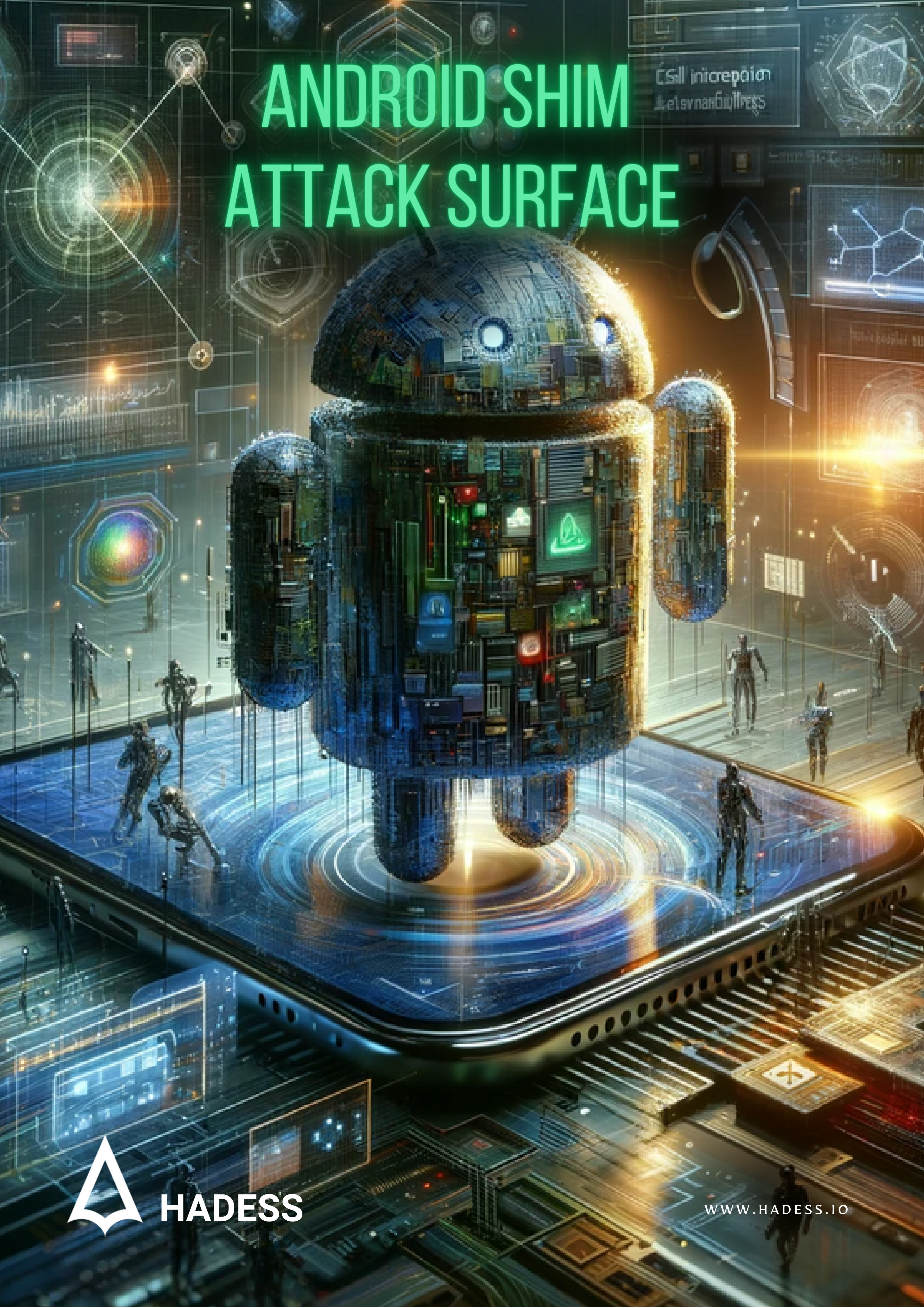
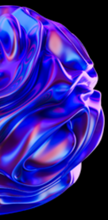
ANDROID SHIM ATTACK SURFACE



HADESS

WWW.HADESS.IO



INTRODUCTION

Android shims are small libraries that act as compatibility layers between different versions of the Android operating system. They allow developers to call newer Android APIs on older platform versions that don't natively support those APIs.

Shims provide backward compatibility by implementing newer APIs on top of older platform functionality. This enables apps built with the latest SDK to still run on older Android versions. Using shims, developers don't have to write custom code or conditional logic to support different API levels.

The Android compatibility package provides official shims for many framework APIs. For example, the v4 support library contains shims for fragments and loader APIs introduced in Android 3.0. This allows using fragments cleanly on Android 1.6 and higher.

Other popular shim libraries include AppCompat for the action bar API and design support library for material design components. These are developed by the Android team to help easily adopt new features and UI paradigms.

Shims are especially useful during Android transitions when many devices still run older OS versions. They prevent fragmentation by letting developers build with new APIs and distribute to all Android versions via Shim backwards compatibility. Overall, shims are an essential tool for Android developers to maximize platform reach.

DOCUMENT INFO

HADESS

To be the vanguard of cybersecurity, Hadess envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish Hadess as a symbol of trust, resilience, and retribution in the fight against cyber threats.

At Hadess, our mission is twofold: to unleash the power of white hat hacking in punishing black hat hackers and to fortify the digital defenses of our clients. We are committed to employing our elite team of expert cybersecurity professionals to identify, neutralize, and bring to justice those who seek to exploit vulnerabilities. Simultaneously, we provide comprehensive solutions and services to protect our client's digital assets, ensuring their resilience against cyber attacks. With an unwavering focus on integrity, innovation, and client satisfaction, we strive to be the guardian of trust and security in the digital realm.

Security Researcher
Negin Nourbakhsh
Eiad Mojo

TABLE OF CONTENT

Executive Summary

This article delves into the critical role of shims in the Android ecosystem, a platform known for its diversity and fragmentation. Shims, as intermediary layers, are instrumental in ensuring compatibility and functionality across various versions and configurations of Android.

The Necessity of Shims in Android's Fragmented Ecosystem Android's vast and fragmented landscape presents unique challenges, particularly in maintaining app compatibility across different versions and devices. Shims emerge as a vital solution to this problem, acting as a bridge between new app developments and legacy systems.

The Challenge of Legacy Support One of the primary challenges in Android development is supporting legacy systems. Shims provide a way to maintain app functionality on older devices without compromising the benefits of new Android features and updates.

Understanding the Shim - A Deep Dive The article offers an in-depth exploration of shims, covering their basic principles, architecture, and the mechanics of how they function within the Android framework. This section is crucial for developers seeking a comprehensive understanding of shims.

Function and Use Cases Shims serve multiple purposes in Android development. Their functions range from ensuring compatibility and enhancing security to facilitating debugging and testing processes. The article outlines various use cases, demonstrating the versatility of shims.

