Inside Android's SafetyNet Attestation

Collin Mulliner & John Kozyrakis

About

Dr.-Ing. Collin Mulliner

collin@mulliner.org

@collinrm

Independent Security Researcher

Mobile Security since 1999

Worked on: J2ME, PalmOS, Symbian, Windows Mobile, iOS and Android Security. Co-authored 'The Android Hacker's Handbook', built an Android-based device.

John Kozyrakis

john@koz.io

@ikoz

Applied Research Lead, Mobile, Synopsys SIG R&D

6y+ Security Consultant @ Cigital

Mobile app protection design & testing for several large US & UK orgs

Mobile static & dynamic analysis tools

Agenda

- Mobile App Security
- SafetyNet & Attestation
- Developer's Perspective
- Bypassing SafetyNet
- Conclusions & Future

Rooting & root detection

Mobile App Security

- App is the gateway to the service
  - More so if mobile first or mobile only (and no public APIs)

- Data displayed & managed by app
  - User is allowed to see content in the app but isn't allowed to copy it

Mobile App Security protects: Service, Revenue, Brand, User / Customer

Rooting

- Why attack a mobile app?
  - Analyse internals, use enrolled identity, disable security controls, use low-level APIs etc
- Having the ability to escalate the privileges of a process to “root”
  - Regain full control over device
  - Just one step towards attacking apps

- Access any resource
  - Take screenshot, debug any app, instrument process
- Read / Write any file
  - Read private app data
- Modify OS and software framework
  - API returns different result

Highly dependent on Android version due to SELinux (longer discussion...)

Attack patterns

- OS Modification
  - Root device ➜ break security assumptions
    (read private data, take screenshot, instrument app, …)
  - Enables post-installation app tampering & hooking

- Static App Modification
  - Make custom app version that does “something else”, bypass security controls

- Network Traffic
  - Modify request / response (mostly solved with TLS and cert-pinning)

OS modification methods

- Userspace vulnerabilities
  - symlink errors, arbitrary write etc
  - various escalation techniques follow

- Kernel / TEE vulnerabilities
  - temporary escalation of privileges of exploit process to root

- Bootloader unlock
  - Allows flashing or booting into custom system images
  - Change recovery -> edit /system via recovery
  - Change kernel -> custom kernel with backdoor to gain root
  - Change operating system -> new OS comes with root preinstalled

Device integrity detection the old Days

- Check for traces for “rooting”
  - Presence of files: access(“/system/xbin/su”, F_OK)
  - Presence of apps: com.chainfire.supersu installed?
  - Presence of running processes, root shells etc
  - Unexpected output of commands, exec(“which su”)
  - ...
- Check for instrumentation tools
  - Xposed installed ?
- Emulator detection
  - if (getDeviceId() == 0) ....

That’s a low bar

- Developer, easy to:
    - Understand
    - Implement
    - Deploy (app doesn’t start or tells backend to deny access)

- Attacker, easy to:
    - Understand
    - Circumvent (remove check from app, rename file, ...)
    - (Ab)use app

Hardcoded checks

- The remote backend does not reliably know if checks were executed
- Device integrity != app integrity
- It all runs within the process space of the (unprivileged) app

- All client-side checks can eventually be bypassed, but we can raise the bar

Attackers can easily disable detections

```
isRooted = findRoot()

if (!isRooted){

    business_logic()

}
```

```
findRoot{

    if (Config.rootDetectON){

        return doChecks()

    }

}
```

Usually easy to change **one** variable and disable all root detection across app

Attackers can easily feed checkers with bad data

- If implemented in Java:
  - Smali editing / repackaging
  - Runtime hooking (substrate, xposed, frida)
- If obfuscated Java:
  - Mass function tracing to discover checks, then hooking of OS APIs
    - access(), open(), stat()
- If implemented in C/C++
  - C API tracing & hooking (frida, library injection etc)
- If syscall invocation via ASM:
  - Syscall tracing & custom kernel hooking

Raising the bar

- **Collect data on the client but enforce restrictions on the backend**

- Attacker can’t just patch out checks but has to
  - Find which pieces of collected data is important (moving target)
  - Fake that data in meaningful ways
    - Much more work and uncertainty about what is used for check

- **This is what SafetyNet Attestation does**

SafetyNet History & Architecture

SafetyNet

The system Google uses to keep the Android ecosystem in check and gather metrics on on-going attacks

- Performs some on-device checks
- Collects device data
- Sends results back to Google for analysis

Google, over time, can create a profile of each device using these data points.

Google also holds "compatibility" profiles for certain devices via CTS

SafetyNet details

- SafetyNet mostly *collects data* as the GMS process
  - Slightly elevated privileges
- Data sent to Google
  - Behavioral analysis
  - Machine learning
  - Visibility over whole ecosystem, attack patterns & trends
  - CTS profile comparison
- System is highly flexible (pushed configs, pushed binary updates)
- High level of integrity protection (signed binaries)
- High complexity

SafetyNet Attestation

SafetyNet Attestation is one of several services offered by SafetyNet to developers.

“OK Google, what do you think about the device I’m running in?”

The response can be:

- This device is definitely tampered & rooted
- This device is tampered in some way that diverges from device profile
  - Not “Google-approved any more”
- All seems good

Attestation result depends on a *subset* of collected data

caveats

- Attestation aims to let developers understand if a ***device*** is tampered
  - Compared to it’s factory state
- It does not warn if the device is vulnerable
  - Although the current patch level & kernel & OS version are collected
- It is not the best way for reasoning about ***application*** integrity

Criticism

- Attestation will not pass on non-CTS devices
  - Depends on Google Play Services
  - Excludes amazon, lineage, cyanogen, copperhead...
  - Some view it as an attempt to further monitor & control the Android ecosystem
  - Some say it's anti-competitive
- Privacy
  - Checks are not transparent
  - Documentation was lacking - getting better over time
  - Initially not obfuscated jar, that changed on Oct 2016
  - Snet attempts to avoid "accidental" collection of private information (strict regexes)
  - Several collectors disabled by default, enabled if/when needed in response to threats
  - Most collected info does not actually require or use elevated system privileges
  - Most ad & root detection libs collect more sensitive info

SafetyNet JAR

- SafetyNet is a Play Services chimera dynamite module
- The code for most collectors/checkers lives in a **signed jar** file (dex)
- This file is downloaded through a static URL by GMS at runtime
    - Loaded into memory
    - Pinned connection
- Safenet jar is updated every couple of months.
- Latest: https://www.gstatic.com/android/snet/11292017-10002001.snet
- Finding the latest:
    - https://www.gstatic.com/android/snet/snet.flags
    - https://www.gstatic.com/android/snet/snet_goog.flags
    - Automate download: https://github.com/anestisb/snet-extractor/ by Anestis @ Census

Snet History (not comprehensive)

