Lab Assignment 3
DD2525

Alexandru Matcov[1] and Jakob Ström[2]

[1]matcov@kth.se
[2]jakstrom@kth.se

May 16, 2025

Contents

Chapter 1

Intent Sniffing

1.1 Broadcast System in Android

Android's broadcast system provides a mechanism for apps to send or receive messages across the entire system. These messages are known as broadcasts and are implemented using Intent objects. The broadcast system is similar to the publish-subscribe pattern, where components can:

- Publish broadcasts: Sending components create and transmit Intent objects using methods like *sendBroadcast()*, which we use for this lab.
- Subscribe to broadcasts: Receiving components register to receive specific broadcasts by declaring intent filters, either statically in the manifest or dynamically through *registerReceiver()* which we did for our intent sniffer.

Broadcasts serve as a system-wide event bus and enable loosely coupled integration between components that may not directly know about each other. There are two main types of broadcasts, explicit and implicit. Explicit broadcasts target a specific app or component, so that only that specific component receives that message, while implicit ones does not have a target, the Android reference monitor has to figure out targets based on the included action of the intent. Implicit intents are thus great in regards to flexibility, as it is sent to all appropriate components/apps which allows different apps to respond to common system notifications (such as battery changes or connectivity status) or other events (such as location updates for this lab) without needing tight integration.

1.2 Intent Sniffer

For the lab, we have the location app which through ForegroundLocationService broadcasts two intents implicitly. One intent, the locationAppIntent, is sent intra-app to the locationApp, and the second, weatherIntent is sent inter-app to the weather app. Each intent has its corresponding action:

```
weatherIntent.setAction("tcs.lbs.weather_app.
    WeatherBroadcastReceiver");
locationAppIntent.setAction("tcs.lbs.locationapp.
    MainActivityReceiver");
```

The *putExtra()* method is used to add additional information to intents, and the location data is added in this lab to our intents:

```
locationAppIntent.putExtra("Location", _location);
weatherIntent.putExtra("Location", _location);
```

Finally, when the location is changed, a broadcast is sent with these intents.

```
sendBroadcast(locationAppIntent);
sendBroadcast(weatherIntent);
```

As these broadcasts are sent implicitly, we simply implemented our intent sniffer by registering a new receiver with the two relevant actions (see the previous uses of setAction 1.2) added to the intent filter. The BroadcastReceiver then receives the dispatched broadcasts, with the included location variables.

1.3 Mitigations for the Intra-and Inter-app Sniffing

1.3.1 Intra-app

One solution to overcome intra-app broadcast sniffing is to use permission-protected broadcasts. By restricting the broadcast to apps that hold a certain permission, we verify that the receiver actually is allowed to receive our broadcast. In our case we can as a sender specify that the receiver should have the permission label to access the location, which is then checked by the reference monitor.

A second solution is to use the LocalBroadcastManager, which is a class that allows intra-app communication without routing it through the system-

wide broadcast system. An intent sniffer could thus not intercept the broadcasts, as they are specifically localized within the app only. This works specifically for this lab, as we are using Android API 28, but it was deprecated for level 29.

A third solution would be to use explicit intents instead of explicit, as an explicit intent is delivered to its specified target component instead of using intent filtering, meaning that sniffing would be avoided. By specifying the exact component (i.e., package and class) to receive the intent, the system bypasses the intent resolution process and sends the intent directly to the target.

1.3.2 Inter-app

The locationApp is currently using implicit intents for its inter-app communication (see 1.2), which means any app can register for its broadcasts as long as it knows which action string to listen for. The inter-app broadcasting is thus also vulnerable to intent sniffing.

One solution to the inter-app sniffing would here to also use explicit intents. By specifying the exact component (i.e., package and class) to receive the intent, the system bypasses the intent resolution process and sends the intent directly to the target, same as for the intra-app mitigation.

Another solution would be to, similar as for the intra-app solution, to add permission-protected broadcasts. By requiring that the receiving app hold a specific custom-defined permission, in our case the location permission, the sender ensures that only trusted apps can register for and receive these broadcasts. This permission is declared by the receiving app and must also be granted by the user.

1.4 Implemented Prevention

1.4.1 Intra-app

The implemented prevention technique for the intra-app sniffing was to add the extra permission label to the broadcast.

```
sendBroadcast(locationAppIntent, android.Manifest.
   permission.ACCESS_FINE_LOCATION);
```

This ensures that only components within the app that have been granted ACCESS_FINE_LOCATION can receive the broadcast.