Intercepting Function and System Calls with Shims A significant aspect of shims is their ability to intercept function and system calls. This capability is crucial for modifying app behavior, debugging, and security testing. The article provides practical examples, including intercepting intents to start new activities, modifying outgoing SMS intents, and altering intent extras before an activity starts.

Role of Shims in Android Development Shims play a pivotal role in Android development. They are key to improving app compatibility across different Android versions, ensuring a consistent user experience, and reducing the complexities associated with Android's fragmentation.

Key Findings

- The Necessity of Shims
- Android's Fragmented Ecosystem
- The Challenge of Legacy Support
- Understanding the Shim - A Deep Dive
- Basic Principles and Architecture
- Function and Use Cases
    - Intercepting Function Calls
    - Intercepting System Calls with Shims
    - Intercepting Intent to Start a New Activity
    - Modifying Outgoing SMS Intent
    - Intercepting Broadcast Intents
    - Altering Intent Extras Before Activity Starts
    - Logging Received Intents in BroadcastReceiver
    - Modifying Intent Data Before Service Starts
    - Intercepting Intent Filters
- Role of Shims in Android Development
- Improving App Compatibility

Abstract

The Android operating system, characterized by its open-source nature and widespread adoption, faces significant security challenges due to its fragmented ecosystem. One such challenge is the attack surface presented by Android shims, intermediary software layers that facilitate compatibility between different versions of the OS and applications. This paper explores the vulnerabilities associated with these shims, highlighting how they can become potential gateways for security breaches.

Shims, while essential for maintaining application functionality across diverse Android versions, inadvertently introduce complexities in the system architecture. These complexities often lead to an expanded attack surface, where inconsistencies and loopholes in the shim layers can be exploited by malicious entities. The paper examines various scenarios where shims can be manipulated, including intercepting and altering system calls, modifying intent data, and unauthorized access to privileged system operations.

We conduct a thorough analysis of the shim architecture in Android, identifying key areas where vulnerabilities are most likely to occur. This includes a review of common practices in shim implementation, such as function hooking and intent interception, and their implications for system security. The paper also discusses real-world cases where shim vulnerabilities have been exploited, providing a practical perspective on the risks involved.

Furthermore, we propose a set of guidelines and best practices for developers and system architects to mitigate the risks associated with shims. This includes recommendations for secure coding practices, regular security audits, and the implementation of robust monitoring systems to detect and respond to potential security threats.

In conclusion, the paper emphasizes the need for a balanced approach in Android development, where the benefits of shims in terms of compatibility and functionality are weighed against the potential security risks they introduce. By shedding light on the vulnerabilities inherent in Android shims, this study aims to contribute to the development of more secure Android systems and applications.

HADESS.IO

Android Shim

I WISH I WAS MORE USER-FRIENDLY :|
AM I LOOSING MEMORY?
I DON'T WANNA GO ONLINE ANYMORE...
OHH... IS THIS A VIRUS?
WHAT IF I FREEZE AGAIN?
MAYBE I SHOULD UPDATE.
KEEP HAVING THOSE CRASH NIGHTMARES :/
I FEEL SPYWARES ALL AROUND ME.
TO REBOOT OR NOT TO REBOOT???
quotesgram.com

01

Attacks

The Necessity of Shims

Android Shims were introduced in Android 4.1 (Jelly Bean) as a way to address the problem of API fragmentation. At the time, there were a large number of different versions of Android in use, and each version had its own set of APIs. This made it difficult for developers to write code that worked on a wide range of devices.
Shims provided a solution to this problem by providing a consistent API surface across different versions of Android. This allowed developers to write code that worked on a wide range of devices, without having to target specific versions of Android.

The need for shims has become even more important in recent years, as the number of different versions of Android in use has continued to grow. In addition, the pace of innovation in the Android ecosystem has accelerated, and new features are being introduced all the time. This means that it is becoming increasingly difficult for developers to keep up with the latest changes and to write code that is compatible with a wide range of devices.

Shims can help developers to address this challenge by providing backported implementations of new features. This means that developers can start using new features in their apps, even if they are not available on all devices.

Examples of the Necessity for Shims:

1. Material Design: Material Design was introduced in Android 5.0 (Lollipop), but it is not available on older versions of Android. AppCompat provides a backported implementation of Material Design so that developers can start using Material Design in their apps, even if they are not targeting Android 5.0 or higher.
2. RecyclerView: RecyclerView is a widget that was introduced in Android 5.0 (Lollipop). It is a more efficient and powerful replacement for the ListView widget. However, RecyclerView is not available on older versions of Android. AppCompat provides a backported implementation of RecyclerView so that developers can start using

RecyclerView in their apps, even if they are not targeting Android 5.0 or higher.

1. Biometric authentication: Biometric authentication features, such as fingerprint scanning and facial recognition, were introduced in Android 6.0 (Marshmallow). However, biometric authentication features are not available on all devices. BiometricPrompt provides a backported implementation of biometric authentication features so that developers can start using biometric authentication in their apps, even if they are not targeting Android 6.0 or higher.

Android's Fragmented Ecosystem

Android is highlighted by its massive install base across a diverse range of devices from various OEMs. However, this expansive reach comes with the downside of significant fragmentation across different Android platform versions.
When new versions of Android launch, they introduce attractive updated capabilities and APIs for developers. However, these new Android releases only reach a small subset of recently launched high-end devices initially. The vast majority of active Android devices run older OS releases that lack the latest APIs.

For example, when Android 7.0 Nougat launched in 2016, nearly 84% of Android devices were still on Android 6.0 or older. Only a fraction were running the latest Nougat release with new features like multi-window support and Direct Reply notifications.

This fragmentation is exacerbated by the diversity of Android device types, including phones, tablets, watches, TVs, and more. Each form factor sees different adoption rates of new OS versions depending on hardware capabilities and OEM update support.

The open Android ecosystem also leads to customizations and modifications by silicon vendors, OEMs, and carriers. This results in multiple flavors of Android releases rather than a unified OS experience across devices.
In summary, Android's exponential growth has led to acute fragmentation across devices, form factors, and OS versions. This poses challenges for app developers aiming to leverage the latest Android capabilities while retaining broad compatibility across the fragmented landscape. Addressing this fragmentation is an ongoing area of focus.

Opportunities of Android Fragmentation

1.it allows for a greater diversity of devices and features in the Android ecosystem. Device manufacturers can differentiate their products by using different versions of Android and by adding their own custom features. This gives consumers more choice when selecting an Android device.

2.Android fragmentation can also lead to a more competitive app market. Developers are constantly competing to create the best possible apps for the Android platform. This competition can lead to lower prices and higher-quality apps for consumers.
3.Android fragmentation can also create opportunities for developers to specialize in specific versions of Android or specific types of devices. For example, a developer could focus on developing apps for older Android devices or for devices with low-end hardware. This can allow developers to build a niche expertise and to create apps that are tailored to the specific needs of their target market.

The Challenge of Legacy Support

Legacy support is major challenge. Android's extensive fragmentation poses immense challenges for app developers to provide legacy support. When new Android versions launch with attractive updated features, developers aim to adopt them in apps quickly. However, a large portion of active devices still run older Android releases lacking these capabilities.