- 1626247 - December 2014
- 1839652 - April 2015
- 2097462 - July 2015
- 2296032 - September 2015
- 2495818 - December 2015
- 10000700 - August 2016
- 10000801 - September 2016
- 10001000 - March 2017
- 10001002 - April 2017
- 10002000 - November 2017
- **10002001 - December 2017**

SafetyNet modules

- apps
- attest
- captive_portal_test
- carrier_info
- davlik_cache_monitor
- device_admin_deactivator
- **device_state**
- event_log
- **su_files**
- gsmcore
- google_page_info
- google_page
- ssl_handshake
- locale
- logcat
- mx_record
- default_packages
- proxy
- ssl_redirect
- sd_card_test
- selinux_status
- **settings**
- **setuid_files**
- sslv3_fallback
- suspicious_google_page
- system_ca_cert_store
- **system_parition_files**
- mount_options
- app_dir_wr
- phonesky
- internal_logs
- app_ops
- snet_verify_apps_api_usage

Example: device_state

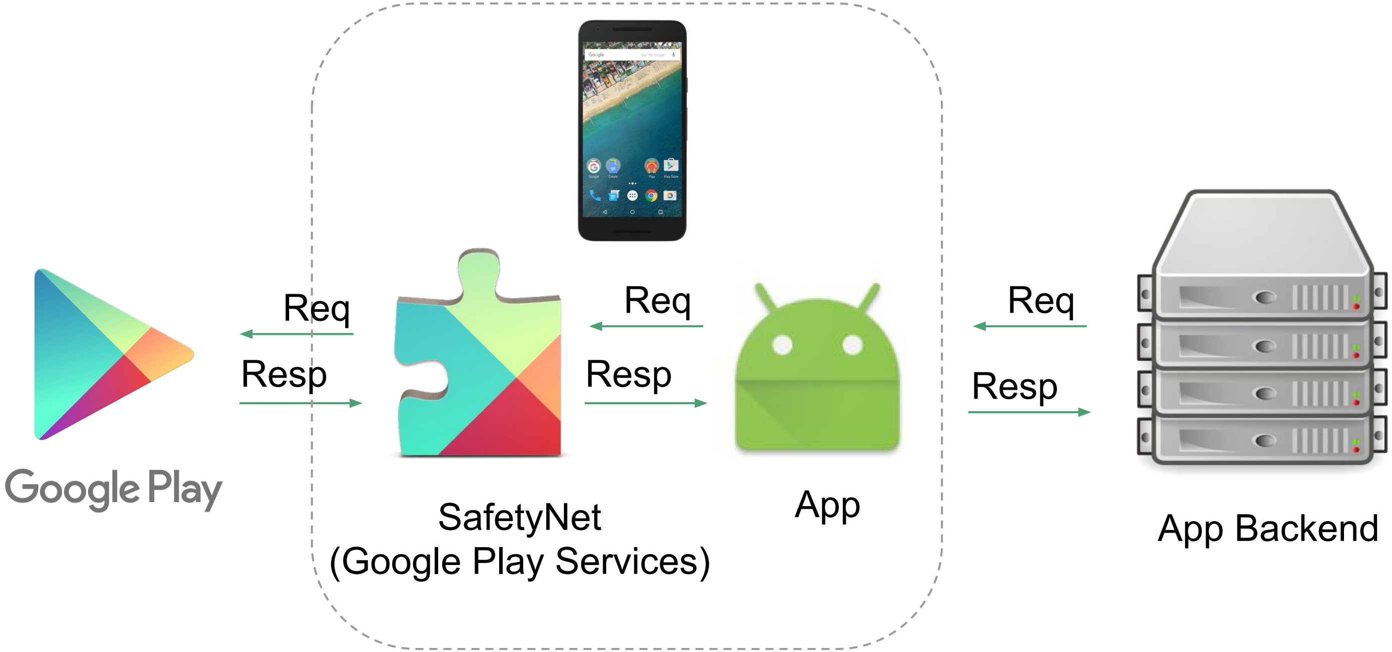
```
static DeviceState getDeviceState(Context ctx, GBundle gbundle) {
    Object propertyName;
    Iterator iter;
    DeviceState deviceState = new DeviceState();
    deviceState.verifiedBootState = DeviceStateChecker.systemPropertyStringValue("ro.boot.verifiedbootstate");
    deviceState.verityMode = DeviceStateChecker.systemPropertyStringValue("ro.boot.veritymode");
    deviceState.securityPatchLevel = DeviceStateChecker.systemPropertyStringValue("ro.build.version.security_patch");
    deviceState.oemUnlockSupported = DeviceStateChecker.systemPropertyIntValue("ro.oem_unlock_supported");
    deviceState.oemLocked = Build$VERSION.SDK_INT > 23 ? DeviceStateChecker.getFlashLockState(ctx) : DeviceStateChecke
    deviceState.productBrand = DeviceStateChecker.systemPropertyStringValue("ro.product.brand");
    deviceState.productModel = DeviceStateChecker.systemPropertyStringValue("ro.product.model");
    deviceState.kernelVersion = Utils.readVirtualFile("/proc/version");
    List systemPropertyNames = gbundle.getSystemPropertyNames();
    if(systemPropertyNames.size() > 0) {
```

- verifiedBootState
  - Verified,
  - SelfSigned
  - Unverified
  - Failed
- verityMode
  - enforcing
  - logging

