CENSUS
IT Security Works

KernelGP: Racing against the Android kernel

Chariton Karamitas
huku@census-labs.com

May 19, 2025

Table of contents

Table of contents

Whoami

- huku a.k.a. Chariton Karamitas
  - Maybe lurking around using other nicks, maybe not
- Vulnerability researcher at CENSUS
  - Android only since 2015
- Binary diffing PhD
- Proud Phrack author
- Usual suspect at a couple of private events

Table of contents

Introduction

Problem statement

- During exploit development, the need to block a kernel thread for a (semi-)controlled amount of time might arise
  - Exploitation of vulnerabilities arising from invalid use of synchronization primitives (i.e., insufficient locking, mismatched lock/unlock APIs etc.)
    - Race against resource lock/unlock
  - Extension of memory allocation lifetime (e.g., for heap spraying)
    - Race against memory allocation/deallocation
  - Explicit ordering of kernel threads
    - Race against check/use (TOCTOU)

- Examples:
  - Race against lock/unlock (or lack thereof...)

    In Racing against the clock, Jann Horn uses `userfaultfd` to make sure an SKB has been freed before being used.
  - Race against memory allocation/deallocation

    Vitaly Nikolenko's Linux Kernel universal heap spray also uses `userfaultfd`.
  - Race against check/use (TOCTOU)

    In DirtyCred kernel blocking primitives are used to explicitly order permission checks and actual data writes (`userfaultfd`, FUSE, and file locks are proposed).

- The `userfaultfd` story:
  - On Linux desktops, exploit developers often abuse `userfaultfd` in order to force the kernel to block at `copy_from_user()`
    - Often abused...? → `vm.unprivileged_userfaultfd`
  - Until recently, this wasn't possible on Android
    - `CONFIG_USERFAULTFD` was not set as it was not needed
    - Vendors proactively disabled it to stop attackers from using it
  - Android 13 introduced the `userfaultfd`-based ART garbage collector
    - Depends on `CONFIG_USERFAULTFD`
    - Will it be there on Android 16?

Introduction

Problem statement

- The FUSE story:
  - On desktops, FUSE is a powerful alternative to `userfaultfd`
    - /dev/fuse is 0666
    - `fusermount` is SUID root
  - On Android, this isn't the case
    - /dev/fuse is root:root 0600
    - No SUID binaries and shit
    - Also SELinux
  - FUSE can't be used on Android
    - Oh really?

Introduction

Problem statement

- We need a toolset of techniques for forcing the Linux kernel to block a-la `userfaultfd`
- Ideally techniques which are hard to mitigate due to architectural design choices
- We will have a look at 4 novel techniques based on Android specific features:
  - 3 can be used from `untrusted_app` and are based on FUSE
  - 1 can be used from `system_server` and `system_app` and is based on IncFS (a.k.a. Incremental File System)

- ...so FUSE is actually available on Android. How come?
  - Scoped storage
  - Proxy file descriptors
- Basically, even though `/dev/fuse` is not accessible by `untrusted_apps`, the Android architecture makes it possible for the latter to use it
  - Allows for using FUSE for exploitation purposes as usual, at the cost of additional complications

Table of contents

Previous work

Basic framework

- Scheduling tricks (CPU affinity, priority etc.) [1]
- Forced preemption via interrupts (e.g., high-resolution timers, reschedule IPIs, memory barriers, hardware IRQs etc.) [1], [2]
- Resource monopolization (e.g., DRAM, PCI) [1], [2]
- Performance degradation (i.e., introduction of delays)

Previous work

Performance degradation (i.e., introduction of delays)

- Caching and page-faults
  - Cache eviction/admission
  - Reads from slow filesystems (e.g., network filesystems, FUSE, multiple device-mapper layers etc.)
  - `userfaultfd`
  - Custom FUSE daemon
- File locks
  - Concurrent operations
  - Advisory locking via `fcntl()`, `flock()`, etc.
  - Mandatory locking; `CONFIG_MANDATORY_FILE_LOCKING` and `mount -o mand`, although have never seen that in practice
- Blocking I/O
  - Reads from empty FIFOs (`pipe()`, `mkfifo()`, `socketpair()`, TTY etc.)
  - Writes to full FIFOs

Previous work

Performance degradation (i.e., introduction of delays)

- Caching and page-faults
  - Cache eviction/admission
  - Reads from slow filesystems (e.g., network filesystems, FUSE, multiple device-mapper layers etc.)
  - `userfaultfd`
  - Custom FUSE daemon
  - Proxy file descriptors
  - IncFS
- File locks
  - Concurrent operations
  - Advisory locking via `fcntl()`, `flock()`, etc.
  - Mandatory locking; `CONFIG_MANDATORY_FILE_LOCKING` and `mount -o mand`, although have never seen that in practice
- Blocking I/O
  - Reads from empty FIFOs (`pipe()`, `mkfifo()`, `socketpair()`, TTY etc.)
  - Writes to full FIFOs
  - Redaction
  - Transcoding

Table of contents

Proxy file descriptors

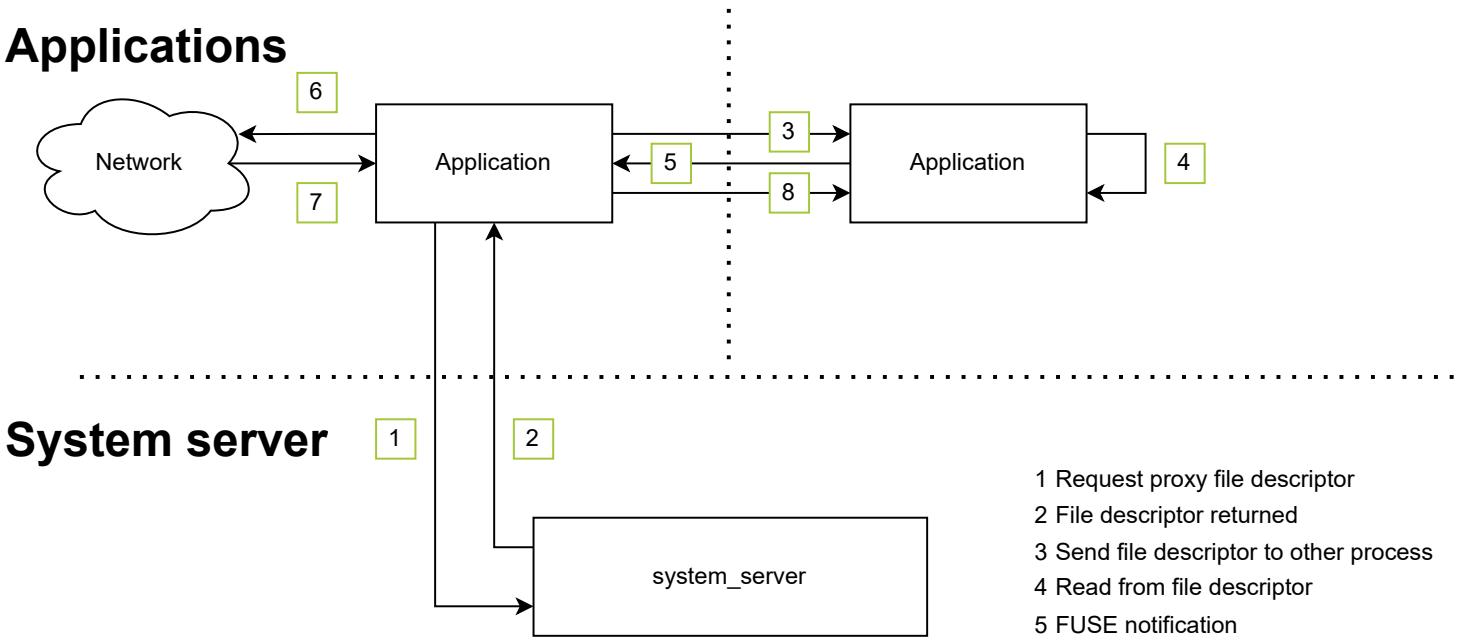
About

- Proxy file descriptors
  - Android abstraction over FUSE
  - System-calls issued on the descriptor are translated to Java callbacks that applications can implement
  - Handed over to applications by the `StorageManagerService`

Proxy file descriptors

About

Figure: Example usage of proxy file descriptor for a hypothetical network filesystem.