For example, when material design launched with Android 5.0 in 2014, developers were eager to build apps with the new UI paradigm. But over 75% of devices at the time used Android 4.4 or lower which did not include native material widgets and themes. another example, when Android 4.0 Ice Cream Sandwich launched in 2011, nearly 70% of active Android devices were still on Android 2.3 Gingerbread.

Developers wanting to leverage new ICS APIs like fragments had to find workarounds to support the large Gingerbread user base.The situation further escalated due to Android's presence across multiple device form factors. Smartphones, tablets, watches, TVs and more meant fragmented OS adoption across device segments. A new Android TV running the latest API level would need apps built for older Android phone versions.

1.it can be difficult to write apps that are compatible with both old and new versions of Android. Android APIs change over time, so developers have to be careful to use APIs that are compatible with all of the versions of Android that they want to support.
2.legacy support can be time-consuming and expensive. Developers have to test their apps on a variety of devices and Android versions to ensure compatibility. This can be a significant investment of time and resources.
3.legacy support can also hinder innovation. Developers are less likely to adopt new Android APIs if they know that their apps will not be compatible with a large number of devices. This can slow down the development of new and innovative features for the Android platform.

How to Address the Challenge of Legacy Support

There are a few things that developers can do to address the challenge of legacy support:

1.Use shims and other tools. Shims are libraries that provide backported implementations of new Android APIs. This allows developers to use new APIs without having to worry about compatibility with older devices.
The shim approach substantially eased these challenges by consolidating legacy support. Well-designed shim libraries with consistent APIs meant developers could focus on building features using the latest capabilities. The shim would internally map calls to appropriate legacy implementations.

Thanks to shims like the AppCompat library, developers reduced complexity of cross-platform legacy support. Apps could easily adopt new Android capabilities while retaining compatibility with older Android versions still active in the fragmented ecosystem.

2.Target a specific range of Android versions. Developers can choose to target a specific range of Android versions when developing their apps. This can make it easier to ensure compatibility and to avoid having to support a wide range of Android versions.

3.Use a modular design. Developers can design their apps in a modular way, with different modules for different features. This allows developers to update individual modules without having to update the entire app.

4.Use a cloud-based backend. Developers can use a cloud-based backend to store data and to provide functionality for their apps. This can help to reduce the amount of code that needs to be maintained on each device.

Understanding the Shim - A Deep Dive

Android shims are specialized compatibility libraries that wrap newer framework APIs to make them available on older platform versions. This is accomplished through various techniques:
Translation layers - The shim translates calls to newer APIs into equivalent calls to legacy APIs to reproduce the expected behavior. For example, the v4 support library added a FragmentCompat translation layer to cast fragment API calls to the platform-specific implementation.

Fallback logic - The shim contains custom fallback code to mimic missing functionality on older platforms. For instance, AppCompat replicates material design widgets like CardView when running on pre-Lollipop devices.
Polyfills - Where feasible, the shim polyfills missing API capabilities directly. The design support library brings ripple animation to pre-Lollipop devices via a polyfill.

Resource mapping - Resources added in newer APIs like themes and drawables are mapped to suitable defaults on older versions. This maintains visual consistency.

Gradual degradation - When a capability cannot be polyfilled, the shim gracefully degrades the experience. For example, RecyclerView on older platforms loses animations but retains core functionality.
Shims aim to balance compatibility with minimizing overhead. They focus on reproducing essential API contracts and behaviors while dropping advanced features when needed. Well-designed shims feel like a natural extension of the framework.

In summary, Android shims leverage various techniques like translation layers, polyfills, and fallback logic to bring newer APIs to older Android versions. They enable writing apps against the latest capabilities while retaining broad device support.

To understand the process of connecting a proximity driver to its Hardware Abstraction Layer (HAL) using a shim, let's break down the steps and the associated code. The example provided is specific to the Acme Proximity Sensor, but the principles can be applied to similar scenarios.

- Shim Implementation: The shim acts as a bridge between the HAL and the specific device driver, in this case, the Acme Proximity Sensor. It implements the device methods open_sensor, close_sensor, and poll_sensor.
- Shim Interface Definition: Located in proximity/include/dev/proximity_sensor.h, this file defines the API for the shim.
- Purpose of the Shim: The shim encapsulates the details of the specific device, making it unnecessary for the HAL user code to interact directly with the device.

```
#ifndef ACME_PROXIMITY_SENSOR_H
#define ACME_PROXIMITY_SENSOR_H
#include "proximity_hal.h"
int open_sensor(proximity_params_t &params);
int poll_sensor(int fd, int precision);
int close_sensor(int fd);
#endif // ACME_PROXIMITY_SENSOR_H
```

Explanation of the Shim Functions
open_sensor(proximity_params_t &params): Initializes the sensor with the given parameters. The params struct contains configuration details like precision bounds.
poll_sensor(int fd, int precision): Retrieves the current proximity value from the sensor. It requires the file descriptor fd and a precision level within specified bounds.
close_sensor(int fd): Shuts down the sensor, identified by the file descriptor fd, to optimize battery usage.

Contextual Notes
User Space vs. Kernel Space: The HAL and shim code run in user space, not as part of the kernel. This is important for understanding the security and performance implications.
Device Driver Interaction: Although the device driver might run partially in the kernel, the HAL and shim do not interact directly with the kernel.
Encapsulation: The shim is designed to encapsulate all the device-specific details, ensuring that the HAL remains generic and reusable for different types of proximity sensors.

Code Usage
HAL Interaction: The HAL will use these shim functions to interact with the Acme Proximity Sensor. The HAL itself does not need to know the specifics of sensor operation, as these are handled by the shim.
Device-Specific Logic: Any logic specific to the Acme Proximity Sensor, such as handling specific hardware quirks or optimizations, should be implemented within the shim.

Implementing a Java application that interacts with a proximity sensor through a Hardware Abstraction Layer (HAL) involves creating a Java-native interface. This interface, or shim, allows Java code to communicate with the native code that directly interacts with the sensor. The AcmeProximitySensor class in Java serves as this interface.