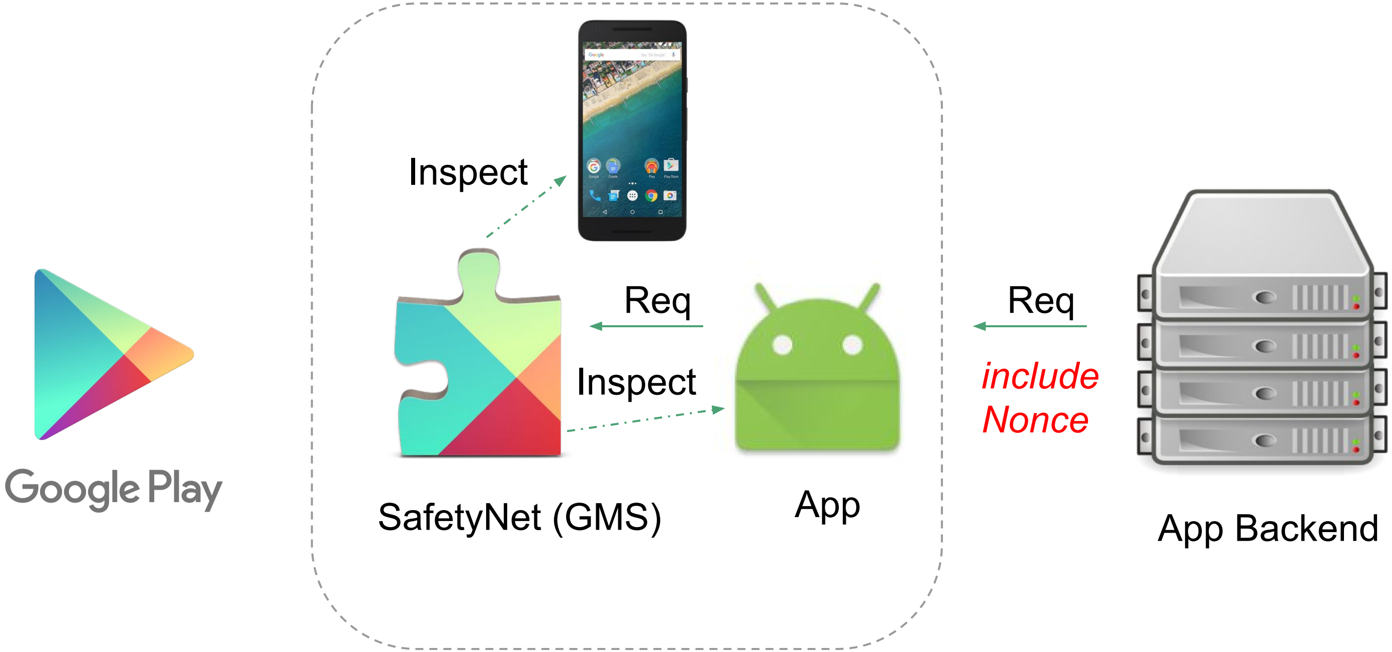
- securityPatchLevel
- oemUnlockSupported
- oemLocked
- productBrand
- productModel
- kernelVersion
- systemPropertyList
- SOFTWARE_UPDATE_AUTO_UPDATE setting
- Samsung fotaclient installation