Proxy file descriptors

Intended use

- Use case: `RevocableFileDescriptor`
  - `system_server` opens a sensitive file and needs to be able to control the lifetime of the underlying file descriptor `mInner`
  - `system_server` creates a proxy file descriptor `mOuter`, that proxies access to `mInner`
  - `system_server` sends `mOuter` to the untrusted recipient
  - `system_server` can close `mInner` at any time to revoke access to the underlying file
- Used by `BlobStoreManagerService` and others

Proxy file descriptors

Revocable file descriptors

- Quick look at `RevocableFileDescriptor`

```
public class RevocableFileDescriptor {
    private FileDescriptor mInner;
    private ParcelFileDescriptor mOuter;
    private volatile boolean mRevoked;

    public RevocableFileDescriptor(Context context, File file)
            throws IOException {
        init(context, Os.open(file.getAbsolutePath(),
            OsConstants.O_CREAT  OsConstants.O_RDWR, 0700));
        ...
    }

    public void init(Context context, FileDescriptor fd, Handler handler)
            throws IOException {
        mInner = fd;
        StorageManager sm = context.getSystemService(StorageManager.class);
        ...

        mOuter = sm.openProxyFileDescriptor(ParcelFileDescriptor.MODE_READ_WRITE,
            mCallback);
    }
    ...
}
```

Proxy file descriptors

Revocable file descriptors

- Quick look at `RevocableFileDescriptor`

```
public class RevocableFileDescriptor {
    ...

    public void revoke() {
        mRevoked = true;
        IoUtils.closeQuietly(mInner);
    }

    public boolean isRevoked() {
        return mRevoked;
    }
    ...
}
```

Proxy file descriptors

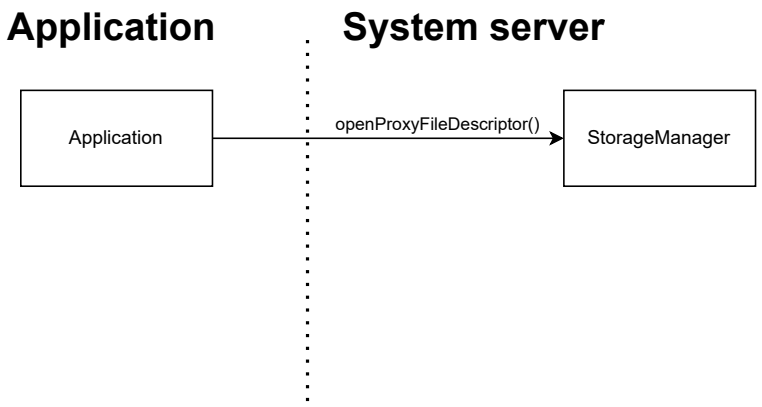
Revocable file descriptors

- Quick look at `RevocableFileDescriptor`

```
public class RevocableFileDescriptor {
    ...

    private final ProxyFileDescriptorCallback mCallback =
            new ProxyFileDescriptorCallback() {
        private void checkRevoked()
                throws ErrnoException {
            if ( mRevoked ) {
                throw new ErrnoException(TAG, OsConstants.EPERM);
            }
        }

        @Override
        public int onRead(long offset, int size, byte[] data)
                throws ErrnoException {
            checkRevoked() ;
            int n = 0;
            while (n < size) {
                ...

                n += Os.pread( mInner , data, n, size - n, offset + n);
            }
            return n;
        }
        ...
    };
}
```

Proxy file descriptors

FUSE mount bridge

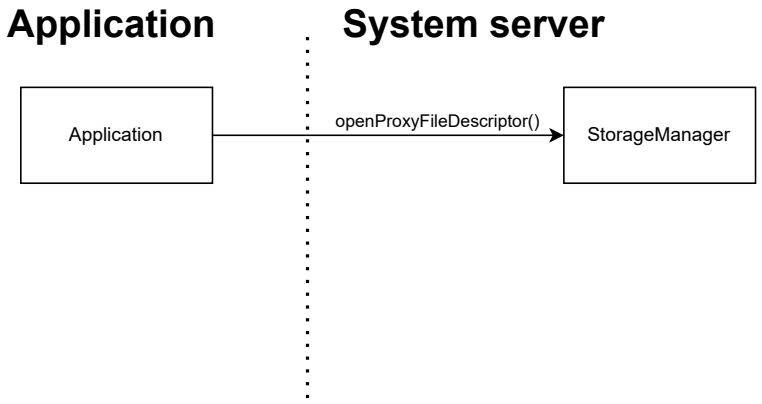


- Application implements `ProxyFileDescriptorCallback`

```
public class MyFileDescriptorCallbacks extends ProxyFileDescriptorCallback
{
    @Override
    public int onRead(long offset, int size, byte[] data)
        throws ErrnoException
    {
        Thread.sleep(2000);
        ...
    }
    ...
}
```

Proxy file descriptors

FUSE mount bridge

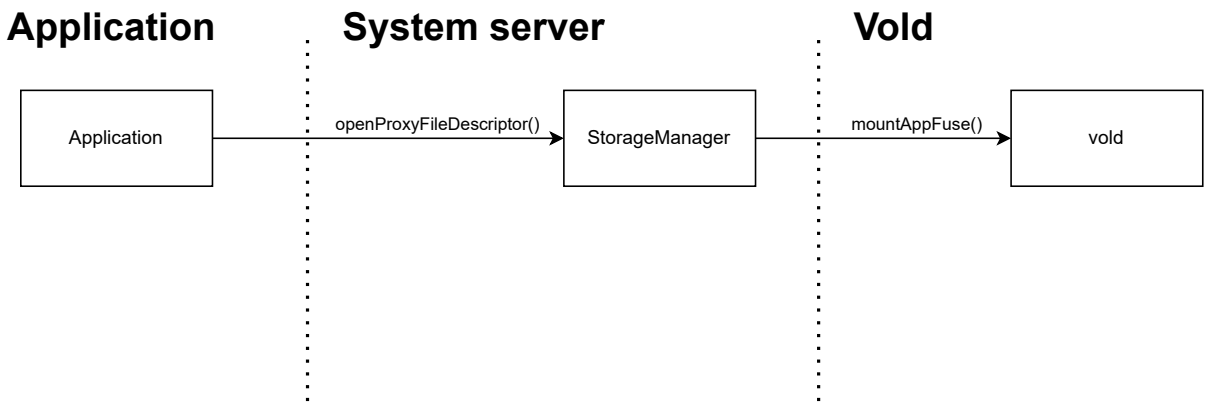


- Application calls `openProxyFileDescriptor()` of `StorageManagerService`

```
StorageManager storageManager =
    (StorageManager)context.getSystemService(Context.STORAGE_SERVICE);

storageManager.openProxyFileDescriptor(ParcelFileDescriptor.MODE_READ_WRITE,
    new MyFileDescriptorCallbacks(), ...);
```

Proxy file descriptors

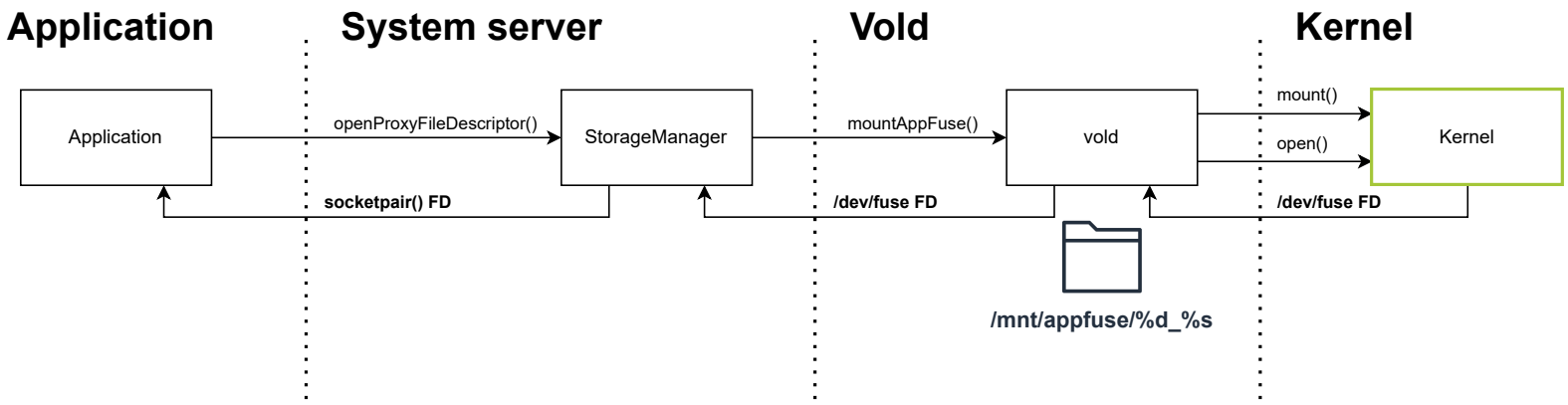
FUSE mount bridge



- `StorageManagerService` communicates with `vold` as it cannot directly access `/dev/fuse`
- Calls `vold`'s `mountAppFuse()` to create the FUSE mount bridge first
- Happens only on the first call to `openProxyFileDescriptor()`

Proxy file descriptors

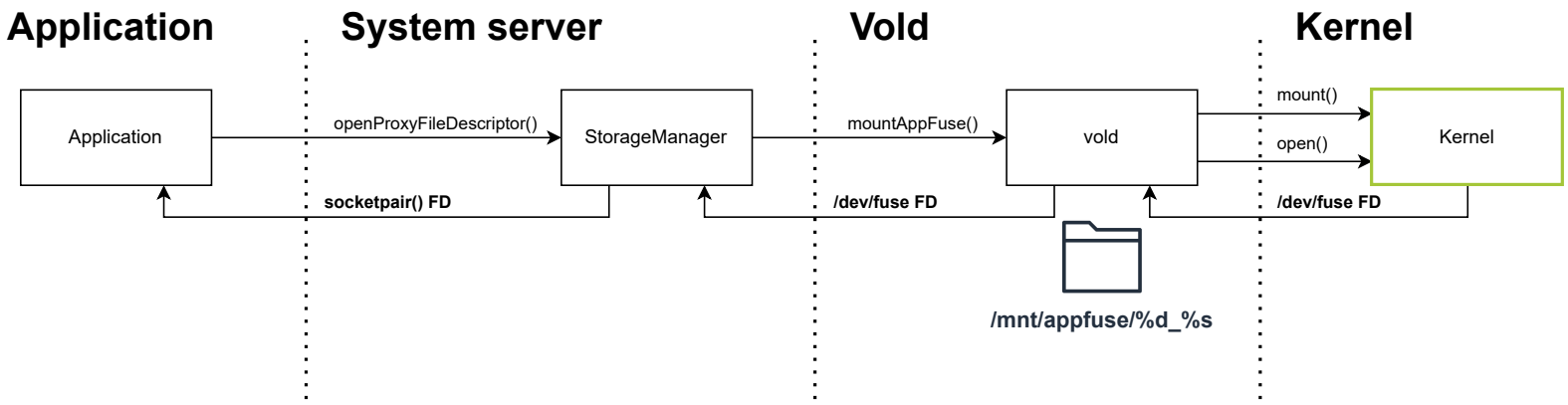
FUSE mount bridge



- `vold` opens `/dev/fuse` and mounts it at `/mnt/appfuse/%d_%s`
- `%d` → Caller UID (i.e., application UID)
- `%s` → Incremental identifier one per application utilizing FUSE