Overview of AcmeProximitySensor Class
The AcmeProximitySensor class in Java acts as a bridge to the native environment. It abstracts the details of the native implementation, providing a clean, Java-friendly API. This class implements AutoCloseable to ensure proper resource management.
Static Initializer Block: Loads the native library containing the implementation for sensor interaction.

```
static {
    System.loadLibrary("acmeproximityjni");
}
```

Native Peer Reference: The peer variable holds a native reference to the HAL object. It's crucial to treat this reference as opaque in Java.

```
private long peer;
```

Initialization Method (init): Initializes the sensor and sets up the peer reference.

```
public void init() throws IOException {
    synchronized (this) {
        if (peer != 0L) {
            return;
        }
        peer = open();
        if (peer == 0L) {
            throw new IOException("Failed to open proximity sensor");
        }
    }
}
```

Poll Method (poll): Interacts with the sensor to get proximity data.

```
public int poll(int precision) throws IOException {
    synchronized (this) {
        if (peer == 0L) {
            throw new IOException("Device not open");
        }
        return poll(peer, precision);
    }
}
```

Close Method (close): Closes the sensor and cleans up resources.

```
@Override
public void close() throws IOException {
    final long hdl;
    synchronized (this) {
        hdl = peer;
        peer = 0L;
    }
    if (hdl == 0L) {
        return;
    }
    if (close(hdl) < 0) {
        throw new IOException("Failed closing proximity sensor");
    }
}
```

Finalizer: Ensures that resources are freed if the object is garbage collected.

```
@Override
protected void finalize() throws Throwable {
    try {
        close();
    } finally {
        super.finalize();
    }
}
```

Native Methods: These are the native methods that interact with the sensor hardware.

```
private static native long open();
private static native int poll(long handle, int precision);
private static native int close(long handle);
```

Library Loading: The System.loadLibrary method is used to load the native library. The library name is transformed according to the platform's conventions (e.g., libacmeproximityjni.so on Linux).
Native Reference Management: The peer variable is crucial for maintaining a reference to the native object. It's important to ensure that this reference is not altered by the Java code.

Resource Management: Implementing AutoCloseable and providing a finalize method ensures that the sensor is properly closed and resources are freed, even if the client code neglects to call close.
Error Handling: The methods throw IOException to signal failures in sensor operations, adhering to Java's exception handling conventions.

The approach for creating a Java Native Interface (JNI) shim involves enhancing the opacity of the native reference in the Java variable. This is achieved by using reflection in the native code to set the value of the reference held in the Java variable. This technique ensures that the Java code cannot directly manipulate the native reference, thereby maintaining the integrity and safety of the system.

Native JNI Implementation
The native JNI function Java_com_acme_device_proximity_AcmeProximitySensor_open is responsible for opening the proximity sensor and setting the peer field in the Java object.

```
JNIEXPORT int JNICALL Java_com_acme_device_proximity_AcmeProximitySensor_open
(JNIEnv *env, jclass klass, jobject instance) {
    // Get the proximity sensor module
    if (hw_get_module(ACME_PROXIMITY_SENSOR_MODULE, &module)) return -1;

    hw_device_t *device;
    if (module->methods->open(module, nullptr, &device)) return -2;

    // Get the field ID for the 'peer' field in the Java object
    jfieldID peer = env->GetFieldID(klass, "peer", "J");
    if (!peer) return -3;

    // Set the 'peer' field in the Java object to the native device reference
    env->SetLongField(instance, peer, reinterpret_cast<jlong>(device));
    return 0;
}
```

Java Implementation
The AcmeProximitySensor class in Java is designed to interact with the native code.

```
public class AcmeProximitySensor implements AutoCloseable {
    // Native reference to the HAL object
    private long peer;

    public void init() throws IOException {
        synchronized (this) {
            int status = open(this);
            if (status != 0) {
                throw new IOException("Failed to open proximity sensor: " + status);
            }
        }
    }

    // Other methods like poll and close would be here

    private static native int open(AcmeProximitySensor instance);
    // Other native methods
}
```

Key Points
JNI Methods: GetFieldID and SetLongField are used to interact with the Java object's fields from native code. GetFieldID obtains a reference to a field, and SetLongField sets its value.

Instance vs. Static Native Methods: The difference lies in the second argument: a class reference for static methods and an instance reference for instance methods. In this case, a static method is used, but the instance is passed explicitly to have access to both the class and the instance.

Enhanced Opacity: By setting the peer field in the native code, any direct use of this field in the Java code becomes an error, ensuring that the native reference remains opaque and secure.

Error Handling: The open method returns a status code, making it clear and unambiguous for error handling in the Java code.

The lifecycle of a companion object in a Java Native Interface (JNI) context highlights the challenges and best practices in handling native resources in Java. The AcmeProximitySensor example illustrates these concepts well.

Managing Native Resources in Java
Explicit Management: The preferred method for managing native resources in Java is explicit management, typically through the Closeable or AutoCloseable interfaces. These interfaces signal to the user that the object holds resources that need to be released explicitly.
AutoCloseable Interface: This interface, introduced in Java 7, is preferable as it allows for more flexible exception handling and is compatible with the try-with-resources statement, which ensures that resources are automatically closed after use.

```
public class AcmeProximitySensor implements AutoCloseable {
    // ... other methods ...

    @Override
    public void close() throws Exception {
        // Code to free native resources
    }
}
```

Using try-with-resources:

```
try (AcmeProximitySensor sensor = new AcmeProximitySensor()) {
    // Use the sensor
} // The sensor is automatically closed here
```

Finalization as a Fallback: While explicit management is preferred, Java provides a mechanism called finalization to handle resource cleanup if an object is garbage collected without being explicitly closed.

Finalizer Method: The finalize() method is called by the garbage collector just before the object's memory is freed. This can be used as a safety net to clean up native resources.

```
@Override
protected void finalize() throws Throwable {
    try {
        close(); // Attempt to free resources
    } finally {
        super.finalize();
    }
}
```

Challenges with Finalization

1. Unpredictability: There is no guarantee about the order in which objects are finalized. This can lead to issues like NullPointerException if other objects have already been finalized.
2. Performance Impact: Finalizers can significantly slow down the garbage collector because they add complexity to the garbage collection process.
3. Memory Management Issues: There's no guarantee on the timing of when the garbage collector will run. This can lead to a buildup of objects waiting to be finalized, potentially causing memory issues, especially with large native resources.
4. Single Thread Execution: Finalizers are typically run on a single thread, which can lead to a backlog if objects are created and discarded at a high rate.

The use of reference queues in Java, offers a sophisticated method for managing the lifecycle of native objects, addressing many of the issues associated with finalizers. This approach is particularly relevant with the deprecation of finalizers in Java 9 in favor of Cleaners. Although Cleaners are not available in Android, the underlying technologies of PhantomReferences and ReferenceQueues are, providing a viable alternative.

Overview of Reference Queues in Java

Reference queues, combined with phantom references, provide a controlled way to perform cleanup actions when objects become eligible for garbage collection. This method does not interfere with the garbage collection process and allows for the ordered freeing of objects.