SafetyNet Attestation: Overview

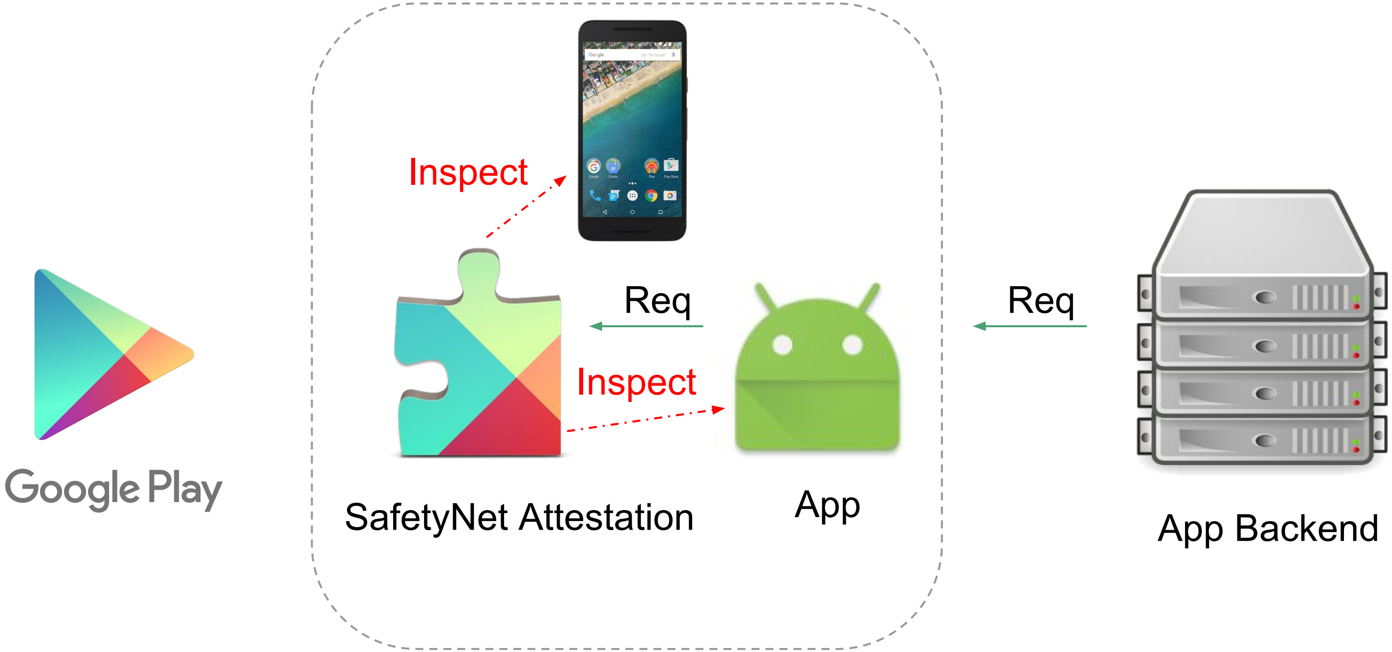
SafetyNet Attestation: Call Chain



SafetyNet Attestation: Request Attestation

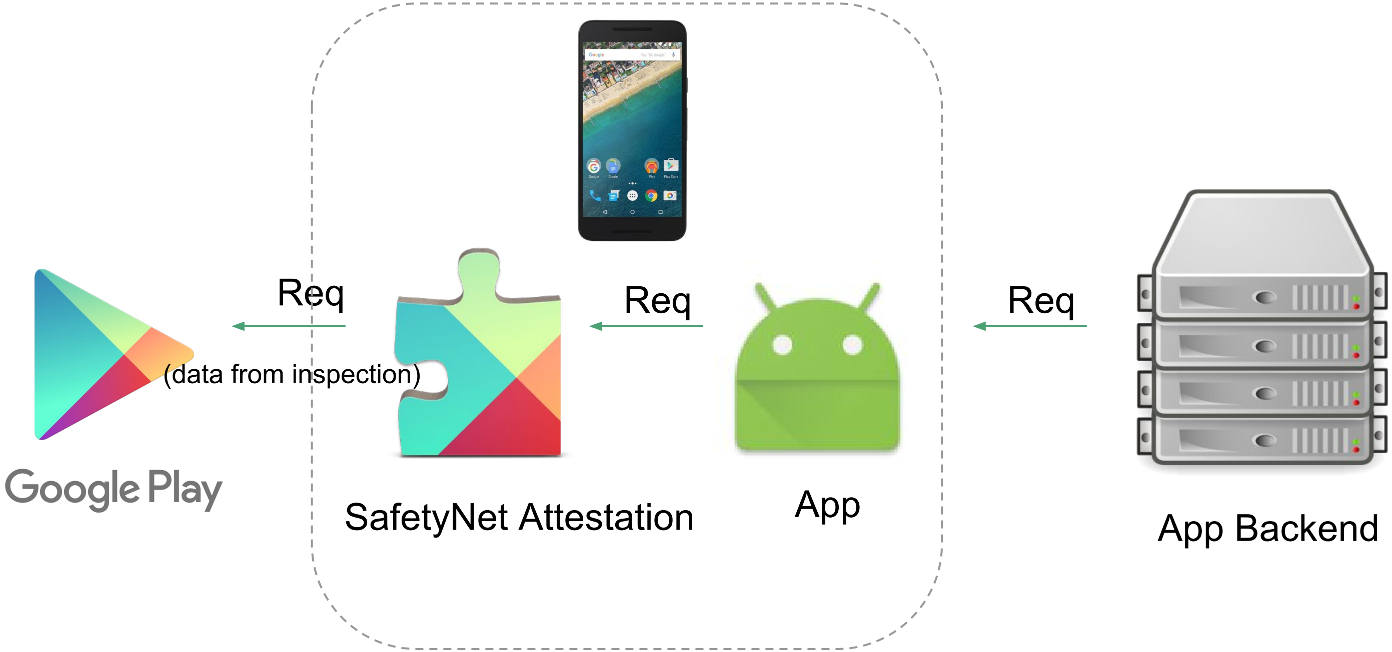


SafetyNet Attestation Overview: Request Attestation