Proxy file descriptors

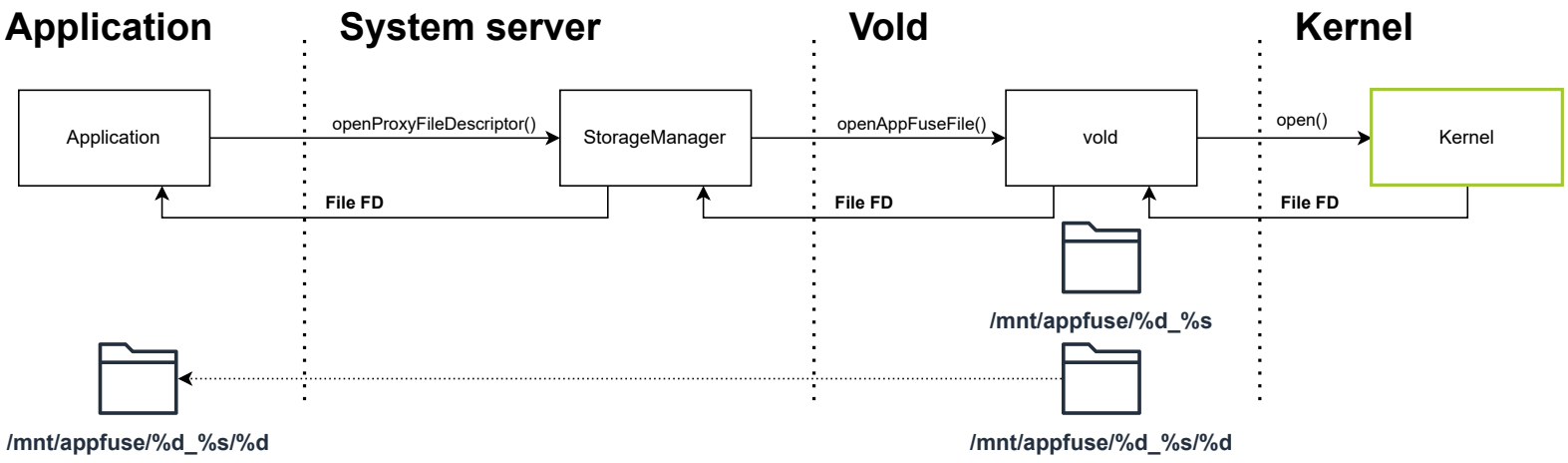
FUSE mount bridge



- The FUSE mount bridge has now been prepared
- `vold` sends the `/dev/fuse` file descriptor to `system_server`
- `system_server` creates a socket pair and sends back one end to the application
- `system_server` does not trust the application to hold a FUSE file descriptor directly

Proxy file descriptors

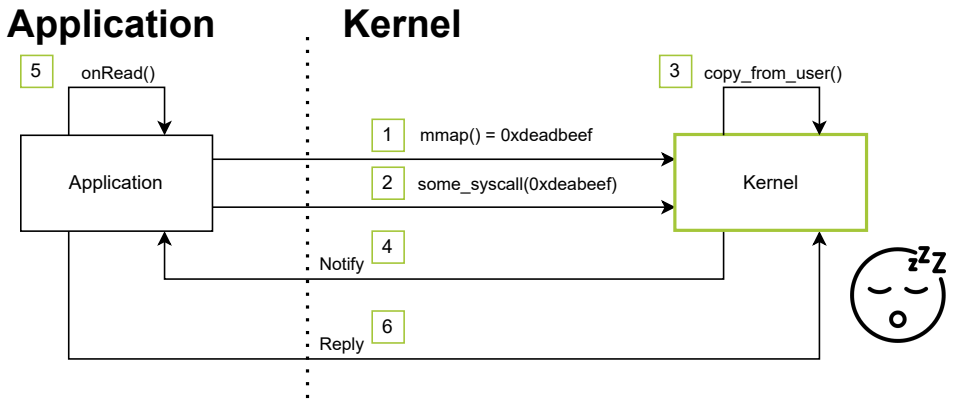
FUSE file



- Now a FUSE file is created
- `system_server` calls `vold`'s `openAppFuseFile()`
- `vold` opens `/mnt/appfuse/%d_%s/%d`
- `%d` → Incremental FUSE file identifier
- The file descriptor is handed over to the application for direct use

Proxy file descriptors

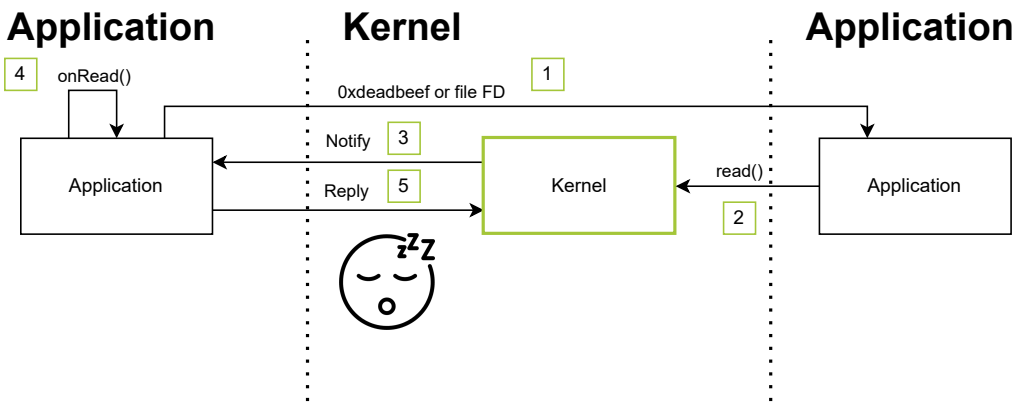
FUSE file

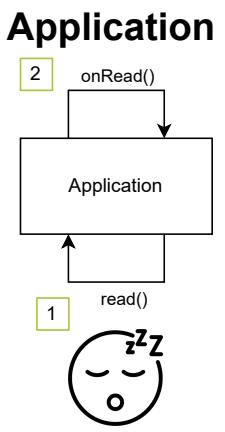


- Application can `mmap()` the FUSE file and use a system-call to have the kernel read or write in the mapped area
- Kernel page-faults and `onRead()` is called to populate the faulting page(s)
- Application introduces latency in `onRead()` (e.g., `Thread.sleep()`)
- Kernel blocks $\longrightarrow$ Success