Implementation Details

- Reference Queue with Phantom References:A PhantomReference is created with a reference to an object and a reference queue.
- When the object becomes eligible for garbage collection, the phantom reference is enqueued in the reference queue.
- Calls to PhantomReference.get() always return null, ensuring the referenced object is not reachable through the phantom reference.
- Managing the Lifecycle of AcmeProximitySensor:The getSensor static method acts as a factory for AcmeProximitySensor instances.
- It creates a native companion object, a Java instance, and a SensorCleaner for cleanup.
- The SensorCleaner is stored in a map to prevent it from being garbage collected prematurely.
- Cleanup Process:The cleanup method is responsible for freeing the native companion object and removing the SensorCleaner from the map.
- It ensures that the native close method is called only once.

```
public class AcmeProximitySensor implements AutoCloseable {
    private final AtomicLong peerRef;

    private AcmeProximitySensor(AtomicLong peerRef) {
        this.peerRef = peerRef;
    }

    public static AcmeProximitySensor getSensor() {
        synchronized (CLEANERS) {
            final AtomicLong peerRef = new AtomicLong(open());
            final AcmeProximitySensor sensor = new AcmeProximitySensor(peerRef);
            CLEANERS.put(peerRef, new SensorCleaner(peerRef, sensor));
            return sensor;
        }
    }

    @Override
    public void close() {
        cleanup(peerRef);
    }

    private static void cleanup(AtomicLong peerRef) {
        // Implementation of cleanup logic
    }

    // Native methods and other Java methods
}
```

Advantages and Challenges

- Advantages:More control over the cleanup process.
- Ability to scale the cleanup process according to application needs.
- Avoids overwhelming the finalizer thread, a common issue with finalizers.
- Challenges:Requires additional infrastructure to schedule and execute cleanup tasks.
- The indeterminate scheduling of object cleanup remains, as objects are enqueued only when the garbage collector runs.

Basic Principles and Architecture

Compatibility Shims adhere to strict backwards compatibility requirements to match native API behavior.

- APIs must function identically to the original implementation. For example, FragmentCompat perfectly reproduces native Fragment APIs.
- Contracts around lifecycles, callbacks, data management etc. are identically replicated. Fragments in v4 support library integrate seamlessly.
- Edge cases and quirks in original APIs are reproduced to avoid breaking client code. Compat must equal Native.
- Performance
- Shims optimize for minimal overhead over native implementations.
- Lightweight proxy implementations avoid unnecessary allocations and overhead.
- Expensive operations like inflating custom widgets are deferred or lazily initialized.
- Performance tuning reuse pools, buffers, and loop constructs to minimize memory and CPU impact.

Encapsulation
Shim internals are fully encapsulated from business logic code.
- - Business logic relies solely on shim-exposed public APIs and remains platform agnostic.
- - Encapsulation prevents shim implementation details from leaking into client code.
- - Internal shim callbacks and handlers are anonymous classes rather than separate classes to avoid exposure.

Degradation
Shims gracefully degrade capabilities rather than rigidly fail.
- - Missing features are dropped from shim implementations while retaining core required functionality.
- - Default resources are supplied when custom ones are unavailable. For example, default styles on older platforms.
- - Degradation handled elegantly so apps remain usable even if missing certain features.
- Isolation

Shims do not modify system code and remain fully self-contained.
- - Shim libraries are distributed separately from OS frameworks to prevent interference.
- - No injection into base Android code to limit side effects. Changes are fully isolated.
- - Allows evolving shim independently and applying bug fixes without impacting the framework.

This combination of rigid compatibility, performant implementation, graceful degradation, and isolation makes Shims powerful compatibility tools.

Function and Use Cases

- - Accessing Newer APIs - The core purpose of shims is allowing developers to access newer APIs on older platform versions that lack native support. For example, using fragments on pre-Honeycomb devices via the v4 support library.
- - Material Design Adoption - Components like AppCompat and the support design library allow easy material design adoption on pre-Lollipop Android versions. This brings material widgets and themes to older devices.
- - Consistent UI Features - Shim libraries provide consistent UI features across versions like action bar and navigation UI that otherwise vary across API levels.
- - Fallback Behaviors - Shims gracefully fallback when native functionality is unavailable. For example, RecyclerView defaults to basic list view behavior on older platforms lacking animation capabilities.
- - Polyfills - At times shims directly polyfill missing functionality such as the RippleDrawable backport for ripple animation effects on older versions.
- - Resource Mapping - Resources like styles and drawables can be intelligently mapped to suitable defaults on older platform versions when custom ones are not available.
- - OEM Extension - OEMs can leverage shims to bring proprietary capabilities to AOSP-based builds. For example, Samsung's shim layers for Galaxy-specific features.

Overall, shims aim to smooth out API and capability differences across Android versions. They enable building robust apps that leverage the latest framework innovations while retaining wide device support.

An "Android shim attack surface" refers to the potential vulnerabilities in the layer (shim) that interfaces between higher-level Android code and lower-level system or hardware functionalities. This shim layer can be a target for attackers seeking to exploit the communication between different layers of the system. Understanding and securing this attack surface is crucial for maintaining the integrity and security of an Android application.

Identifying the Attack Surface

1. Intercepting System Calls: Shims often involve system calls or interactions with the operating system. Monitoring these calls can reveal potential vulnerabilities.
2. Analyzing Permissions: Shims might require elevated permissions to interact with hardware or system resources. Overly broad permissions can be a security risk.
3. Inspecting Data Handling: How data is passed to and from the shim can expose vulnerabilities, especially if the data is not properly validated or sanitized.
4. Reviewing Third-Party Libraries: If the shim uses external libraries, these can be a source of vulnerabilities.

Using Frida for Security Analysis

Frida is a dynamic instrumentation toolkit that allows you to inject your own scripts into black box processes. It's widely used for security testing, especially in mobile environments like Android.

Example: Intercepting Function Calls

Suppose you want to monitor the behavior of a function within the shim layer. You can use Frida to hook into this function and log its arguments and return values.

Step 1: Set Up Frida

- Ensure Frida is installed on your machine and the Frida server is running on your Android device.

Step 2: Write a Frida Script

Create a JavaScript file (e.g., hook.js) with the following content:

```
Java.perform(function () {
    var TargetClass = Java.use("com.example.shim.TargetFunctionClass");

    TargetClass.targetFunction.implementation = function (arg) {
        console.log("TargetFunction called with argument: " + arg);
        var retVal = this.targetFunction(arg);
        console.log("TargetFunction returned: " + retVal);
        return retVal;
    };
});
```

This script hooks into TargetFunction of TargetFunctionClass, logs its argument and return value, and then proceeds with the original implementation.