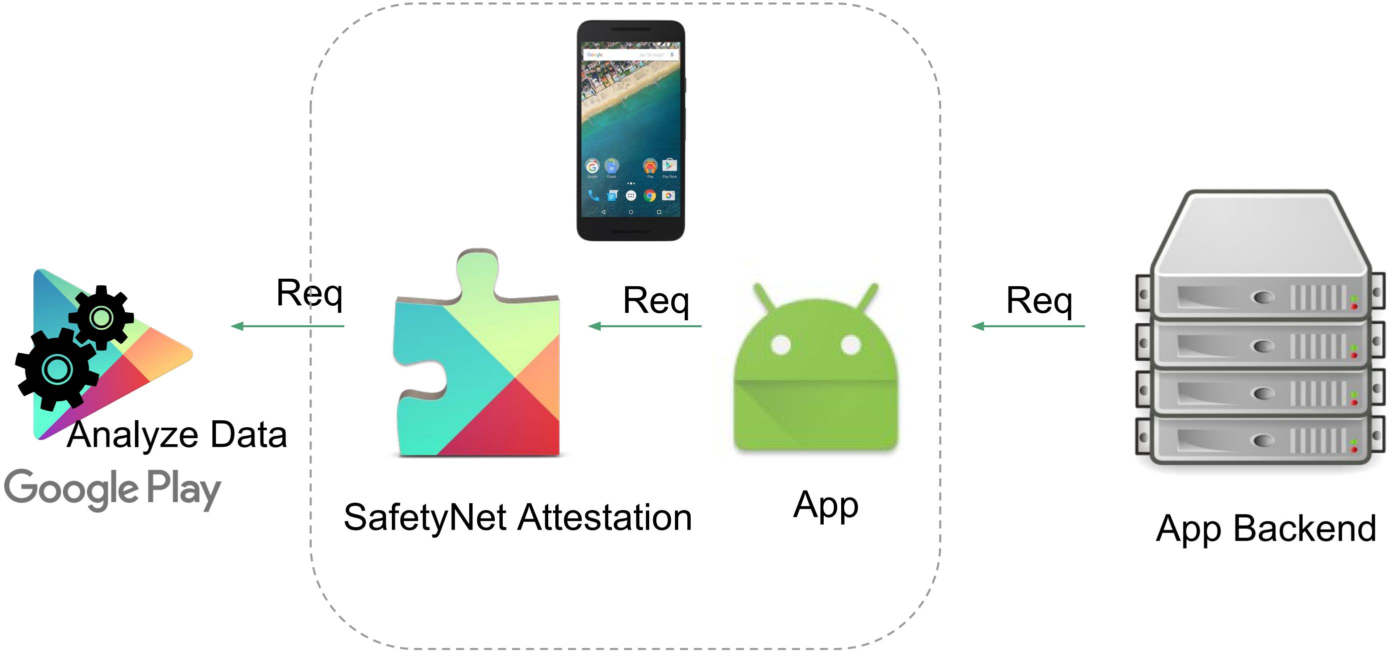


This is what every app used to implement for themselves

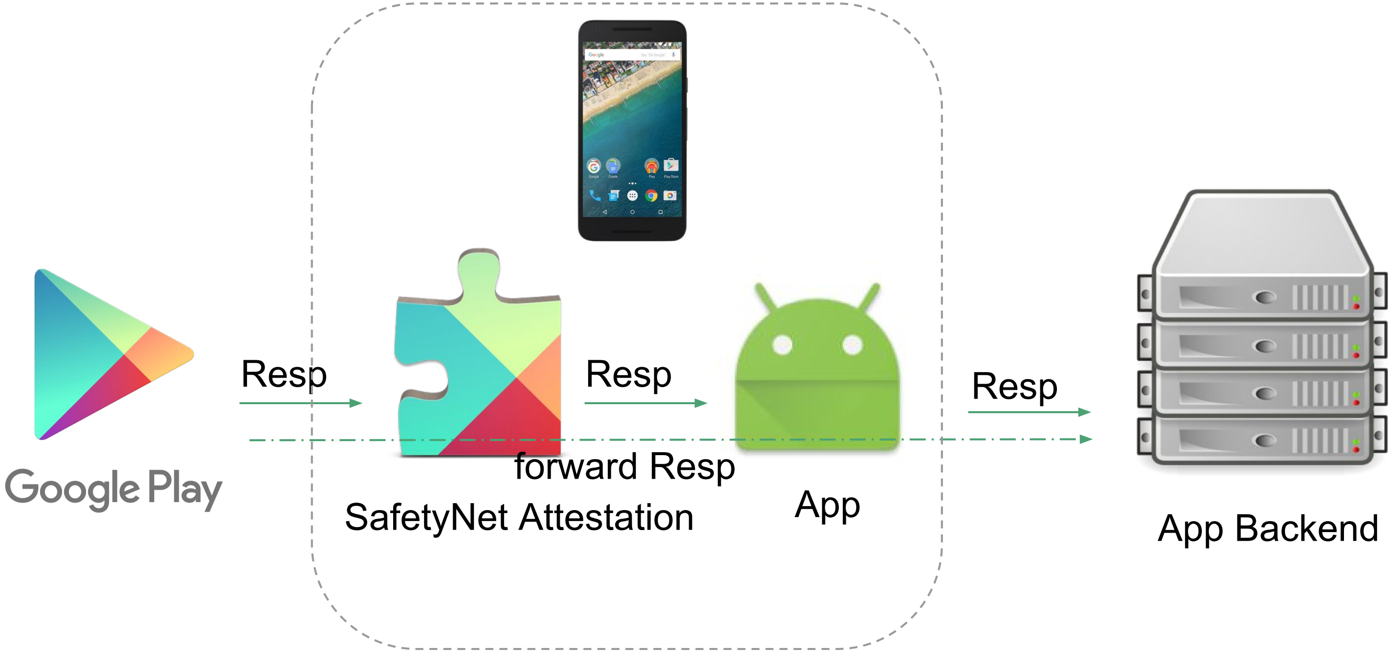
SafetyNet Attestation: Forward Data



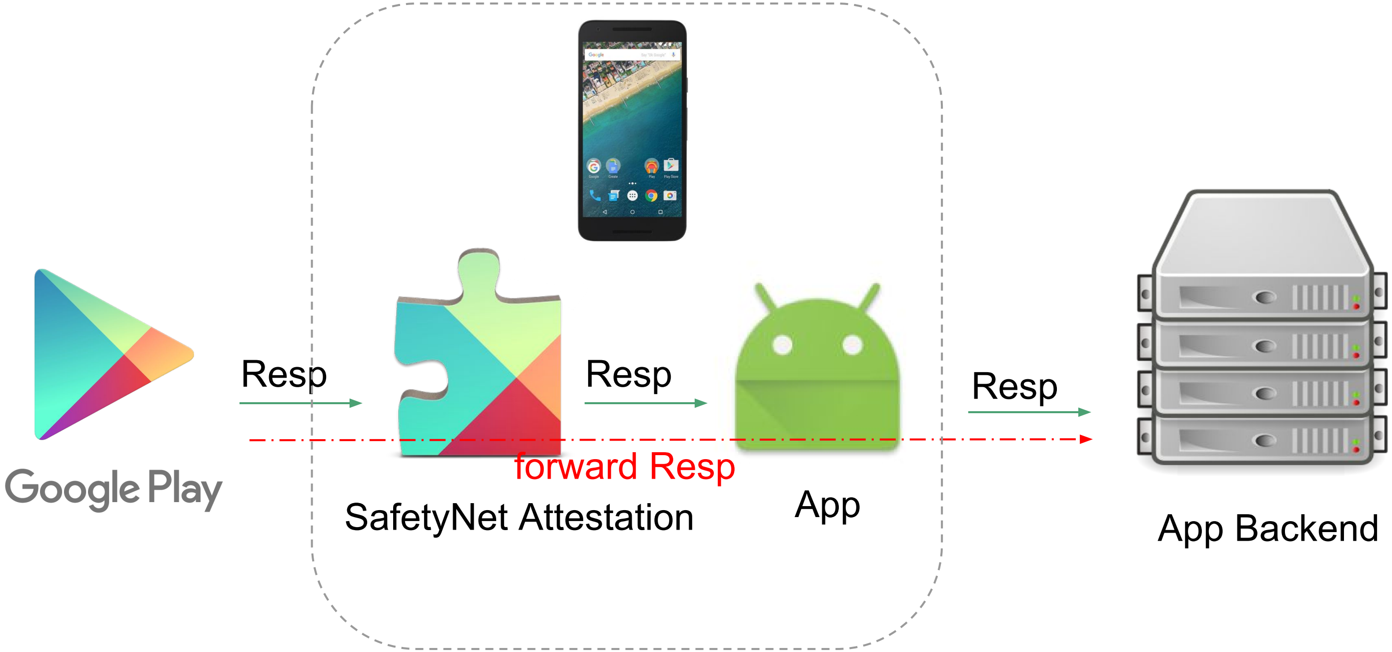
SafetyNet Attestation: Attest Device & App