Proxy file descriptors

FUSE file

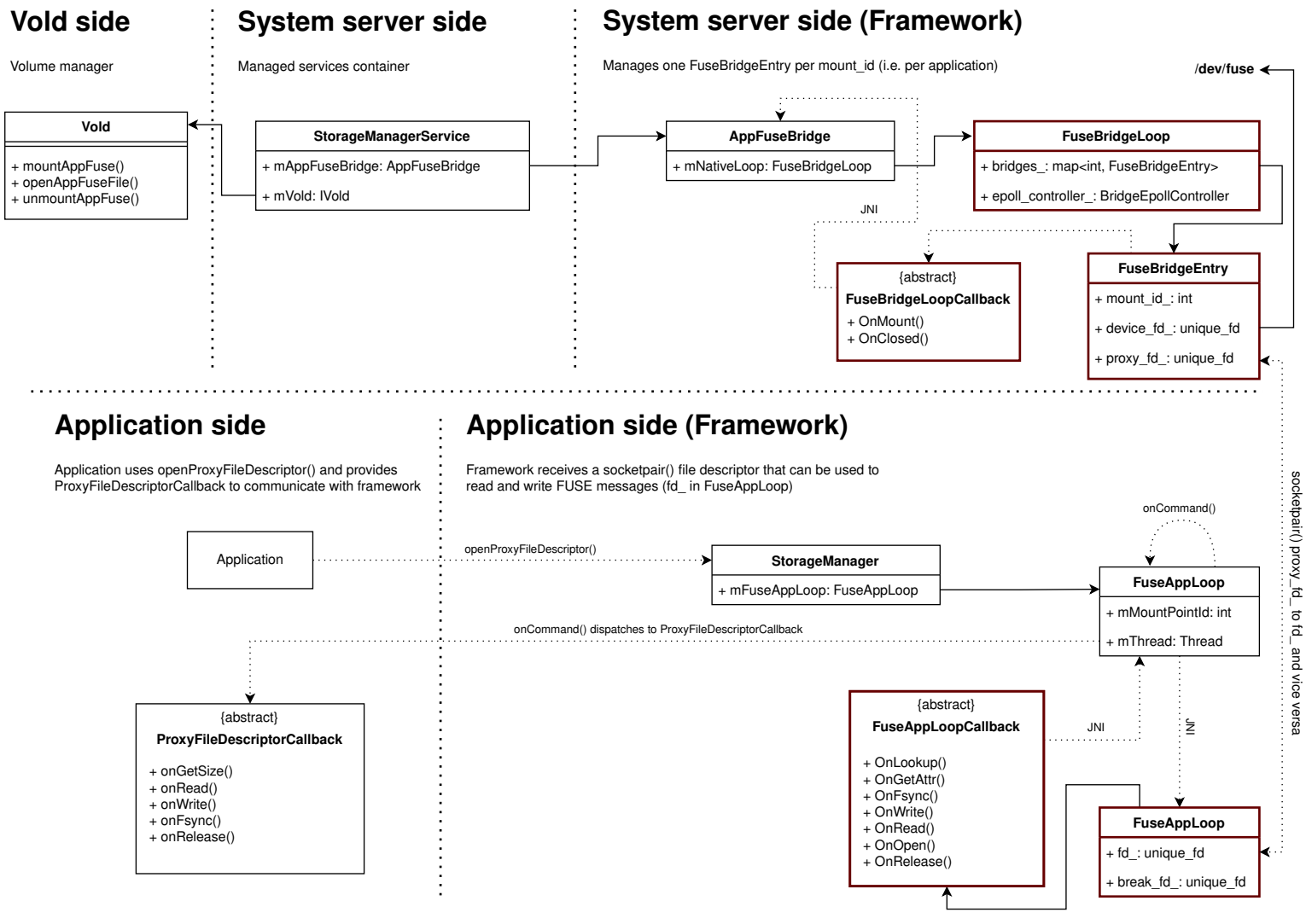




- Application can send the FUSE file to any other application over Binder, or use the file itself
- System-calls issued on the FUSE file descriptor (e.g., `read()`) result in `MyFileDescriptorCallbacks` methods being called (e.g., `onRead()`)
- Added bonus $\rightarrow$ Can block any target process

Proxy file descriptors

Architecture overview



Proxy file descriptors

Proof-of-concept

- Proxy file descriptor PoC
  1. Implement `ProxyFileDescriptorCallback` callbacks
     - Make `onRead()`, `onWrite()` etc. block using `Thread.sleep()`
  2. Create proxy file descriptor using `openProxyFileDescriptor()` of `StorageManagerService`
  3. Use `mmap()` on the above file descriptor
  4. Pass the above mapping to the kernel to have the latter block on page-fault
  5. Close the file descriptor and clean-up

Table of contents

Redaction and transcoding

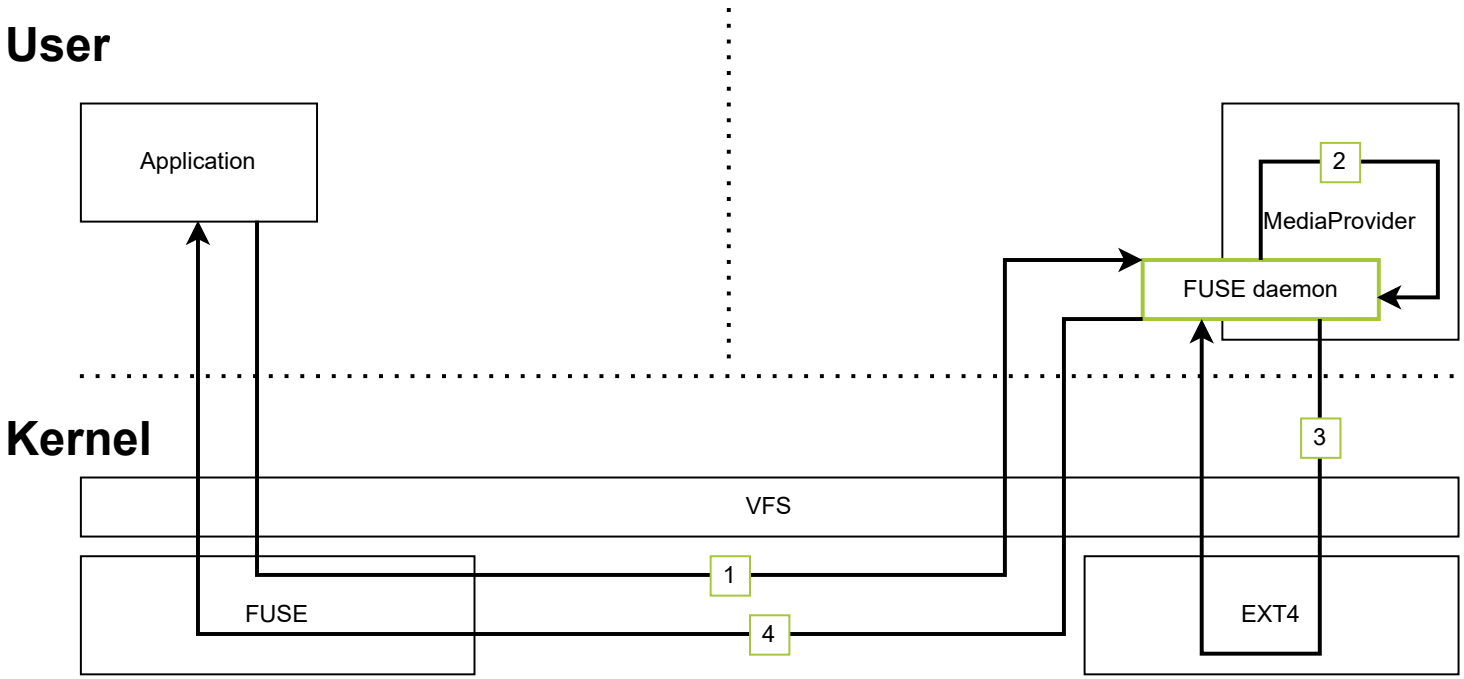
Scoped storage

- Scoped storage
  - Successor to SDCardFS
  - Used for intercepting filesystem-related system-calls from userland to certain locations under `/sdcard`
  - Benefits
    - Improved developer experience
    - Fine-grained access control
    - Privacy
  - Used heavily by the MediaProvider, which is nowadays a separate module
    - An interesting attack surface by itself, but not the topic of today's presentation

Redaction and transcoding

Scoped storage

Figure: Scoped storage on Android. See Demystifying Android's Scoped Storage Defense.



Redaction and transcoding

About

- `MediaProvider` provides automatic redaction and transcoding of media files
  - Redaction $\rightarrow$ EXIF tags and ISO boxes are filtered censored
  - Transcoding $\rightarrow$ Media files are automatically converted to supported media formats
- The official Scoped storage documentation mentions redaction, but no further insights are given:

  Android 11 or higher supports Filesystem in Userspace (FUSE), which enables the MediaProvider module to examine file operations in user space and to gate access to files based on the policy to allow, deny, or redact access.

- On the contrary, transcoding is explained in great depth:

  For devices with compatible media transcoding enabled, Android can automatically convert videos (up to one minute in length) recorded in formats such as HEVC or HDR when the videos are opened by an app that doesn't support the format.

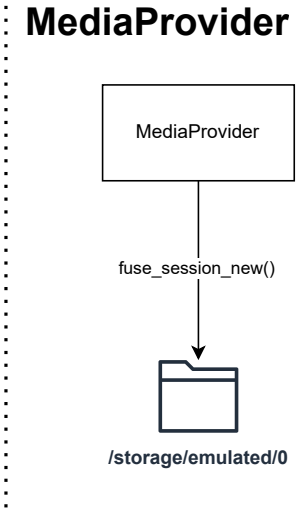
Redaction and transcoding

About

- Here's how it works:
  - Camera application captures video in high definition and stores it in HEVC or HDR format
  - Media player application does not support these formats
  - Media player uses the `MediaProvider` to access the video file and requests transcoding to a supported format and/or censoring of location data found in metadata
  - `MediaProvider` performs the required actions automatically and sends transcoded and/or censored data back to the media player
- Implementation based on FUSE

Redaction and transcoding

MediaProvider FUSE architecture



- `MediaProvider` mounts FUSE on the external storage volume:

```
static struct fuse_lowlevel_ops ops{
    .open = pf_open,
    .read = pf_read,
    ...
};

struct fuse_session *se = fuse_session_new(&args, &ops, sizeof(ops), &fuse_default);
```

- Now all accesses under `/storage/emulated/0` are intercepted by `MediaProvider`'s `FuseDaemon`