Step 3: Run the Frida Script

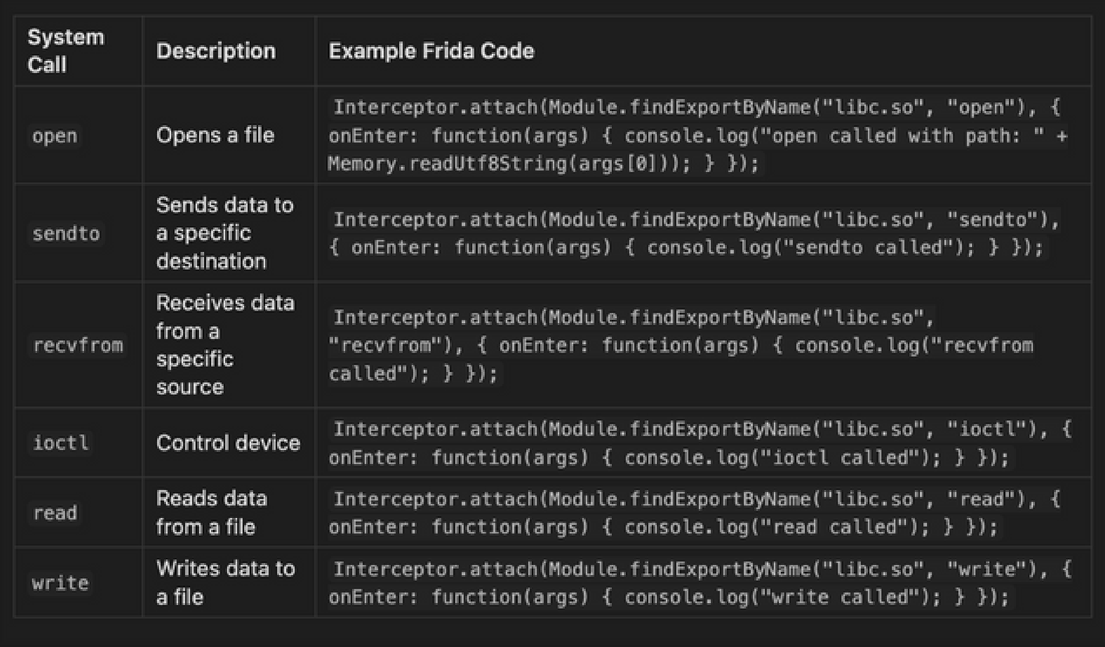
Run the script with Frida:

```
frida -U -l hook.js -f com.example.yourapp --no-pause
```

This command starts your app (com.example.yourapp), injects the hook.js script, and attaches Frida to it without pausing the app.

Intercepting System Calls with Shims

To intercept system calls, you can use Frida, a dynamic instrumentation toolkit. Below is a table outlining some important Android system calls that can be intercepted, along with example Frida code snippets for each.



System Call	Description	Example Frida Code
`open`	Opens a file	`Interceptor.attach(Module.findExportByName("libc.so", "open"), { onEnter: function(args) { console.log("open called with path: " + Memory.readUtf8String(args[0])); } });`
`sendto`	Sends data to a specific destination	`Interceptor.attach(Module.findExportByName("libc.so", "sendto"), { onEnter: function(args) { console.log("sendto called"); } });`
`recvfrom`	Receives data from a specific source	`Interceptor.attach(Module.findExportByName("libc.so", "recvfrom"), { onEnter: function(args) { console.log("recvfrom called"); } });`
`ioctl`	Control device	`Interceptor.attach(Module.findExportByName("libc.so", "ioctl"), { onEnter: function(args) { console.log("ioctl called"); } });`
`read`	Reads data from a file	`Interceptor.attach(Module.findExportByName("libc.so", "read"), { onEnter: function(args) { console.log("read called"); } });`
`write`	Writes data to a file	`Interceptor.attach(Module.findExportByName("libc.so", "write"), { onEnter: function(args) { console.log("write called"); } });`

- Set Up Frida:Install Frida on your machine and the Frida server on the Android device.
- Ensure the target app is installed on the device.
- Write a Frida Script:Use the example code snippets in the table as a starting point.
- Save the script in a file (e.g., hook.js).
- Run the Frida Script:Execute the script with Frida: frida -U -l hook.js -f com.example.yourapp --no-pause.

Manipulating shims for sending and receiving intents in Android can be achieved through dynamic instrumentation tools like Frida or Xposed Framework. Here, I'll provide 10 practical scenarios where you can manipulate intents, along with example code snippets for both Frida and Xposed.

Scenario 1: Intercepting Intent to Start a New Activity

```
Frida:

Java.perform(function () {
    var Activity = Java.use("android.app.Activity");
    Activity.startActivity.overload('android.content.Intent').implementation = function (intent) {
        console.log("Starting activity with intent: " + intent.toString());
        this.startActivity(intent);
    };
});


Xposed:

XposedHelpers.findAndHookMethod("android.app.Activity", lpparam.classLoader, "startActivity", Intent.class, new
XC_MethodHook() {
    @Override
    protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
        Intent intent = (Intent) param.args[0];
        XposedBridge.log("Starting activity with intent: " + intent.toString());
    }
});
```

Scenario 2: Modifying Outgoing SMS Intent

```
Frida:

Java.perform(function () {
    var SmsManager = Java.use("android.telephony.SmsManager");
            SmsManager.sendTextMessage.overload('java.lang.String',   'java.lang.String',   'java.lang.String',
'android.app.PendingIntent',   'android.app.PendingIntent').implementation   =   function   (destinationAddress,
scAddress, text, sentIntent, deliveryIntent) {
        console.log("Sending SMS to: " + destinationAddress + " with text: " + text);
        text = "Modified: " + text; // Modify the SMS text
        this.sendTextMessage(destinationAddress, scAddress, text, sentIntent, deliveryIntent);
    };
});


Xposed:

XposedHelpers.findAndHookMethod("android.telephony.SmsManager",     lpparam.classLoader,     "sendTextMessage",
String.class, String.class, String.class, PendingIntent.class, PendingIntent.class, new XC_MethodHook() {
    @Override
    protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
        String text = (String) param.args[2];
        XposedBridge.log("Sending SMS with text: " + text);
        param.args[2] = "Modified: " + text; // Modify the SMS text
    }
});
```

Scenario 3: Intercepting Broadcast Intents

```
Frida:

Java.perform(function () {
    var Context = Java.use("android.content.Context");
    Context.sendBroadcast.overload('android.content.Intent').implementation = function (intent) {
        console.log("Broadcasting intent: " + intent.toString());
        this.sendBroadcast(intent);
    };
});


Xposed:

XposedHelpers.findAndHookMethod("android.content.ContextWrapper",     lpparam.classLoader,     "sendBroadcast",
Intent.class, new XC_MethodHook() {
    @Override
    protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
        Intent intent = (Intent) param.args[0];
        XposedBridge.log("Broadcasting intent: " + intent.toString());
    }
});
```

Scenario 4: Altering Intent Extras Before Activity Starts

```
Frida:

Java.perform(function () {
    var Activity = Java.use("android.app.Activity");
    Activity.startActivity.overload('android.content.Intent').implementation = function (intent) {
        var extras = intent.getExtras();
        if (extras != null) {
            extras.putString("extra_key", "modified_value");
        }
        this.startActivity(intent);
    };
});



Xposed:

XposedHelpers.findAndHookMethod("android.app.Activity", lpparam.classLoader, "startActivity", Intent.class, new
XC_MethodHook() {
    @Override
    protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
        Intent intent = (Intent) param.args[0];
        intent.putExtra("extra_key", "modified_value");
    }
});
```

Scenario 5: Logging Received Intents in BroadcastReceiver

```
Frida:

Java.perform(function () {
    var BroadcastReceiver = Java.use("android.content.BroadcastReceiver");
    BroadcastReceiver.onReceive.implementation = function (context, intent) {
        console.log("Received intent: " + intent.toString());
        this.onReceive(context, intent);
    };
});


Xposed:

XposedHelpers.findAndHookMethod("android.content.BroadcastReceiver",      lpparam.classLoader,      "onReceive",
Context.class, Intent.class, new XC_MethodHook() {
    @Override
    protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
        Intent intent = (Intent) param.args[1];
        XposedBridge.log("Received intent: " + intent.toString());
    }
});
```