SafetyNet Attestation: Deliver Result

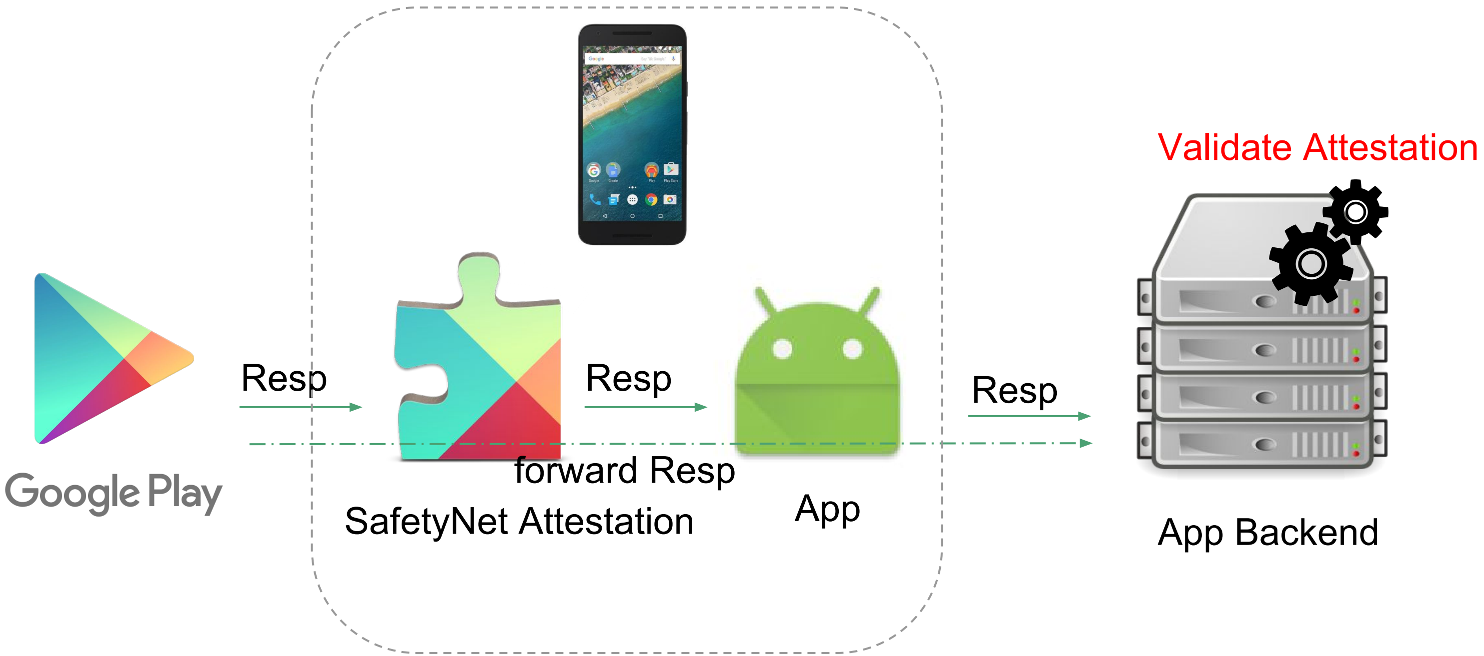


SafetyNet Attestation: Deliver Result



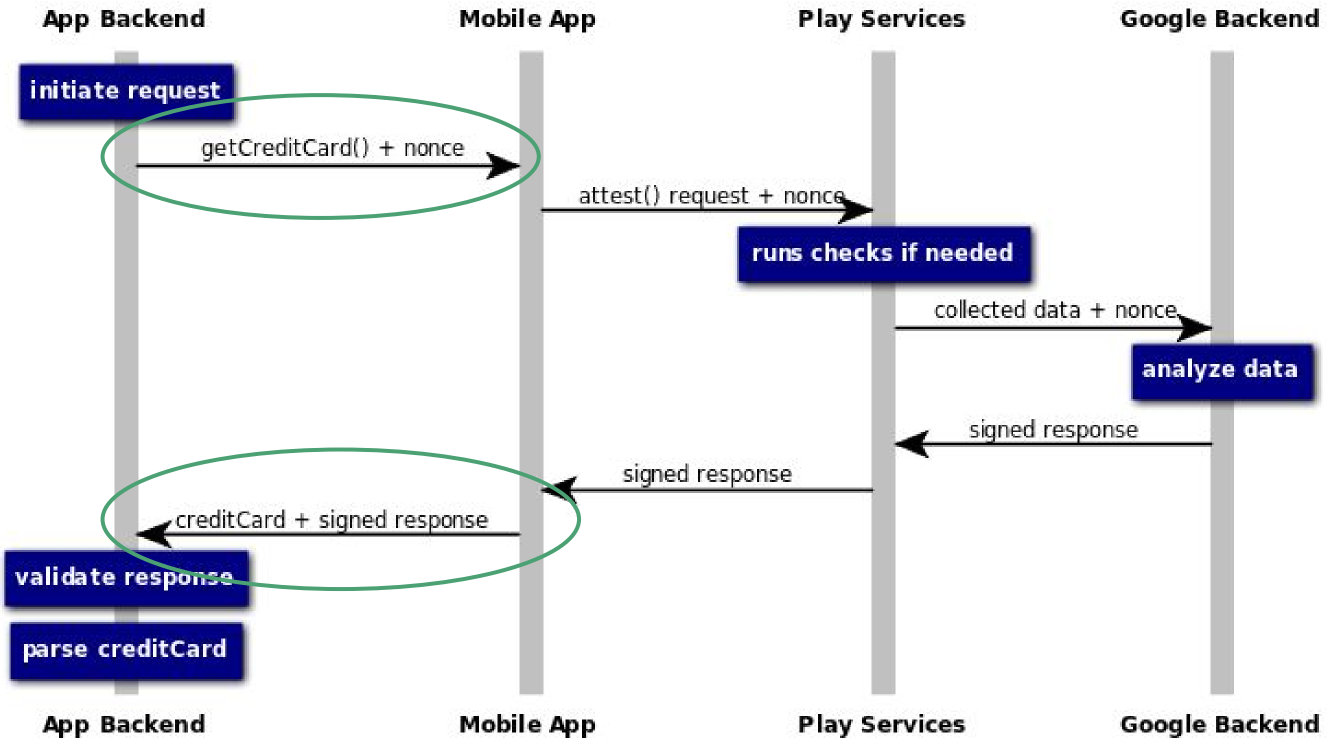
Response is cryptographically protected - signed by Google

SafetyNet Attestation: Deliver Result



Using it in apps

Ideal implementation



Reference: https://www.synopsys.com/blogs/software-security/using-safetynet-api/

Attestation result validation

can be implemented in multiple ways, not all of them are secure

- Where to validate?
  - Only at server, not inside mobile app
- How to use?
  - Tie validation to your own APIs is ideal
  - Run attest/validate throughout user session, not just on app start
- Use & validate nonces
- Check all returned fields
- Check crypto
- Decide if using just basicIntegrity or ctsProfileMatch too
- Handle errors

Attestation Result

Cert Chain .

Attestation Data .

Signature

Format: JSON Web Signature (JWS)

base64(rsa_sign(sha-256(base64(header)+base64(attest_data))))

