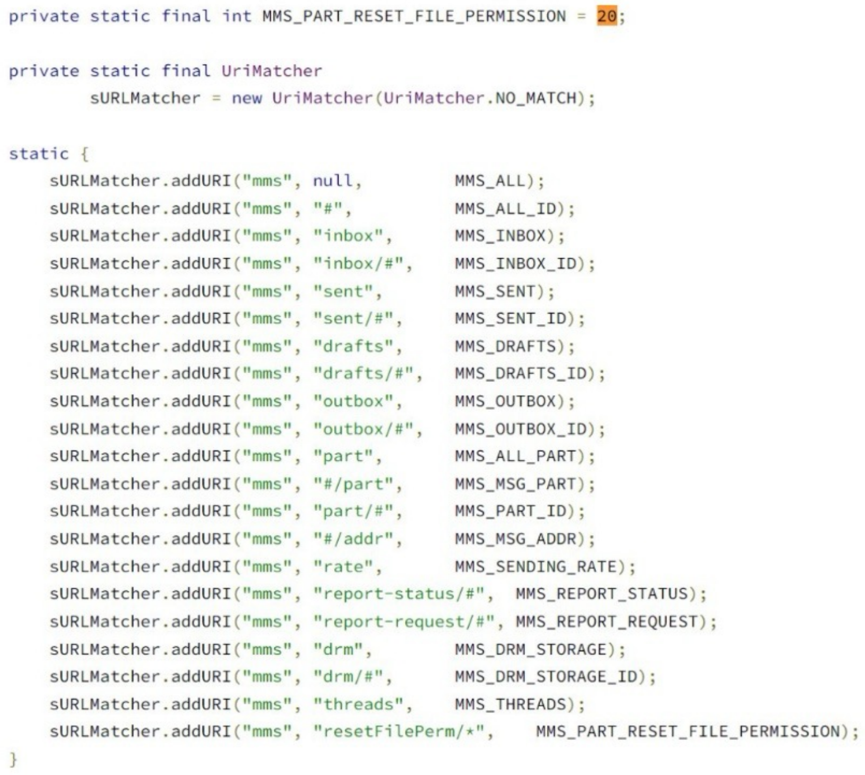
```
# Start: <com.android.providers.telephony.MmsProvider: int update(android.net.Uri,android.content.ContentValues,java.lang.String,java.lang.String[])>
# Target: staticinvoke <android.system.Os: void chmod(java.lang.String,int)>($r3, 420)

from z3 import *

variable0 = BitVec('variable0',32)    # Heap<MmsProvider.sURLMatcher{2078498804}>.match(){2027476570}
variable1 = String('variable1')    # Class<<Input1>{1444655575}.getPathSegments(){1124221382}.get(){1504137168}>
s = Solver()

s.add(And(And(And((variable0 != 11), (variable0 != 12)), (variable0 == 20)), (variable1 == "java.lang.String")))

if s.check() == sat:
    m = s.model()
    print(m)
    print("Satisfiable!")
else:
    print("Unsatisfiable!")
```

Some Manual Effort...

Z3 Solver Result



```
[variable1 = "java.lang.String", variable0 = 20]
Satisfiable!
```



```
private static final int MMS_PART_RESET_FILE_PERMISSION = 20;

private static final UriMatcher
        sURLMatcher = new UriMatcher(UriMatcher.NO_MATCH);

static {
    sURLMatcher.addURI("mms", null,         MMS_ALL);
    sURLMatcher.addURI("mms", "#",          MMS_ALL_ID);
    sURLMatcher.addURI("mms", "inbox",      MMS_INBOX);
    sURLMatcher.addURI("mms", "inbox/#",    MMS_INBOX_ID);
    sURLMatcher.addURI("mms", "sent",       MMS_SENT);
    sURLMatcher.addURI("mms", "sent/#",     MMS_SENT_ID);
    sURLMatcher.addURI("mms", "drafts",     MMS_DRAFTS);
    sURLMatcher.addURI("mms", "drafts/#",   MMS_DRAFTS_ID);
    sURLMatcher.addURI("mms", "outbox",     MMS_OUTBOX);
    sURLMatcher.addURI("mms", "outbox/#",   MMS_OUTBOX_ID);
    sURLMatcher.addURI("mms", "part",       MMS_ALL_PART);
    sURLMatcher.addURI("mms", "#/part",     MMS_MSG_PART);
    sURLMatcher.addURI("mms", "part/#",     MMS_PART_ID);
    sURLMatcher.addURI("mms", "#/addr",     MMS_MSG_ADDR);
    sURLMatcher.addURI("mms", "rate",       MMS_SENDING_RATE);
    sURLMatcher.addURI("mms", "report-status/#",  MMS_REPORT_STATUS);
    sURLMatcher.addURI("mms", "report-request/#", MMS_REPORT_REQUEST);
    sURLMatcher.addURI("mms", "drm",        MMS_DRM_STORAGE);
    sURLMatcher.addURI("mms", "drm/#",      MMS_DRM_STORAGE_ID);
    sURLMatcher.addURI("mms", "threads",    MMS_THREADS);
    sURLMatcher.addURI("mms", "resetFilePerm/*",    MMS_PART_RESET_FILE_PERMISSION);
}
```

Malicious URI to query:

content://mms/resetFilePerm/..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2Fdata%2Fuser_de%2F0%2Fcom.android.providers.telephony%2Fdatabases

Taint Analysis:

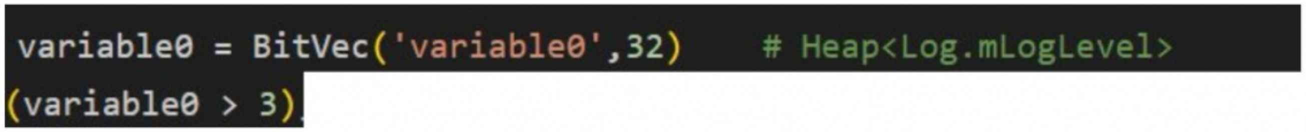
Confirms tainted source uri.getPathSegments() reaches sink os.chmod()

Result:

Chmod /data/user_de/com.android.providers.telephony/databases

Constrained TP found in ThemeCenter (S20, Android 12)

- ThemeManagerService onDestory() method write to /data/log/ThemeCenter.log without sanitization
- PolyScope identifies /data/log as risky directory attackable by processes with log DAC group (e.g., adb)
- Path constraint collected indicates:



```
variable0 = BitVec('variable0',32)    # Heap<Log.mLogLevel>
(variable0 > 3)
```

- Log.mLoglevel can be set through dialer special code "*#9900#", if debug level is set to high, the log write operation could be vulnerable to symbolic link attack (arbitrary write)

Thanks!
Questions?