Scenario 6: Intercepting Intent Results

```
Frida:

Java.perform(function () {
    var Activity = Java.use("android.app.Activity");
        Activity.onActivityResult.overload('int',  'int',  'android.content.Intent').implementation  =  function
(requestCode, resultCode, data) {
         console.log("Activity result with requestCode: " + requestCode + ", resultCode: " + resultCode + ",
data: " + data);
        this.onActivityResult(requestCode, resultCode, data);
    };
});



Xposed:

XposedHelpers.findAndHookMethod("android.app.Activity",  lpparam.classLoader,  "onActivityResult",  int.class,
int.class, Intent.class, new XC_MethodHook() {
    @Override
    protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
        Intent data = (Intent) param.args[2];
        XposedBridge.log("Activity result with data: " + data);
    }
});
```

Scenario 7: Modifying Intent Data Before Service Starts

```
Frida:

Java.perform(function () {
    var Context = Java.use("android.content.Context");
    Context.startService.overload('android.content.Intent').implementation = function (intent) {
        console.log("Starting service with intent: " + intent.toString());
        // Modify intent as needed
        return this.startService(intent);
    };
});


Xposed:

XposedHelpers.findAndHookMethod("android.content.ContextWrapper",     lpparam.classLoader,     "startService",
Intent.class, new XC_MethodHook() {
    @Override
    protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
        Intent intent = (Intent) param.args[0];
        XposedBridge.log("Starting service with intent: " + intent.toString());
        // Modify intent as needed
    }
});
```

Scenario 8: Intercepting Intent Filters

```
Frida:

Java.perform(function () {
    var IntentFilter = Java.use("android.content.IntentFilter");
    IntentFilter.addAction.overload('java.lang.String').implementation = function (action) {
        console.log("IntentFilter adding action: " + action);
        this.addAction(action);
    };
});
```

The concept of a "native shim" for loading JNI (Java Native Interface) files in Android for active debugging is an advanced technique used in reverse engineering and security analysis. The provided code is an example of such a shim, designed to load a native library and execute its JNI_OnLoad function, which is typically called when a library is first loaded by the Android runtime. This approach allows a debugger to hook into and analyze the behavior of native code that would normally be invoked from Java (Dalvik/ART) context.
Let's discuss how this native shim works and then explore 10 practical scenarios where it can be applied.

Understanding the Native Shim Code

1. Library Loading: The shim uses dlopen to dynamically load a specified native library. This is useful for libraries that are loaded at runtime by an Android application.
2. JNI_OnLoad Invocation: After loading the library, it looks for the JNI_OnLoad function, which is a standard entry point for JNI libraries. It then calls this function, initializing the library in a controlled environment.
3. JavaVM Initialization: The shim initializes a JavaVM instance, which is necessary for JNI functions to interact with Java objects and the Android runtime.
4. Environment Setup: It sets up the necessary environment for the library to run, including class paths and library paths.

Practical Scenarios for Using the Native Shim

1. Debugging JNI Library Initialization: Analyze how a JNI library initializes and interacts with the JavaVM during its startup phase.
2. Reverse Engineering Native Libraries: Understand the behavior of native code within Android applications, especially those that don't have source code available.
3. Analyzing Security Vulnerabilities: Check for potential security vulnerabilities in the way a JNI library handles input from Java code.
4. Testing JNI Function Calls: Debug and test individual JNI functions by invoking them directly from the shim.
5. Manipulating JNI Environment: Modify the JavaVM and JNI environment to observe how the native library behaves under different conditions.
6. Hooking JNI Functions: Use the shim to hook JNI functions and monitor their inputs and outputs for analysis.
7. Simulating Android Runtime for Libraries: Simulate the Android runtime environment for libraries that expect certain Android-specific behaviors or contexts.
8. Bypassing Java Layer for Testing: Directly test native libraries without the need to go through the Java layer, speeding up the debugging process.
9. Dynamic Analysis of Native Code: Perform dynamic analysis by attaching tools like GDB or IDA Pro to the shim process.
10. Exploring Library Dependencies: Investigate how a native library interacts with other system libraries or Android components.

Commands for Using the Native Shim
To use the native shim, you would typically compile the C code into an executable and then run it on an Android device or emulator. Here's a general outline of the steps:
Compile the Shim:

```
arm-linux-androideabi-gcc -o native-shim native-shim.c -ldl
```

Push the Shim and Library to the Device:

```
adb push native-shim /data/local/tmp/
adb push your-native-library.so /data/local/tmp/
```

Set Executable Permissions:

```
adb shell chmod +x /data/local/tmp/native-shim
```

Set Executable Permissions:

```
adb shell chmod +x /data/local/tmp/native-shim
```

Run the Shim:

```
adb shell /data/local/tmp/native-shim /data/local/tmp/your-native-library.so
```

Attach Debugger: Attach your debugger (GDB/IDA/etc.) to the running native-shim process for analysis.

Role of Shims in Android Development

Shims in Android development play a crucial role in ensuring compatibility and stability across different versions of the Android operating system and various hardware configurations. In the context of software development, a "shim" is typically a small piece of code that provides compatibility between two otherwise incompatible systems or components. In Android, shims are used for several key purposes:

Backward Compatibility: One of the primary uses of shims in Android is to maintain backward compatibility. As Android evolves, new APIs and features are introduced, and older ones may be deprecated or changed. Shims allow apps developed for older versions of Android to run on newer versions without requiring significant changes to the app's code. This is crucial for the Android ecosystem, given the vast diversity of devices and OS versions in use.

Hardware Abstraction: Android runs on a wide variety of hardware configurations. Shims can be used to abstract hardware differences so that the same app can run on devices with different hardware components. For example, a shim might allow an app that uses a specific sensor to interact with different types of that sensor across various devices.

Vendor Customizations: Android is open-source, allowing device manufacturers (like Samsung, Huawei, etc.) to modify the OS for their specific devices. Shims can be used by these manufacturers to ensure that their customizations and additions are compatible with standard Android apps and services, and vice versa.

Security and Privacy: Sometimes, shims are introduced to enforce new security and privacy policies without breaking existing applications. For instance, if a new version of Android restricts access to certain APIs for security reasons, a shim might be used to provide alternative implementations of these APIs that adhere to the new security guidelines.

Performance Optimizations: In some cases, shims are used to optimize performance for specific hardware configurations. They can enable certain optimizations or features that are only available on particular devices or chipsets.

Testing and Debugging: Shims can also be useful in testing and debugging applications. They can be used to simulate different environments or conditions, allowing developers to test how their app behaves under various scenarios.

API Level Bridging: Google often provides support libraries and Jetpack components that act as shims. For example, the Android Jetpack libraries provide newer APIs to apps running on older versions of Android, allowing developers to use modern features while still supporting older devices.