eyJhbGciOiJSUzI1NiIsIng1YyI6WyJNSUlFZmpDQ0EyYWdBd0lCQWdJSVZaeDlNZDVhb3JVd0RRWUpLb1pJaHZjTkFRRUxCUUF3U1RFTE1Ba0dBMVVFQmhNQ1ZWTXhFekFSQmdOVkJBb1RDa2R2YjJkc1pTQkpibU14S1RBakJnTlZCQU1USE
VkdmIyZHNaU0JKYm5SbGNtNWxkQ0JCZFhSb2IzSnBkSGtnUnpJd0hoY05NVFV3T0RNeE1qQXpOalE0V2hjTk1UWXdPRE13TURBd01EQXdXakJzTVFzd0NRWURWUVFHRXdKVlV6RVRNQkVHQTFVRUNBd0tRMkZzYVdadmNtNXBZVEVXTUJRR0Ex
VUVCd3dOVFc5MWJuUmhhVzRnVm1sbGR6RVRNQkVHQTFVRUNnd0tSMjl2WjJ4bElFbHVZekViTUJrR0ExVUVBd3dTWVhSMFpYTjBMbUZ1WkhKdmFXUXVZMjl0TUlJQklqQU5CZ2txaGtpRzl3MEJBUUVGQUFPQ0FROEFNSUlCQ2dLQ0FRRUEzaW
pVemNKOHl3NmhlYnpiQTRYbDJsOTM0dG96SFYyNWdJZ2VMNnU0eWVNNE4yMTh4WitPMWhkelBLbmR6bjArc1VuUHNTekl6SWZiMzV3Nk9xRDlxLysydlk5OUN3T2c0RXF2QXU2OTV1ZjVibzFjNk4rcHpNOWRWMDZIR3dSdUUxUE1OY2Y4Y01C
UEJDZy9jWmo2bUlsbFdGVXFERlFmVE5tL25vU0lucmg2WUpUOWhvdUJ6U2d5ZE1Kb2NsYnZEdjlEcThFQ1lWUVhFanA4Z00yVWNnOTNTZXhjb2xmZCtLVUFrNXdkaVBTeXhINFVRaDFvV25iMFR1bzJzeUpQZHh1cWQ3MVRFd1NweE5wcDZxZE
Ficy9XNE8vZ2swMVVxWEVqbFZvaFhmSE1sbHZsZEd5dWhEM0Z0dFIzOEFEb0dRaWVUVnlzK2VaZWY3ZXYzem9uNFFJREFRQUJvNE1CUlRDQ0FVRXdIUVlEVlIwbEJCWXdGQVlJS3dZQkJRVUhBd0VHQ0NzR0FRVUZCd01DTUIwR0ExVWRFUVFX
TUJTQ0VtRjBkR1Z6ZEM1aGJtUnliMmxrTG1OdmJUQm9CZ2dyQmdFRkJRY0JBUVJjTUZvd0t3WUlLd1lCQlFVSE1BS0dIMmgwZEhBNkx5OXdhMmt1WjI5dloyeGxMbU52Y1M5SFNVRkhNaTVqY25Rd0t3WUlLd1lCQlFVSE1BR0dIMmgwZEhBNk
x5OWpiR2xsYm5Sek1TNW5iMjluYkdVdVkyOXRMMjlqYzNBd0hRWURWUjBPQkJZRUZIVGh6cHVGbTNYcGs5c2xScDlRLzNSTGVNK2NNQXdHQTFVZEV3RUIvd1FDTUFBd0h3WURWUjBqQkJnd0ZvQVVTdDBHRmh1ODltaTFkdldCdHJ0aUdycGFn
Uzh3RndZRFZSMGdCQkF3RGpBTUJnb3JCZ0VFQWRaNUFnVUJNREFHQTFVZEh3UXBNQ2N3SmFBam9DR0dIMmgwZEhBNkx5OXdhMmt1WjI5dloyeGxMbU52Y1M5SFNVRkhNaTVqY213d0RRWUpLb1pJaHZjTkFRRUxCUUFEZ2dFQkFENkxLN25UZl
haUzZEMTg1ZlQvencxVGp0SUxOditrYlE3bVJZT2Z6dzY5bW1xWGNaeFppZllsNXRsdWVNZ0xzWFNFOWJQRXNKZk9hZzJLSnFiTVhXUUpGR1F5cmJ1OGszeDZXNDEvNWkzdUl6ZWsvTm5hZ00yV2hmK2lYcWcrdkxmakgyVlJoRmtQQ2k4Z21D
TDZneEZidm5ldUd5UlpyMEErS3NOUUxMMW1SQ3RjLzZRYWF0ZWV5Uy9TMmVGcVJaT2NJN2hpak95QTdvRUo4ZDNJMnl0ZXdJSmlWd2dMZDNmYWRyekpwVmFyN1ZRR2ljRnJUK0doVnpHS1d4U1E0VEQzdUhZY0hZTAwR2VYUVoxMms3SEtEWD
RpRUNrek9jMEtXbG1WVXNXMXRrMTJnTitXQXlkM0QrVkdhV1lwQjNYeWd4VytTd3JrSkZoalpOaURBRkE9IiwiTUlJRDhEQ0NBdGlnQXdJQkFnSURBanFETUEwR0NTcUdTSWIzRFFFQkN3VUFNRUl4Q3pBSkJnTlZCQVlUQWxWVE1SWXdGQVlE
VlFRS0V3MUhaVzlVY25WemRDQkpibU11TVJzd0dRWURWUVFERXhKSFpXOVVjblZ6ZENCSGJHOWlZV3dnUTBFd0hoY05NVE13TkRBMU1UVXhOVFUyV2hjTk1UWXhNak14TWpNMU9UVTVXakJKTVFzd0NRWURWUVFHRXdKVlV6RVRNQkVHQTFVRU
NoTUtSMjl2WjJ4bElFbHVZekVsTUNNR0ExVUVBeE1jUjI5dloyeGxJRWx1ZEdWeWJtVjBJRUYxZEdodmNtbDBlU0JITWpDQ0FTSXdEUVlKS29aSWh2Y05BUUVCQlFBRGdnRVBBREDNDQVFvQ2dnRUJBSndxQkhkYzJGQ1JPZ2FqZ3VEWVVVaThp
VC94R1hBYWlFWis0SS9GOFluT0llNWEvbUVOdHpKRWlhQjBDMU5QVmFUT2dtS1Y3dXRaWDhiaEJZQVN4RjZVUDd4YlNEajBVL2NrNXZ1UjZSWEV6L1JURGZSSy9KOVUzbjIrb0d0dmg4RFFVQjhvTUFOQTJnaHpVV3gvL3pvOHB6Y0dqcjFMRV
FUcmZTVGU1dm44TVhIN2xOVmc4eTVLcjBMU3krckVhaHF5ekZQZEZVdUxIOGdaWVIvTm5hZytZeXVFTldsbGhNZ1p4VVlpK0ZPVnZ1T0FTaERHS3V5Nmx5QVJ4em1aRUFTZzhHRjZsU1dNVGxKMTRyYnRDTW9VL000aWFyTk96MFlEbDVjRGZz
Q3gzbnV2UlRQUHVqNXh0OTcwSlNYQ0RUV0puWjM3RGhGNWlSNDN4YStPY21rQ0F3RUFBYU9CNXpDQjVEQWZCZ05WSFNNRUdEQVdnQlRBZXBob2pZbjdxd1ZrREJGOXFuMWx1TXJNVGpBZEJnTlZIUTRFRmdRVVN0MEdGaHU4OW1pMWR2V0J0cn
RpR3JwYWdTOHdEZ1lEVlIwUEFRSC9CQVFEQWdFR01DNEdDQ3NHQVFVRkJ3RUJCQ0l3SURBZUJnZ3JCZ0VGQlFjd0FZWVNhSFIwY0RvdkwyY3VjM2x0WTJRdVkyOXRNQklHQTFVZEV3RUIvd1FJTUFZQkFmOENBUUF3TlFZRFZSMGZCQzR3TERB
cW9DaWdKb1llYUhSMGNEb3ZMMmN1YzNsdFkySXVZMjl0TDJOeWJITXZaM1JuYkc5aVlXd3VZM0pzTUJjR0ExVWRJQVFRTUE0d0RBWUtLd1lCQkFIV2VRSUZBVEFOQmdrcWhraUc5dzBCQVFzRkFBT0NBUUVBcXZxcElNMXFaNFB0WHRSKzNoM0
VmK0FsQmdERkpQdXB5QzF0ZnQ2ZGdtVXNnV00wWmo3cFVzSUl0TXN2OTErWk9tcWNVSHFGQll4OTBTcEloTk1KYkh6Q3pUV2Y4NEx1VXQ1b1grUUFpaGNnbHZjcGpacE55NmplaHNnTmIxYUhBMzBEUDl6NmVYMGhHZm5JT2k5UmRvekhRWkp4
anlYT04vaEtUQUFqNzhRMUVLN2dJNEJ6ZkUwMExxaHVrTllRSHBtRWN4cHc4dTFWRHU0WEJ1cG43akxyTE4xbkJ6LzJpOEp3M2xzQTVyc2Iwell1hSW14c3NEVkNiSkFKUFpQcFpBa2lEb1VHbjhKeklkUG1YNERrallVaU9uTURzV0NPcm1qaT
lENlg1MkFTQ1dnMjNqclc0a09WV3plQmtvRWZ1NDNYclZKa0ZsZVcyVjQwZnNnMTJBPT0iXX0.eyJub25jZSI6IjFYSlNLUDJqWXAxRk1abkVaWUk5RlE9PSIsInRpbWVzdGFtcE1zIjoxNDQ2NzYwMzgyMjQ3LCJhcGtQYWNrYWdlTmFtZSI6
ImNvbS5leGFtcGxlLnNhZmV0eW5ldHRlc3Quc2FmZXR5bmV0dGVzdCIsImFwa0RpZ2VzdFNoYTI1NiI6Imh6TGJPSWlYYURSLzVRM014MVljNTQyV29OT2lnc3V1MEhwWFJFYTRqU0k9IiwiY3RzUHJvZmlsZU1hdGNoIjp0cnVlLCJleHRlbn
Npb24iOiJDUjM3cjh1QVoya0ciLCJhcGtDZXJ0aWZpY2F0ZURpZ2VzdFNoYTI1NiI6WyJmM2ZrbHp5Q1BPMXo5LzB6bytoR29haE8rcE9nWGR6UW5adnk5blFDQ2FvPSJdfQ.bZJj8fZeWuByLg4u34S4Kr0wMsCQqJuLpvGjnGhzFKmSPzT2H
VUUPjCZ8IAtTg-XCP2eAcRr_FhEMaHthkUsw3OmCgw-V-dMb6IiJIcPiEvDfkeSqbLGkoXEWW8uqSxy0iXxLTNrNX20oIviCEznFvVgoBwZVLS7vtsK1Ak8Fzb1Kmr2NiTcd1VqdvcoQ-cvqc-benqdJpYNcTE2Qp534B_nuimiC_ZJoKWpSAT
Ie5-Ge4CkOeHC1ilw76aWRyb7rh4GAchqs-_QDQucFTbZFpfK4q7-pDLgCtYiqgsiv8959lllooP8sHxRMd-d99rckkekUnGCdqbM8xyNmkgc8A