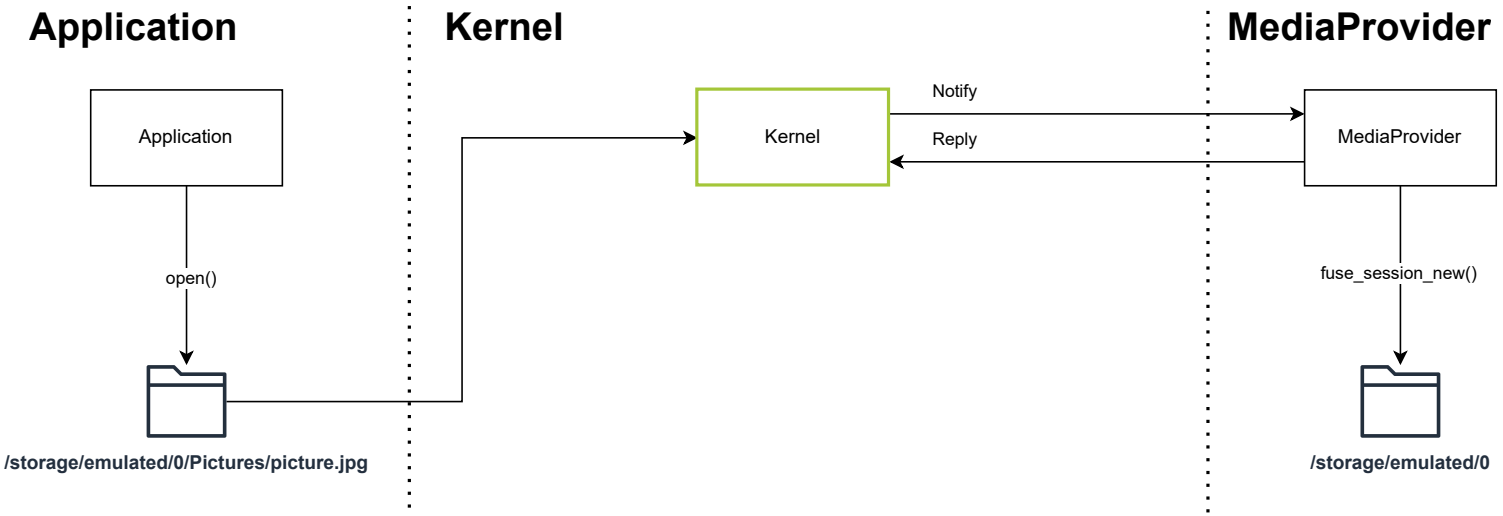
Redaction and transcoding

MediaProvider FUSE architecture

- Application opens a media file under the external storage volume
- Direct filesystem access (e.g., `open()` in C, or `File` in Java)
- Indirect access via content-providers

Redaction and transcoding

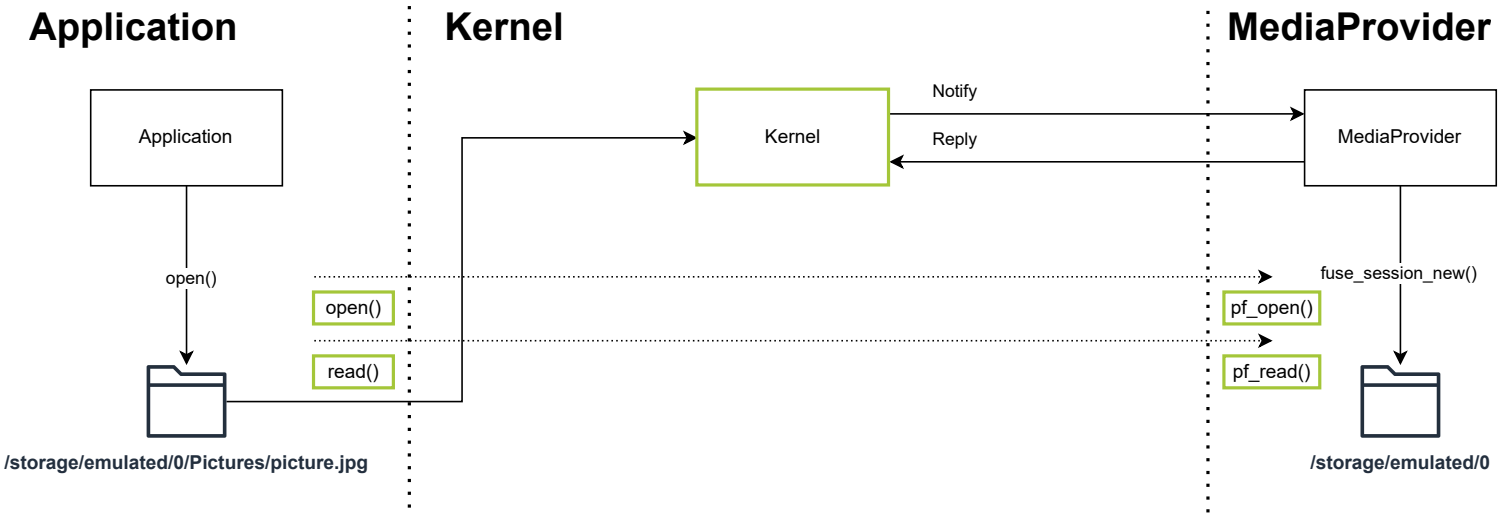
MediaProvider FUSE architecture



- Kernel notifies the process that owns the FUSE mount-point (i.e., `MediaProvider`)
- `MediaProvider` processes the notification, performs arbitrary actions and replies to the kernel notification
- Kernel returns from `open()`
- File descriptor can be mapped using `mmap()` and passed to the kernel

Redaction and transcoding

MediaProvider FUSE architecture



- Eventually application's `open()` results in `MediaProvider`'s `pf_open()`
- Same can be said for other filesystem-related system-calls (e.g., `read()`, `write()`)
- To introduce latencies during page-faults we need to force `MediaProvider` to do extra work during `pf_read()`

Redaction and transcoding

Getting there

- `pf_open()` does some lookups and calls `OnFileOpenForFuse()` (Java)

```
static void pf_open(fuse_req_t req, fuse_ino_t ino, struct fuse_file_info* fi) {
    node* node = fuse->FromInode(ino);
    ...

    const string& io_path = get_path(node);
    const string& build_path = node->BuildPath();
    ...

    const OpenInfo open_info = parse_open_flags(io_path, fi->flags);
    ...

    std::unique_ptr<FileOpenResult> result = fuse->mp-> OnFileOpen (
            build_path, io_path, ctx->uid, ctx->pid, node->GetTransformsReason(),
            open_info.for_write, !open_info.for_write, true);
    ...

    fuse_reply_open(req, fi);
}
```

Redaction and transcoding

Getting there

- `OnFileOpenForFuse()` checks for synthetic paths, retrieves redaction ranges and notifies the transcode-helper

```
@Keep
public FileOpenResult onFileOpenForFuse(String path, String ioPath, int uid, int tid,
        int transformsReason, boolean forWrite, boolean redact,
        boolean logTransformsMetrics) {
    boolean isSuccess = false;
    boolean forceRedaction = false;
    final int userId = UserHandle.myUserId();
    ...

    if ( isSyntheticPath (path, userId)) {

        if ( isRedactedPath (path, userId)) {
            path = ioPath;
            redact = true;
            forceRedaction = true;
        }
    }
    ...

    if (isSuccess && logTransformsMetrics) {
        notifyTranscodeHelperOnFileOpen (path, ioPath, originalUid, transformsReason);
    }
    ...

    return new FileOpenResult(0, originalUid, mediaCapabilitiesUid,
            redact ? getRedactionRangesForFuse (path, ioPath, originalUid, uid, tid,
                    forceRedaction) : new long[0]);
}
```

Redaction and transcoding

Getting there

- Redaction ranges are tuples of the form (start, end) of file offsets holding sensitive metadata

```
public static long[] getRedactionRanges(FileInputStream fis, String mimeType)
        throws IOException {
    final LongArray res = new LongArray();
    ...

    if (ExifInterface.isSupportedMimeType(mimeType)) {
        final ExifInterface exif = new ExifInterface(fis.getFD());
        for (String tag : REDACTED_EXIF_TAGS ) {
            final long[] range = exif. getAttributeRange (tag);
            if (range != null) {
                res.add(range[0]);
                res.add(range[0] + range[1]);
            }
        }
        res.addAll(XmpDataParser.getRedactionRanges(exif));
    }
    ...
}
```

Redaction and transcoding

Getting there

- Redaction ranges are tuples of the form (start, end) of file offsets holding sensitive metadata

```
public static long[] getRedactionRanges(FileInputStream fis, String mimeType)
        throws IOException {
    ...

    if (IsoInterface.isSupportedMimeType(mimeType)) {
        final IsoInterface iso = IsoInterface.fromFileDescriptor(fis.getFD());
        for (int box : REDACTED_ISO_BOXES ) {
            final long[] ranges = iso. getBoxRanges (box);
            for (int i = 0; i < ranges.length; i += 2) {
                long boxTypeOffset = ranges[i] - 4;
                res.add(boxTypeOffset);
                res.add(ranges[i + 1]);
            }
        }
        res.addAll(XmpDataParser.getRedactionRanges(iso));
    }

    return res.toArray();
}
```