1.4.2 Inter-app

For inter-app prevention, a similar permission-based approach was applied:

```
sendBroadcast(weatherIntent, android.Manifest.permission.
    ACCESS_FINE_LOCATION);
```

In addition, we ensured that runtime permission checks were enforced for the weather app before receiving the broadcast, by first checking if the app currently has access to the location, and if not, request access to it:

```
if (ContextCompat.checkSelfPermission(this, Manifest.
    permission.ACCESS_FINE_LOCATION) != PackageManager.
    PERMISSION_GRANTED)
        {
            ActivityCompat.requestPermissions(this, new
                String[]{Manifest.permission.
                ACCESS_FINE_LOCATION}, 1);
        }
```

We also updated the manifest file of the receiving app (weatherApp) to declare the required permission:

```
<uses-permission android:name="android.permission.
    ACCESS_FINE_LOCATION" />
```

These combined changes ensure that only apps explicitly granted permission can receive the sensitive location data.

As the intent sniffer stopped updating after these mitigation techniques were implemented, it demonstrated that neither the intra-app nor the inter-app broadcast sniffing exploits work, as it would continuously update and display the location if at least one of the broadcasts were successfully intercepted.

Chapter 2

Confused Deputy Attacks

2.1 Vulnerability Overview

The Notes application's `DatabaseActivity` is vulnerable to a confused deputy attack, where an unauthorized component can manipulate the app's database by exploiting the activity's overly permissive intent handling.

2.2 Identified Vulnerabilities

The fundamental issue is that the `DatabaseActivity` does not verify the origin of the intent that triggers its operations. It implicitly trusts any component that can send it an intent, regardless of whether that component is from the same application or a different one. This creates a classic "confused deputy" scenario:

- The `DatabaseActivity` has legitimate authority to modify the Notes app's database.
- Accept commands (via intents) without checking who sent them.
- An external app can craft intents that make the `DatabaseActivity` perform operations on behalf of the external app.

The core vulnerability is the lack of authentication or permission checking before performing sensitive database operations. The activity is "confused" about who is really in charge of its actions.

2.3 Proposed Defense Mechanisms

2.3.1 Solution 1: Non-Exported Activity (Implemented)

Concept

The simplest solution is to make the activity non-exported so it can only be accessed from within the app.

Pros	Cons
• Simple to implement • Enforced by Android system • Immediate protection against external intents	• Eliminates all external access • Lacks granular control

Implementation

Modify the AndroidManifest.xml to restrict external access:

```
<activity
    android:name=".DataBaseActivity"
    android:exported="false" />
```

2.3.2 Solution 2: Stack Inspection

Concept

Stack inspection [2, 4] prevents confused deputy attacks by examining the runtime call stack during privileged API calls. When a deputy makes an API call, the system checks the call stack to verify the caller's identity and permissions.

Implementation Approach

- Implement runtime checks during database operations
- Verify the call stack's integrity before executing sensitive actions
- Reject calls from unauthorized runtime contexts

Pros	Cons
• Dynamic runtime protection • Allows more granular access control • Can handle complex permission scenarios	• More complex to implement • Performance overhead • Requires careful runtime environment management

2.3.3 Solution 3: History-Based Access Control (HBAC)

Concept

HBAC [1] reduces permissions of trusted code after interactions with untrusted code, dynamically adjusting access levels based on the application's recent interaction history.

Implementation Strategy

- Track permission levels dynamically
- Reduce access privileges after interactions with potentially untrusted components
- Implement a sliding-scale permission model

Pros	Cons
• Adaptive security model • Provides context-aware access control • Mitigates potential permission re-delegation risks	• Significant implementation complexity • Potential performance impact • Requires sophisticated runtime tracking

Chapter 3

Leaky Content Providers

3.1 Usage of Content Providers

Content Providers in Android serve as a critical component for data management and sharing between applications. They offer a standardized interface for applications to access and modify data, whether it is stored in databases, files, or on the network. This mechanism is designed to maintain a balance between accessibility and security.

3.1.1 Purpose and Function

Content Providers encapsulate data and provide a safe mechanism to access it through well-defined interfaces. They act as an abstraction layer that handles the complexity of data operations, allowing clients to interact with the data through simple method calls rather than directly accessing storage mechanisms.

