Application Security: Root Detection with Google Play Integrity API

Initial Root Detection Logic

Our initial approach to root detection involved implementing a series of checks within the application. These checks were designed to identify common indicators of a rooted environment. The methods employed included:

- **File System Checks:** We searched for the presence of well-known root-associated files and directories (e.g., `/system/app/Superuser.apk`, `/sbin/su`, `/system/xbin/su`). The existence of these files strongly suggests that the device has been rooted.
- **Memory Allocation Checks:** Certain rooting techniques involve specific memory allocation patterns. We implemented checks to detect unusual memory map allocations that are characteristic of root access.
- **Executing SU Commands:** A definitive way to confirm root access is to attempt the execution of the `su` command. If the command executes successfully, it signifies that the app is running with elevated privileges, indicating a rooted device.
- **Check Package Name :** We have also implemented logging to capture all apps installed on the device through their package names. While the logs successfully capture most apps, the Magisk app was not recorded.

These methods proved effective on devices that were openly rooted. However, the security landscape is constantly evolving, and sophisticated rooting tools emerged to circumvent these traditional detection methods.

The Challenge: Magisk and Root Bypass Tools

As our application security evolved, we observed that advanced users and malicious actors could employ tools like **Magisk**. Magisk is a popular systemless rooting solution that offers a suite of features, including **MagiskHide** (now referred to as **Magisk DenyList**) and various modules designed to conceal root access from applications. These tools actively work to bypass standard root detection checks by:

- Intercepting and modifying system calls that our detection logic relied upon.
- Hiding the presence of root-specific files and binaries.
- Altering memory structures to appear as a non-rooted device.

The effectiveness of Magisk and similar bypass tools rendered our initial detection logic insufficient, as it could be easily circumvented. This prompted us to seek a more robust and reliable solution.

The Problem: Our existing root detection mechanisms, while functional on basic rooted devices, were being bypassed by advanced tools like Magisk, compromising the security posture of our application.

Introducing the Google Play Integrity API

To overcome the limitations of our previous methods, we integrated the **Google Play Integrity API**. This API is a powerful, Google-provided solution designed to help developers protect their applications from various forms of fraud and abuse, including compromised devices. It provides a comprehensive suite of integrity signals that can be used to determine if an app is running on a genuine device, protected by Google Play.

How the Google Play Integrity API Works

The integration of the Google Play Integrity API involves a multi-step process, ensuring a secure and verifiable check of the device's integrity:

1. **Token Generation (Client-side):** On the Android device, using Kotlin, we generate an integrity token. This process is initiated by the application and utilizes the Play Integrity SDK. The API client is initialized, and a request is made to obtain an integrity token. This token is unique to the device and the app instance and is signed by Google Play.
2. **Token Transmission to Backend:** The generated integrity token is then securely transmitted from the mobile application to our backend server.
3. **Token Decryption and Verification (Backend):** Upon receiving the token, our backend server is responsible for its validation. This involves communicating with Google's Play Integrity API services. The process typically includes:
    - **Decrypting the Token:** The token is encrypted, and our backend uses specific Google Play services APIs to decrypt and decode its contents.
    - **Validating with Google's OAuth:** The decrypted token is passed to Google's OAuth APIs for a thorough verification. This step confirms that the token is legitimate, was issued by Google Play for our specific application, and has not been tampered with.
4. **Receiving Device Integrity Verdict:** In response to the successful verification, Google's APIs return a device integrity verdict. This verdict provides detailed information about the device's integrity, including whether it is considered rooted, running a modified OS, or exhibiting other signs of compromise.
5. **Decision Making:** Based on the integrity verdict received from Google, our application can then make informed decisions. If the device is flagged as compromised or rooted, we can restrict access to sensitive features or revoke permissions to protect our application and user data.

The Concern: Server-side Internet Connectivity

A significant challenge we encountered lies in the backend's requirement for internet connectivity. To validate the integrity token, our application server must communicate with Google's external APIs (`oauth.googleapis.com` and Play Integrity validation endpoints). This dependency poses a problem in scenarios where the application server itself might not have direct, reliable internet access, or when there are strict network segmentation policies in place.

The Concern: Our UAT App server needs internet access to communicate with Google APIs for token validation. If the app server lacks this connectivity, the integrity check cannot be performed.

Proposed Solution: A Microservice for Integrity Checks

To address the internet connectivity concern for our main application server, we are considering a microservice architecture. The proposed solution involves deploying a dedicated, lightweight **microservice** whose sole responsibility is to handle the communication with Google's Play Integrity API for token validation. This microservice would be hosted in an environment where internet access is guaranteed and properly configured.

How the Microservice Architecture Works:

1. **App Server Receives Token:** The main application server receives the integrity token from the mobile client, as before.
2. **Forwarding to Microservice:** Instead of directly calling Google APIs, the app server forwards the integrity token to the dedicated microservice. This communication between the app server and the microservice would typically occur over a local network or internal service bus, which is generally reliable.
3. **Microservice Validation:** The microservice, residing in an environment with internet access, then performs the necessary API calls to Google's Play Integrity validation endpoints to verify the token.
4. **Returning the Verdict:** The microservice receives the integrity verdict from Google and returns this result (e.g., "granted," "denied," "unknown") back to the main application server.
5. **App Server Action:** The main application server acts upon the verdict received from the microservice, enforcing security policies accordingly.

This architectural approach isolates the internet-dependent validation logic, allowing our core application server to function even in environments with limited external connectivity. It enhances modularity and maintainability while ensuring that robust root detection can still be effectively implemented.

Conclusion

The evolution from traditional file-based root detection to leveraging the Google Play Integrity API represents a significant advancement in our application's security. While the initial methods were straightforward, they proved insufficient against modern obfuscation techniques. The Play Integrity API offers a more reliable and comprehensive solution by utilizing Google's trusted infrastructure. The architectural consideration of a dedicated microservice for handling external API communication further strengthens our ability to implement this critical security feature, ensuring a more secure and trustworthy experience for our users.