Redaction and transcoding

Getting there

- List of sensitive EXIF tags and ISO boxes

```
String[] REDACTED_EXIF_TAGS = new String[]{
        ExifInterface.TAG_GPS_ALTITUDE,
        ExifInterface.TAG_GPS_ALTITUDE_REF,
        ExifInterface.TAG_GPS_AREA_INFORMATION,
        ExifInterface.TAG_GPS_DOP,
        ExifInterface.TAG_GPS_DATESTAMP,
        ExifInterface.TAG_GPS_DEST_BEARING,
        ExifInterface.TAG_GPS_DEST_BEARING_REF,
        ExifInterface.TAG_GPS_DEST_DISTANCE,
        ExifInterface.TAG_GPS_DEST_DISTANCE_REF,
        ExifInterface.TAG_GPS_DEST_LATITUDE,
        ExifInterface.TAG_GPS_DEST_LATITUDE_REF,
        ExifInterface.TAG_GPS_DEST_LONGITUDE,
        ExifInterface.TAG_GPS_DEST_LONGITUDE_REF,
        ExifInterface.TAG_GPS_DIFFERENTIAL,
        ExifInterface.TAG_GPS_IMG_DIRECTION,
        ExifInterface.TAG_GPS_IMG_DIRECTION_REF,
        ExifInterface.TAG_GPS_LATITUDE,
        ExifInterface.TAG_GPS_LATITUDE_REF,
        ExifInterface.TAG_GPS_LONGITUDE,
        ExifInterface.TAG_GPS_LONGITUDE_REF,
        ExifInterface.TAG_GPS_MAP_DATUM,
        ExifInterface.TAG_GPS_MEASURE_MODE,
        ExifInterface.TAG_GPS_PROCESSING_METHOD,
        ExifInterface.TAG_GPS_SATELLITES,
        ExifInterface.TAG_GPS_SPEED,
        ExifInterface.TAG_GPS_SPEED_REF,
        ExifInterface.TAG_GPS_STATUS,
        ExifInterface.TAG_GPS_TIMESTAMP,
        ExifInterface.TAG_GPS_TRACK,
        ExifInterface.TAG_GPS_TRACK_REF,
        ExifInterface.TAG_GPS_VERSION_ID,
};
```

```
int[] REDACTED_ISO_BOXES = new int[]{
        IsoInterface.BOX_LOCI,
        IsoInterface.BOX_XYZ,
        IsoInterface.BOX_GPS,
        IsoInterface.BOX_GPSO,
};
```

Redaction and transcoding

Getting there

- `getExifAttribute()` returns the first occurrence of an EXIF tag

```
private @Nullable ExifAttribute getExifAttribute(@NonNull String tag) {
    ...

    for (int i = 0; i < EXIF_TAGS.length; ++i) {
        Object value = mAttributes[i].get(tag);
        if (value != null) {
            return (ExifAttribute) value;
        }
    }
    return null;
}
```

- `getBoxRanges()` returns all relevant ISO boxes

```
public @NonNull long[] getBoxRanges(int type) {
    LongArray res = new LongArray();
    for (Box box : mFlattened) {
        if (box.type == type) {
            for (int i = 0; i < box.range.length; i += 2) {
                res.add(box.range[i] + box.headerSize);
                res.add(box.range[i] + box.range[i + 1]);
            }
        }
    }
    return res.toArray();
}
```

- So, we can't have several `TAG_GPS_ALTITUDE` EXIF tags, but we can have several `BOX_LOCI` ISO boxes!

Redaction and transcoding

Getting there

- How does a synthetic path look like?

```
@VisibleForTesting
static String getSyntheticRelativePath() {
    return buildPath(null, TRANSFORMS_DIR, SYNTHETIC_DIR).getPath();
}
```

```
public static boolean isSyntheticPath(String path, int userId) {
    final String syntheticDir = buildPrimaryVolumeFile(userId,
            getSyntheticRelativePath()).getAbsolutePath();
    return path != null && startsWith(path, syntheticDir);
}
```

- `buildPath()` → `/storage/emulated/0/.transforms/synthetic`

Redaction and transcoding

Getting there

- How does a redacted path look like?

```
public static final String REDACTED_URI_ID_PREFIX = "RUID";
```

```
public static String getRedactedRelativePath() {
    return buildPath(null, TRANSFORMS_DIR, SYNTHETIC_DIR, REDACTED_DIR).getPath();
}
```

```
public static boolean isRedactedPath(String path, int userId) {
    if (path == null) return false;

    final String redactedDir = buildPrimaryVolumeFile(userId, getRedactedRelativePath())
            .getAbsolutePath();
    final String fileName = extractFileName(path);

    return fileName != null
            && startsWith(path, redactedDir)
            && startsWith(fileName, REDACTED_URI_ID_PREFIX)
            && fileName.length() == REDACTED_URI_ID_SIZE;
}
```

- `buildPath()` → `/storage/emulated/0/.transforms/synthetic/redacted`
- `fileName` → `RUID` + 32 characters

Redaction and transcoding

Getting there

- But who creates these files anyway? The `MediaProvider`!

```
@Nullable @VisibleForTesting Uri getRedactedUri(@NonNull Uri uri) {
    if (! isUriSupportedForRedaction (uri)) { ... }
    ...

    try (final Cursor c = helper.runWithoutTransaction(
            (db) -> db.query("files",
                    new String[]{FileColumns.REDACTED_URI_ID}, FileColumns._ID + "=?",
                    new String[]{uri.getLastPathSegment()}, null, null, null))) {
        ...

        String redactedUriID =
            c.getString(c.getColumnIndex(FileColumns.REDACTED_URI_ID));
        if (redactedUriID == null) {
            redactedUriID = REDACTED_URI_ID_PREFIX +
                    UUID.randomUUID().toString().replace("-", "");
            ContentValues cv = new ContentValues();
            cv.put(FileColumns.REDACTED_URI_ID, redactedUriID);
            int rowsAffected = helper.runWithTransaction(
                    (db) -> db.update("files", cv, FileColumns._ID + "=?",
                            new String[]{uri.getLastPathSegment()}));
            ...
        }

        final Uri.Builder builder = ContentUris.removeId(uri).buildUpon();
        builder.appendPath(redactedUriID);
        ...
    }
}
```

Redaction and transcoding

Getting there

- How can one reach `getRedactedUri()`? `MediaProvider` implements `ContentProvider`'s `call()`!

```
@Override
public Bundle call(String method, String arg, Bundle extras) {
    ...

    try {
        return callInternal(method, arg, extras);
    } finally {
        Trace.endSection();
    }
}
```

```
private Bundle callInternal(String method, String arg, Bundle extras) {
    switch (method) {
        ...

        case MediaStore.GET_REDACTED_MEDIA_URI_CALL: {
            return getResultForGetRedactedMediaUri(extras);
        }
        case MediaStore.GET_REDACTED_MEDIA_URI_LIST_CALL: {
            return getResultForGetRedactedMediaUriList(extras);
        }
        ...
    }
}
```

Redaction and transcoding

Getting there

- `MediaStore` (front-end to `MediaProvider`) has a wrapper

```
@RequiresApi(Build.VERSION_CODES.S)
@Nullable
public static Uri getRedactedUri(@NonNull ContentResolver resolver, @NonNull Uri uri) {
    final String authority = uri.getAuthority();
    try (ContentProviderClient client = resolver.acquireContentProviderClient(authority)) {
        final Bundle in = new Bundle();
        final String userId = uri.getUserInfo();
        in.putParcelable(EXTRA_URI, maybeRemoveUserId(uri));
        final Bundle out = client.call(GET_REDACTED_MEDIA_URI_CALL, null, in);
        return maybeAddUserId((Uri) out.getParcelable(EXTRA_URI), userId);
    } catch (RemoteException e) {
        throw e.rethrowAsRuntimeException();
    }
}
```

Redaction and transcoding

Getting there

- So, now we know what happens when an application opens a file via the `MediaProvider` (`pf_open()` etc.)
- `MediaStore.openFileDescriptor()` will return a `ParcelFileDescriptor` to the caller
- Reading from that descriptor will trigger `pf_read()` on the `MediaProvider` process
- Let's see what happens

Redaction and transcoding

Getting there

- `pf_read()` actually triggers the transcoding and redaction flows!

```
static void pf_read(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
        struct fuse_file_info* fi) {
    handle* h = reinterpret_cast<handle*>(fi->fh);
    ...

    const bool direct_io = !h->cached;
    struct fuse* fuse = get_fuse(req);

    node* node = fuse->FromInode(ino);

    if (!node->IsTransformsComplete()) {
        if (!fuse->mp-> Transform (node->BuildPath(), node->GetIoPath(),
                node->GetTransforms(), node->GetTransformsReason(),
                req->ctx.uid, h->uid, h->transforms_uid)) {
            fuse_reply_err(req, EFAULT);
            return;
        }
        node->SetTransformsComplete(true);
    }

    fuse->fadviser.Record(h->fd, size);

    if (h->ri-> isRedactionNeeded ()) {
        do_read_with_redaction (req, size, off, fi, direct_io);
    } else {
        do_read(req, size, off, fi, direct_io);
    }
}
```

Redaction and transcoding

Getting there

- `do_read_with_redaction()` iterates through all `ranges[]` and filters out the corresponding data

```
static void do_read_with_redaction(fuse_req_t req, size_t size, off_t off,
        fuse_file_info* fi, bool direct_io) {
    handle* h = reinterpret_cast<handle*>(fi->fh);

    std::vector<ReadRange> ranges;
    h->ri->getReadRanges(off, size, &ranges);
    ...

    const size_t num_bufs = ranges.size();
    ...

    for (int i = 0; i < num_bufs; ++i) {

        const ReadRange& range = ranges[i];
        if (range.is_redaction) {
            create_mem_fuse_buf(range.size, &(bufvec.buf[i]), get_fuse(req));
        } else {
            create_file_fuse_buf(range.size, range.start, h->fd, &(bufvec.buf[i]));
        }
    }

    fuse_reply_data(req, &bufvec, static_cast<fuse_buf_copy_flags>(0));
}
```

- An attacker can introduce delays by adding a large number of censored ISO boxes in a MP4 file!