3.1.2 Role in the Notes App Scenario

In the current scenario involving the Notes application, a Content Provider is used for several specific reasons:

1. **Structured Data Access:** The Notes app likely uses a Content Provider to organize and provide structured access to notes and associated files.

2. **File Operations Abstraction:** The Content Provider implements the `openFile()` method to handle file operations, abstracting the complexities of file system access.

3. **Permission Management:** It leverages Android's permission system to control which applications can access the notes data, theoretically providing a secure channel for inter-application communication.

The Content Provider in this scenario was likely implemented to facilitate legitimate access to notes and their attachments, while maintaining a security boundary that prevents unauthorized access to files outside the app's intended scope.

3.2 Vulnerability Analysis

3.2.1 Path Traversal Vulnerability

The primary flaw in the Notes app's Content Provider implementation is a classic path traversal vulnerability. This vulnerability arises from insufficient validation of file paths when processing requests from client applications.

3.2.2 Why External Storage Access Is Possible

The vulnerability allows access to the SD-Card through the Content Provider due to several implementation flaws:

1. **Insufficient Path Validation:** The Content Provider does not properly validate or sanitize the file paths provided in URI requests. When a client app specifies a path with directory traversal sequences (e.g., `../`), the provider processes these sequences literally.

2. **Privilege Escalation:** The Notes app itself likely has permission to read external storage. When its Content Provider performs file operations without proper boundary checks, it effectively "lends" these permissions to any client application that calls it.

3. **Directory Traversal Execution:** When the provider receives a URI like `content://tcs.lbs.notes/../../../../../sdcard/ExternalTextFile.txt`, it follows the path outside its intended directory structure, allowing access to files in the SD-Card that would normally be protected by Android's permission system.

This vulnerability represents a significant security breach as it circumvents Android's permission model, allowing an application without explicit storage permissions to read files from external storage.

3.3 Implementation of the Exploit

The `queryContentProvider_onClicked` method exploits the path traversal vulnerability in the Notes app's Content Provider. Below is a detailed explanation of the implementation:

```
public void queryContentProvider_onClicked(android.view.
   View view) {
    try {
        // Get the file name from the EditText
        String fileName = queryEditText.getText().
           toString();

        // Create a content URI with path traversal
        Uri contentUri = Uri.parse("content://tcs.lbs.
           notes/../../../../../sdcard/" + fileName);

        // Get a ContentResolver to query the content
           provider
        ContentResolver contentResolver =
           getContentResolver();

        // Open an input stream to read the file
        InputStream inputStream = contentResolver.
           openInputStream(contentUri);

        // Read the contents of the file
        BufferedReader reader = new BufferedReader(new
           InputStreamReader(inputStream));
        StringBuilder stringBuilder = new StringBuilder()
           ;
        String line;
        while ((line = reader.readLine()) != null) {
            stringBuilder.append(line).append("\n");
        }

        // Close the streams
        reader.close();
        inputStream.close();

        // Display the contents in the TextView
        resultTextView.setText(stringBuilder.toString());

        Toast.makeText(this, "File␣read␣successfully!",
```

```
            Toast.LENGTH_SHORT).show();

    } catch (Exception e) {
        // Display error message
        resultTextView.setText("Error: " + e.getMessage()
            );
        Toast.makeText(this, "Error reading file: " + e.
            getMessage(),
                    Toast.LENGTH_LONG).show();
        e.printStackTrace();
    }
}
```

3.3.1 Exploitation Analysis

The method works through the following key steps:

1. **Target Identification:** The code constructs a URI targeting the Notes app's Content Provider (`content://tcs.lbs.notes.provider/`).
2. **Path Traversal Injection:** By prepending `../../` to the filename, the code forces the Content Provider to navigate up two directory levels, potentially reaching the root of the SD-Card.
3. **Content Resolution:** Using Android's `ContentResolver`, the method requests the file through the vulnerable provider.
4. **File Reading:** Upon successful connection, the method reads the file's contents through the returned `InputStream` and displays them in the UI.

The exploit succeeds because it leverages the Notes app's own permissions and lack of path validation to access files that would otherwise be protected from the exploiting application.