Implementing the shim for the Acme Proximity Sensor involves creating a bridge between the Hardware Abstraction Layer (HAL) and the device driver. The provided example is a stub implementation, which means it simulates the behavior of the sensor without interacting with actual hardware. This is useful for testing or as a placeholder until the real device driver is available.

Shim Implementation
The implementation is in proximity/dev/proximity_sensor.cpp. Here's an overview of the code:

```cpp
#include "dev/proximity_sensor.h"

// Stub implementation of the proximity sensor functions

int open_sensor(proximity_params_t &params) {
    // Initialize parameters with mock values
    params.precision_min = 0;
    params.precision_range = 100;
    params.proximity_min = 0;
    params.proximity_range = 100;
    return 0; // Return a fake file descriptor
}

int poll_sensor(int fd, int precision) {
    // Return mock proximity values based on the precision
    if (precision < 0) {
        return -1;
    } else if (precision < 70) {
        return 60;
    } else if (precision < 100) {
        return 63;
    } else {
        return -1;
    }
}

int close_sensor(int fd) {
    // Close the sensor (stub implementation)
    return 0;
}
```

Building the Shim as a Library
To integrate this shim into the system, it needs to be compiled as a library. The Android.bp build configuration file is used for this purpose. Here's how you can define the library in Android.bp:

```
cc_library {
    name: "libacmeproximityshim",
    defaults: [ "vendor.acme.one.proximity.defaults", ],
    srcs: [ "dev/proximity_sensor.cpp", ],
    header_libs: [ "libhardware_headers", ],
    local_include_dirs: [ "include", ],
}

cc_library_headers {
    name: "libacmeproximityshim_headers",
    defaults: [ "vendor.acme.one.proximity.defaults", ],
    header_libs: [ "libhardware_headers", ],
    export_header_lib_headers: [ "libhardware_headers", ],
    export_include_dirs: ["include"],
}
```

cc_library: This section defines the library libacmeproximityshim. It includes the source file proximity_sensor.cpp and specifies dependencies and include directories.

cc_library_headers: This part defines the header library for libacmeproximityshim. It makes the headers available to other components that depend on this library.

Integration with HAL: Once built, this library can be used by the HAL and other components (like a daemon or a binderized HAL) to interact with the proximity sensor through the stubbed functions.

Improving App Compatibility

In the context of Android development, improving app compatibility is a significant challenge due to the platform's fragmentation and the continuous evolution of its operating system. Shims play a crucial role in addressing these challenges, ensuring that applications remain functional and consistent across different devices and OS versions. Here's how shims contribute to improving app compatibility in Android development:

- Handling API Level Differences: Android's API levels change as the operating system evolves. New features are added, and some older features are deprecated or removed. Shims act as a bridge for apps targeting older API levels, allowing them to run on newer versions of Android. This is particularly important because it prevents applications from breaking or behaving unpredictably when a user upgrades their device's operating system.
- Standardizing Across Devices: The Android ecosystem is known for its wide range of devices with varying hardware capabilities, screen sizes, and resolutions. Shims help standardize application behavior across this diversity. For instance, a shim might provide a unified way to handle screen layouts or hardware sensor data, ensuring that an app offers a consistent user experience regardless of the device it’s running on.
- Dealing with Manufacturer Customizations: Different manufacturers often customize Android, adding unique features or modifying existing ones. Shims can ensure that applications work not only on the standard Android OS but also on these customized versions. They can provide alternative implementations or adjustments needed to accommodate specific customizations made by manufacturers.
- Runtime Adjustments: Shims can be used to modify the behavior of an application at runtime based on the OS version, device capabilities, or other conditions. This adaptability is crucial for enhancing compatibility, as it allows apps to function optimally in a variety of environments without needing to be rewritten for each specific scenario.
- Smooth Transition to New Android Versions: As new versions of Android are released, shims help developers transition their apps smoothly. They provide a way to start using new APIs and features while maintaining compatibility with older versions. This gradual transition is essential for developers who need time to fully adopt the latest changes in the Android ecosystem.
- Deprecation Management: When certain APIs are deprecated in newer versions of Android, shims can offer backward-compatible alternatives. This ensures that applications continue to function while developers update their codebase to comply with the latest Android standards.
- Enhanced Testing and Quality Assurance: Shims also play a role in testing, allowing developers to simulate how an app behaves across different Android versions and devices. This is crucial for identifying and fixing compatibility issues, ensuring that updates or new releases do not negatively impact users on older versions.

Bridging the Old and the New

This concept primarily revolves around ensuring that applications remain functional and relevant across various versions of the Android operating system, which is known for its rapid evolution and diversity in terms of API levels and device capabilities. Here's a deeper look into how shims help in bridging the old and the new in Android development:

- Supporting Legacy Systems: Many users continue to operate on older versions of Android for various reasons, ranging from personal preference to hardware limitations. Shims enable developers to build apps that are compatible with newer versions of Android while still functioning correctly on older versions. This backward compatibility is crucial for reaching a broader audience and maintaining a stable user base.
- Facilitating Gradual Adoption of New APIs: As new versions of Android are released, they often come with new or updated APIs. Shims allow developers to adopt these new APIs gradually, without immediately dropping support for older versions. This means that an app can stay up-to-date with the latest functionalities offered by the newest Android version while still being accessible to users on older versions.
- Handling Deprecated APIs: Over time, certain Android APIs become deprecated, meaning they are no longer recommended for use and might be removed in future releases. Shims can provide alternative implementations or workarounds for these deprecated APIs, allowing apps to continue functioning while developers transition to newer APIs or design patterns.
- Ensuring Consistent App Behavior: Different Android versions can exhibit varied behaviors even with the same app code. Shims help in standardizing app behavior across these versions, ensuring that users have a consistent experience regardless of the specific Android version their device is running.
- Simplifying Development and Testing: Developing for multiple versions of Android can be challenging. Shims simplify this process by abstracting the complexities associated with different API levels. They also make it easier to test apps across various Android versions, ensuring compatibility and smooth performance.
- Enabling Feature Backporting: In some cases, shims allow newer features to be backported to older versions of Android to some extent. This means that even if a feature is introduced in a newer version of Android, developers can use shims to make these features available (or provide similar functionality) on older versions.

Conclusion

The native shim approach is a powerful technique for debugging and analyzing JNI libraries in Android. It provides a controlled environment to load and interact with native code, making it an invaluable tool for reverse engineering and security analysis. The practical scenarios outlined above demonstrate the versatility of this method in various contexts of Android development and security research.

HADESS

cat ~/.hadess

"Hadess" is a cybersecurity company focused on safeguarding digital assets and creating a secure digital ecosystem. Our mission involves punishing hackers and fortifying clients' defenses through innovation and expert cybersecurity services.

Website:

WWW.HADESS.IO

Email

MARKETING@HADESS.IO

To be the vanguard of cybersecurity, Hadess envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish Hadess as a symbol of trust, resilience, and retribution in the fight against cyber threats.