Redaction and transcoding

Proof-of-concept

- Redaction PoC
  1. Create MP4 file (e.g., `test.mp4`) with thousands of LOCI ISO boxes
  2. Make sure `test.mp4` is indexed by the `MediaProvider`
  3. Use `MediaStore.getRedactedUri()` to get a redaction URI for `test.mp4`
  4. Use `MediaStore.openFileDescriptor()` on the above URI
  5. Use `mmap()` on the above file descriptor
  6. Pass the above mapping to the kernel; kernel thread will block on page-fault while `MediaProvider`'s FUSE daemon performs the redaction process
  7. Close the file descriptor and clean-up

Redaction and transcoding

Getting there

- `transcode()` blocks waiting for a response from the `MediaTranscodingService`

```
public boolean transcode(String src, String dst, int uid, int reason) {
    StorageTranscodingSession storageSession = null;
    TranscodingSession transcodingSession = null;
    CountDownLatch latch = null;
    boolean result = false;
    int errorCode = TranscodingSession.ERROR_SERVICE_DIED;
    int failureReason = TRANSCODING_DATA__FAILURE_CAUSE__TRANSCODING_SERVICE_ERROR;
    ...

    storageSession = mStorageTranscodingSessions.get(src);
    if (storageSession == null) {
        transcodingSession = enqueueTranscodingSession(src, dst, uid, latch);
        storageSession = new StorageTranscodingSession(transcodingSession, latch,
                src, dst);
        mStorageTranscodingSessions.put(src, storageSession);
        ...
    }
    ...

    failureReason = waitTranscodingResult(uid, src, transcodingSession, latch);
    errorCode = transcodingSession.getErrorCode();
    result = failureReason == TRANSCODING_DATA__FAILURE_CAUSE__CAUSE_UNKNOWN;
    ...

    return result;
}
```

Redaction and transcoding

Getting there

- `enqueueTranscodingSession()` sends an asynchronous transcoding request to the `MediaTranscodingService`

```
private TranscodingSession enqueueTranscodingSession(String src, String dst, int uid,
        final CountDownLatch latch) throws UnsupportedOperationException, IOException {
    final MediaTranscodingManager mediaTranscodeManager =
            mContext.getSystemService(MediaTranscodingManager.class);
    File file = new File(src);
    File transcodeFile = new File(dst);
    Uri uri = Uri.fromFile(file);
    Uri transcodeUri = Uri.fromFile(transcodeFile);
    ParcelFileDescriptor srcPfd = ParcelFileDescriptor.open(file,
            ParcelFileDescriptor.MODE_READ_ONLY);
    ParcelFileDescriptor dstPfd = ParcelFileDescriptor.open(transcodeFile,
            ParcelFileDescriptor.MODE_READ_WRITE);
    ...

    VideoTranscodingRequest request =
            new VideoTranscodingRequest.Builder(uri, transcodeUri, format)
                    .setClientUid(uid)
                    .setSourceFileDescriptor(srcPfd)
                    .setDestinationFileDescriptor(dstPfd)
                    .build();
    TranscodingSession session = mediaTranscodeManager. enqueueRequest (request, ...)
    ...

    return session;
}
```

Redaction and transcoding

Getting there

- `waitTranscodingResult()` computes a timeout based on the file size and uses the latch to block

```
private static final double TRANSCODING_TIMEOUT_COEFFICIENT = 10;

private int getTranscodeTimeoutSeconds(String file) {
    double sizeMb = (new File(file).length() / (1024 * 1024));
    sizeMb = Math.max(sizeMb, 1);
    return (int) (sizeMb * TRANSCODING_TIMEOUT_COEFFICIENT);
}
```

```
private int waitTranscodingResult(int uid, String src, TranscodingSession session,
        CountDownLatch latch) {
    ...

    int timeout = getTranscodeTimeoutSeconds(src);
    ...

    boolean latchResult = latch.await(timeout, TimeUnit.SECONDS);
    int sessionResult = session.getResult();
    boolean transcodeResult = sessionResult == TranscodingSession.RESULT_SUCCESS;
    ...
}
```

Redaction and transcoding

Proof-of-concept

- Transcoding PoC
  - Depends on property `sys.fuse.transcode_enabled` (set to `true` by default)
  - Can be configured:
    - Using code
    - In the application manifest
  - For demonstration purposes we hook the `MediaProvider` to force transcoding even when the victim application does not meet the requirements (e.g., YouTube)

Redaction and transcoding

Proof-of-concept

- Transcoding PoC
  1. Create MP4 file (e.g., `test.mp4`) in HEVC or HDR format
  2. Make sure `test.mp4` is indexed by the `MediaProvider`
  3. Use `MediaStore.openFileDescriptor()` on the above URI
  4. Use `mmap()` on the above file descriptor
  5. Pass the above mapping to the kernel; kernel thread will block on page-fault while `MediaTranscodingService` performs the media transcoding process
  6. Close the file descriptor and clean-up

Table of contents

Incremental filesystem

About

- Android 11 introduced incremental application installation
- Depends on Incremental File System kernel driver
- Enables the Android OS to receive streamed APKs over ADB
- But, in fact, it's more powerful than that (e.g., streaming from Google Play)
  - Application files can be prioritized to stream in an order from a source
  - Certain parts of the application can start executing before, a potentially large, APK download has finished
- Incremental File Streaming of a Package File for Application Installation on a Mobile Electronic Device by J. Eason et al.

Incremental filesystem

About

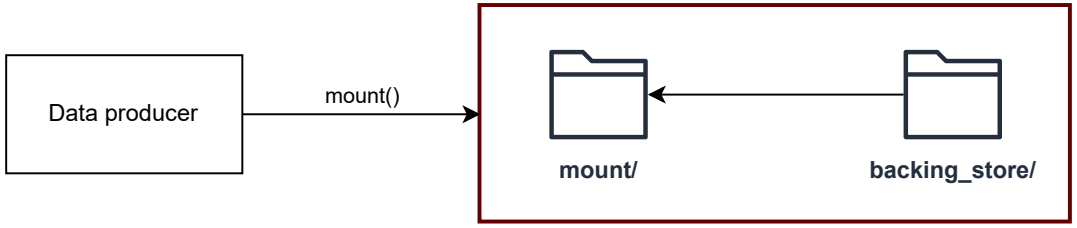
- IncFS is a stackable filesystem

```
# cd /data/local/tmp
# mkdir backing_store mount
# mount -t incremental-fs backing_store/ mount/
# ls -la mount/
total 3
drwxr-xr-x  4 root  root     0 2025-04-28 22:53 .
drwxrwx--x 10 shell shell 3452 2025-04-28 22:50 ..
-rw-rw-rw-  1 root  root     0 1970-01-01 02:00 .blocks_written
drwxr-xr-x  2 root  root     0 2025-04-28 22:53 .incomplete
drwxr-xr-x  2 root  root     0 2025-04-28 22:53 .index
-rw-rw-rw-  1 root  root     0 1970-01-01 02:00 .log
-rw-rw-rw-  1 root  root     0 1970-01-01 02:00 .pending_reads
```

- Interacting with IncFS requires use of `ioctl()`
- Open file descriptor to `.pending_reads`, use it as the first argument to `ioctl()`
  - `INCFS_IOC_CREATE_FILE` → Create file
  - `INCFS_IOC_FILL_BLOCKS` → Write data in file

Incremental filesystem

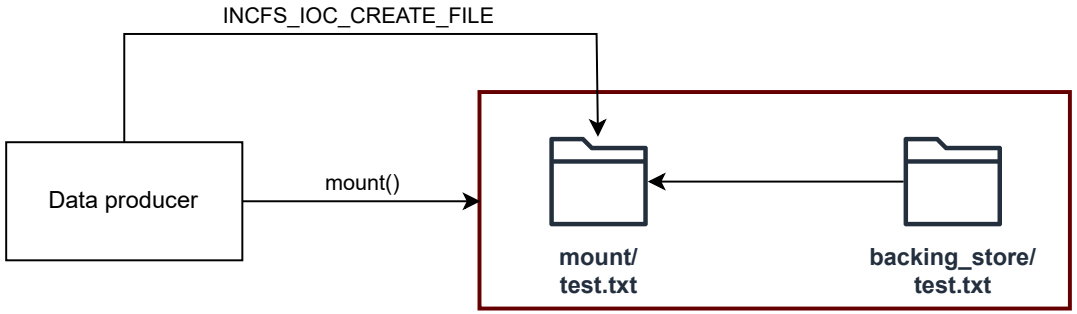
About



- A data producer mounts the incremental filesystem
- This can be done via `vold`
  - `IVold::mountIncFs()` and `IVold::unmountIncFs()`
  - `system_server` → `vold`
  - `system_app` → `incremental_service` → `vold`