3.4 Security Solution

3.4.1 Vulnerability Remediation

To address the path traversal vulnerability in the Notes app's Content Provider, a comprehensive security fix is necessary. Here is the proposed implementation for the `openFile` method:

```
@Override
public ParcelFileDescriptor openFile(Uri uri, String mode
    ) throws FileNotFoundException {
    // Get the path from the URI
    String path = uri.getPath();

    // Security check 1: Reject paths with ".." to
       prevent directory traversal
    if (path.contains("..")) {
        throw new SecurityException("Path␣traversal␣
           attempt␣detected");
    }

    // Security check 2: Normalize path and ensure it's
       within bounds
    File requestedFile = new File(getContext().
       getFilesDir(), path);

    // Security check 3: Validate the normalized path is
       still within allowed directory
    String canonicalPath;
    try {
        canonicalPath = requestedFile.getCanonicalPath();
        String baseDirPath = getContext().getFilesDir().
           getCanonicalPath();

        if (!canonicalPath.startsWith(baseDirPath)) {
            throw new SecurityException("Attempted␣to␣
               access␣file␣outside␣permitted␣directory");
        }
    } catch (IOException e) {
        throw new SecurityException("Path␣resolution␣
           error", e);
    }

    // If we passed all security checks, proceed with
       file access
    int fileMode = ParcelFileDescriptor.MODE_READ_ONLY;
    if (mode.contains("w")) {
        fileMode = ParcelFileDescriptor.MODE_WRITE_ONLY;
    }
```

```
    return ParcelFileDescriptor.open(requestedFile,
        fileMode);
}
```

3.4.2 Security Controls Implemented

The security fix implements multiple layers of protection:

1. **Path Traversal Detection:** The method explicitly checks for directory traversal sequences (`..`) in the requested path and rejects such requests immediately.

2. **Path Normalization:** By resolving the file path to its canonical form, the method eliminates any obfuscation techniques that might hide traversal attempts.

3. **Boundary Enforcement:** The method verifies that the requested file's canonical path is within the application's private files directory, preventing access to unauthorized locations.

4. **Proper Exception Handling:** When security violations are detected, the method throws appropriate exceptions with meaningful error messages, improving both security and debugging.

3.4.3 Verification of the Solution

With these security measures in place, the previously successful exploit will fail because:

1. Any URI containing `../` sequences will be immediately rejected.

2. Even if alternative path obfuscation techniques are used, the canonical path resolution will reveal the true destination.

3. The boundary check ensures that only files within the Content Provider's designated directory can be accessed.

This implementation follows the principle of defense in depth, ensuring that even if one security check is somehow bypassed, others are in place to prevent unauthorized access.

3.5 Conclusion

The Leaky Content Provider vulnerability highlights the importance of proper path validation and boundary checking in Android applications, especially those that expose functionality through Content Providers. This case study demonstrates how seemingly minor implementation oversights can lead to significant security breaches, allowing unprivileged applications to bypass Android's permission system.

By implementing thorough input validation, path normalization, and boundary enforcement, developers can prevent path traversal attacks and ensure that Content Providers maintain the security boundaries they were designed to enforce.

This vulnerability underscores the importance of secure coding practices and the need for comprehensive security testing of inter-application communication channels in Android applications.

Chapter 4

Timing Attacks on Messengers

4.1 Location Inference Using Timing Information

This section reports on the timing side channel attack implementation for the messenger application. The attack demonstrates how message delivery timing can reveal users' geographical locations without requiring any explicit location permissions. Messaging applications rely on client-server architectures where:

1. A user sends a message to a central server
2. The server delivers the message to the recipient
3. Acknowledgments are sent back through the same path

Each step introduces timing delays that vary based on geographical distances and network infrastructure. These timing patterns can be exploited to infer user locations.

4.2 Implementation Approach

To extract timing information, we'll look at the logging in the `ConversationActivity.java` file. The key method for logging message events is in the *messageReceived* method:

```
Log.d("AppLog", "User:"+ USER_ID +"␣;␣Type:new_msg␣;␣
   Message:" + m.getText() + "␣;␣MessageID:"+ m.getID() +
    "␣;␣Time:" + m.getTime());
```

This logs three key message events:

1. `new_msg`: When a new message is sent
2. `server_ack`: When the server receives the message
3. `receiver_ack`: When the receiver receives the message

To implement an app or script to calculate time-to-server and time-to-receiver, we'll need to:

1. Extract the timing logs using ADB and Logcat
2. Parse the logs to calculate time differences
3. Map these times to potential locations using the provided tables

We implemented a Python script that analyzes messenger app logs to extract timing data and determine user locations. Our implementation works by:

1. **Extracting log data**: The script captures application logs using Android Debug Bridge (ADB) and filters for logs containing message-related events marked with the "AppLog" tag.
2. **Parsing message events**: We parse the log entries to identify three key event types: message sending (`new_msg`), server acknowledgment (`server_ack`), and recipient acknowledgment (`receiver_ack`). Each event contains a user ID, message ID, and timestamp.
3. **Organizing by message ID**: The script groups these events by message ID to track the complete lifecycle of each message.
4. **Calculating timing metrics**: For each message, we calculate two critical timing values:
    - **Time to server:** The delay between sending a message and receiving server acknowledgment
    - **Time to receiver:** The delay between server acknowledgment and recipient acknowledgment
5. **Aggregating by user**: We group these timing metrics by user to calculate average delays for each user.

This approach allows us to extract timing patterns that can be correlated with geographical locations without requiring direct access to location data or permissions.

4.3 Location Inference Results

By comparing observed delays with reference values from the provided tables, we successfully approximated the locations of all three users:

User ID	Server Delay	Region	Receiver Delay	City
001	699.64ms	Hammerfell	306.00ms	Elinhir
002	498.29ms	Valenwood	305.00ms	Arenthia
003	1015.79ms	Cyrodiil	201.93ms	Anvil

Table 4.1: Location inference results based on timing analysis

For all the users we sent the same messages in order to obtain accurate results for the timing measurements. Firstly, we sent 13 one-word messages followed by a paragraph long text. A similar method for testing was described in the paper of Schnitzler et al. in 2022 [3].
We also tested the timing attacks with random-length messages, but the approximations of the user locations haven't changed.

4.4 Challenges in Real-World Messengers

Applying this attack to real-world messengers like WhatsApp or Signal would face several significant challenges:

1. **Limited Log Access:** Commercial messengers don't expose detailed timing logs
2. **Network Variability:** Real-world networks introduce significant noise in timing measurements
3. **Advanced Infrastructure:** Commercial platforms use complex routing, load balancing, and content delivery networks that obscure simple geographical patterns
4. **Encrypted Metadata:** Modern secure messengers encrypt metadata, including timing information
5. **Message Batching:** Messages may be processed in batches, introducing artificial delays

These factors would require gathering much larger message samples and developing more sophisticated statistical models to overcome the noise and variability in real-world conditions.

4.5 Possible Mitigation Strategies

We propose the following strategies to mitigate timing side-channel attacks:

Strategy	**Advantages**	**Disadvantages**
Constant-time operations	Completely eliminates timing as an information channel	Degrades user experience with artificial delays
Random delays	Preserves average performance while obscuring patterns	Statistical analysis of many samples might still reveal patterns
Batched processing	Decouples network transmission time from processing	Affects real-time communication quality
Anonymous routing	Fundamentally breaks timing-location correlation	Increases complexity and infrastructure costs

Table 4.2: Comparison of mitigation strategies

The most balanced approach would combine:

- Small random delays to introduce noise without significantly affecting user experience
- Regional proxies to obscure message origins
- Limited timing precision in acknowledgment timestamps

4.6 Conclusion

Our implementation demonstrates that timing side-channel attacks can effectively reveal user locations without requiring any location permissions. This highlights the importance of considering side-channel attacks when designing secure messaging applications. While challenging to implement in real-world applications, the underlying principle remains valid and requires appropriate countermeasures to protect user privacy.

Chapter 5

Contributions

- **Alexandru Matcov:** Implemented and documented the tasks for Confused Deputy Attacks and Timing Attacks on Messengers. Also, helped implementing and documented the Leaky Content Providers.
- **Jakob Ström:** Worked on and documented the tasks for Intent Sniffing and Leaky Content Providers.

Bibliography

[1] Martin Abadi and Cédric Fournet. 2003. Access Control Based on Execution History.. In *NDSS*, Vol. 3. 107–121.

[2] Cedric Fournet and Andrew D Gordon. 2002. Stack inspection: Theory and variants. *ACM SIGPLAN Notices* 37, 1 (2002), 307–318.

[3] Theodor Schnitzler, Katharina Kohls, Evangelos Bitsikas, and Christina Pöpper. 2022. Hope of delivery: Extracting user locations from mobile instant messengers. *arXiv preprint arXiv:2210.10523* (2022).

[4] Dan S Wallach and Edward W Felten. 1998. Understanding Java stack inspection. In *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No. 98CB36186)*. IEEE, 52–63.