Incremental filesystem

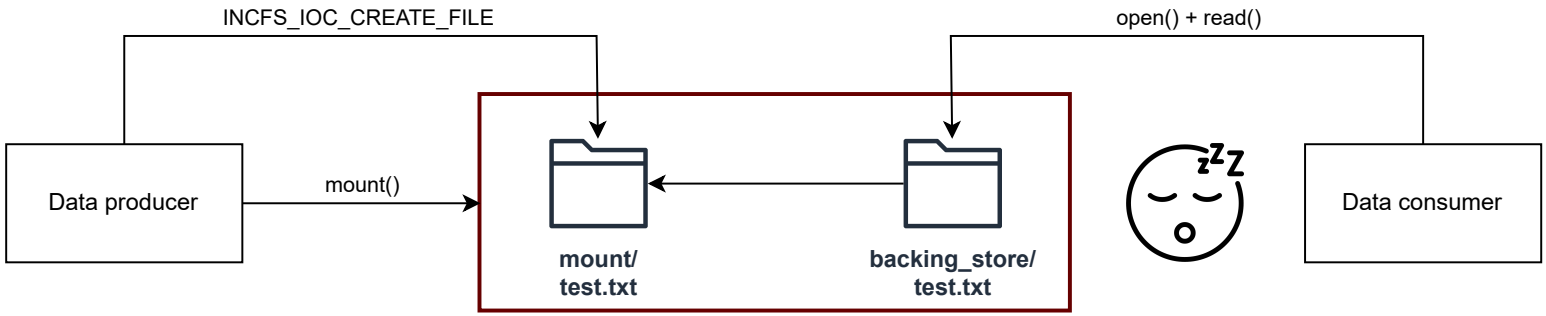
About



- Data producer creates a file using `INCFS_IOC_CREATE_FILE`
- The file appears under both `backing_store/` and `mount/`

Incremental filesystem

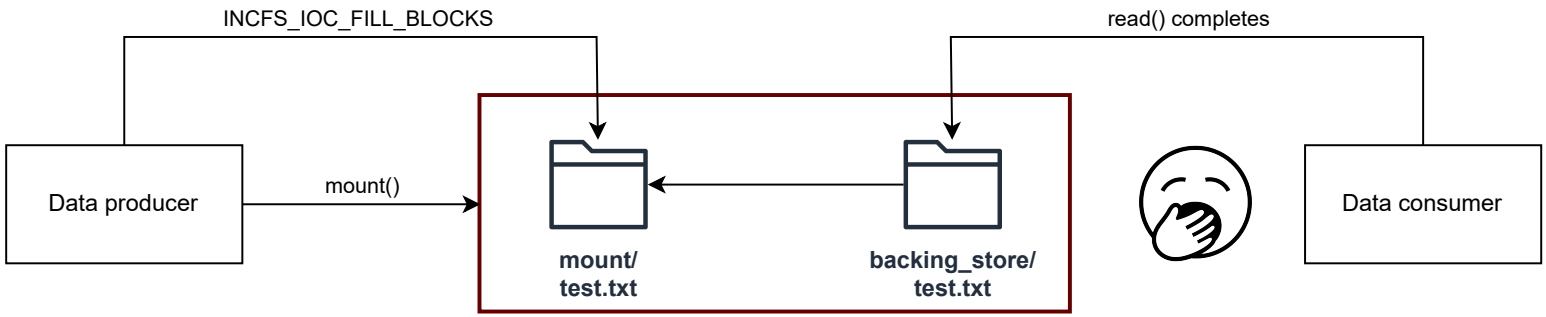
About



- A data consumer calls `open()` and `read()` on the newly created file under `backing_store/`
- Consumer blocks because no data has been written yet

Incremental filesystem

About



- Data producer uses `INCFS_IOC_FILL_BLOCKS` to write block(s) of data in the newly created file
- As soon as the requested blocks are available, the consumer is woken up
- The kernel can play the role of the consumer
  - File can be `mmap()`'ed and passed to the kernel
  - Blocking read initiated on page-fault

Incremental filesystem

Getting there

- `struct file_operations .read_iter` is set to `generic_file_read_iter()`
- Page-cache is populated via `read_folio()`

```
static int read_folio(struct file *f, struct folio *folio)
{
    loff_t offset = 0, size = 0;
    ssize_t total_read = 0;
    struct data_file *df = get_incfs_data_file(f);
    int block_index, result = 0;
    ...

    block_index = (offset + df->df_mapped_offset) /
        INCFS_DATA_FILE_BLOCK_SIZE;
    size = df->df_size;
    ...

    while (offset + total_read < size) {
        ssize_t bytes_to_read = min_t(loff_t, size - offset - total_read,
              INCFS_DATA_FILE_BLOCK_SIZE);
        result = read_single_page_timeouts(df, f, block_index,
            range(page_start + total_read, bytes_to_read),
            tmp, &delayed_min_us);
        total_read += result;
        block_index++;
        ...
    }
    ...
}
```

Incremental filesystem

Getting there

- We eventually get to `wait_for_data_block()`

```
static int read_single_page_timeouts(struct data_file *df, struct file *f,
        int block_index, struct mem_range range, struct mem_range tmp,
        unsigned int *delayed_min_us)
{
    struct incfs_read_data_file_timeouts timeouts = {
        .max_pending_time_us = U32_MAX,
    };
    ...

    return incfs_read_data_file_block(range, f, block_index, tmp,
        &timeouts, delayed_min_us);
}
```

```
ssize_t incfs_read_data_file_block(struct mem_range dst, struct file *f, int index,
        struct mem_range tmp, struct incfs_read_data_file_timeouts *timeouts,
        unsigned int *delayed_min_us)
{
    ssize_t result;
    struct data_file_block block = {};
    struct data_file *df = get_incfs_data_file(f);
    ...

    result = wait_for_data_block(df, index, &block, timeouts, delayed_min_us);
    ...

    return result;
}
```

Incremental filesystem

Getting there

- `wait_for_data_block()` will wait until the required block of data arrives

```
static int wait_for_data_block(struct data_file *df, int block_index,
        struct data_file_block *res_block,
        struct incfs_read_data_file_timeouts *timeouts,
        unsigned int *delayed_min_us)
{
    struct data_file_block block = {};
    ...

    error = get_data_file_block(df, block_index, &block);
    ...

    if (is_data_block_present(&block)) {
        ...
    } else {
        if (timeouts && timeouts->max_pending_time_us) {
            read = add_pending_read(df, block_index);
            ...
        }
        ...
    }
    ...

    wait_res =
        wait_event_interruptible_timeout(segment->new_data_arrival_wq,
            ( is_read_done(read)),
            usecs_to_jiffies(timeouts->max_pending_time_us));

    remove_pending_read(df, read);
    ...
}
```

Incremental filesystem

Getting there

- Pending reads unblock when data blocks are filled via `ioctl()`

```
static long ioctl_fill_blocks(struct file *f, void __user *arg)
{
    struct incfs_fill_blocks __user *usr_fill_blocks = arg;
    struct incfs_fill_blocks fill_blocks;
    struct data_file *df = get_incfs_data_file(f);
    ...

    if (copy_from_user(&fill_blocks, usr_fill_blocks, sizeof(fill_blocks)))
        return -EFAULT;
    ...

    for (i = 0; i < fill_blocks.count; i++) {
        struct incfs_fill_block fill_block = {};
        ...

        if (fill_block.flags & INCFS_BLOCK_FLAGS_HASH) {
            error = incfs_process_new_hash_block(df, &fill_block, data_buf);
        } else {
            error = incfs_process_new_data_block(df, &fill_block, data_buf, &complete);
        }
        ...
    }
    ...
}
```

Incremental filesystem

Getting there

- On success, `incfs_process_new_data_block()` calls `notify_pending_reads()`

```
int incfs_process_new_data_block(struct data_file *df, struct incfs_fill_block *block,
        u8 *data, bool *complete)
{
    struct data_file_segment *segment = NULL;
    struct data_file_block existing_block = {};
    int error = 0;
    ...

    segment = get_file_segment(df, block->block_index);
    ...

    error = get_data_file_block(df, block->block_index, &existing_block);
    ...

    if (is_data_block_present(&existing_block))
        return 0;
    ...

    error = incfs_write_data_block_to_backing_file(bfc, range(data, block->data_len),
        block->block_index, df->df_blockmap_off, flags);
    ...

    if (!error)
        notify_pending_reads (mi, segment, block->block_index);
    ...

    return error;
}
```

Incremental filesystem

Proof-of-concept

- We now have enough information for a PoC
  1. Get `IVold` binder interface
     - Alternatively use `IIncrementalService`
  2. Create empty directories `backing_store/` and `mount/`
  3. Call `IVold::mountIncFs()`
  4. Use `INCFS_IOC_CREATE_FILE` to create `test.txt`
  5. Map `backing_store/test.txt` using `mmap()`
  6. Pass it to the kernel and have the latter block
  7. Use `INCFS_IOC_FILL_BLOCKS` to write data in the file and unblock the kernel
  8. Call `IVold::unmountIncFs()` to clean-up

Table of contents

Conclusions and takeaways

- We have showcased 4 novel techniques for achieving primitives similar to those of `userfaultfd()`
  - Available under `untrusted_app`, `system_app` and `system_server` SELinux domains
  - Source domains from where kernel LPEs are usually executed
- For long time FUSE and others were considered unusable on Android (or no one wanted to talk about them)
- Dig deep in the AOSP ecosystem and you will find gems, trust me, there are more
- Maybe we will talk about them next year? `:-)`

Table of contents

Questions

Questions? Kthanxbai!