Check crypto!

- Extract JWS cert chain
    - (there should only be one chain)
- Validate chain
- Pin anchor (google)
- OSCP/CRL check certs
- Valid leaf hostname
    - attest.google.com,
- validate JWS signature

Attestation Result

- JWS object - signed by Google
- Contains nonce, package name, certificate details etc


```
{
    "nonce": "R2Rra24fVm5xa2Mg",
    "timestampMs": 9860437986543,
    "apkPackageName": "com.package.name.of.requesting.app",
    "apkCertificateDigestSha256": ["base64 encoded, SHA-256 hash of the certificate used to sign requesting app"
    "apkDigestSha256": "base64 encoded, SHA-256 hash of the app's APK",
    "ctsProfileMatch": true,
    "basicIntegrity": true,
}
```

ctsProfileMatch & basicIntegrity

Device Status	Value of "ctsProfileMatch"	Value of "basicIntegrity"
Certified, genuine device that passes CTS	`true`	`true`
Certified device with unlocked bootloader	`false`	`true`
Genuine but uncertified device, such as when the manufacturer doesn't apply for certification	`false`	`true`
Device with custom ROM (not rooted)	`false`	`true`
Emulator	`false`	`false`
No device (protocol emulator script)	`false`	`false`
Signs of system integrity compromise, such as rooting	`false`	`false`
Signs of other active attacks, such as API hooking	`false`	`false`

SafetyNet and the Nonce

Nonce ➔ number used once

- **Prevent replay and reuse of attestation result**
  - **Also sharing between users/devices...**
- Nonce needs to be unique (used once!)
- Derive from account information or transaction information
- Nonce needs to be verified correctly
  - Time diff {nonce gen / “timestamp” field in attest resp | packet timestamp}
  - Nonce value check

Handle errors!

Error cases

The JWS message can also show several types of error conditions:

- A `null` result indicates that the call to the service didn't complete successfully.
- An `"error"` field indicates that an issue occurred, such as a network error or an error that an attacker feigned. Most errors are transient and should be absent if you retry the call to the service. You may want to retry a few more times with increasing delays between each retry. Keep in mind, however, that if you trigger more than 5 calls per minute, you could exceed the rate limit, which causes the remaining requests during that minute to return an error automatically.

> **Note:** If an error occurs, the result cannot represent a passed test, as an attacker might intentionally trigger such an error.

Errors!

```
{"extension":"CaOav6U9qRO1",
"ctsProfileMatch":false,
"nonce":"Ehq+1HB3KyRWAT8zv\/vDmw==",
"apkCertificateDigestSha256":[],
"timestampMs":1471950172731,
"basicIntegrity":false}
```

The package name and APK digests are missing!

Again this is a side note in their documentation.

No actual example in their docs!

```
{"extension":"CYOUMWN1YUXN",
"Error":"internal_error",
"apkCertificateDigestSha256":[]}"
```

This means the API works but the attestation failed to run!

Attestation: just an API Call away!?

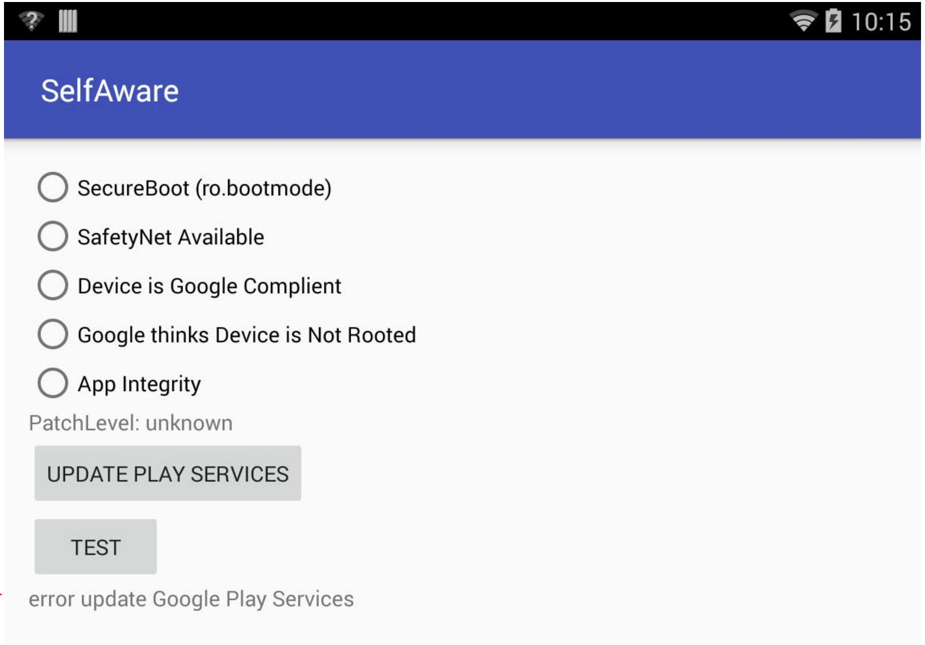
- **All API calls can and WILL fail in the wild!**
  - Solution: report failure codes to your backend (only you can decide what to do)

- Connection to GoogleApiClient fails
  - General connection error ➔ retry
  - Error code 2 ➔ Google PlayServices doesn’t support SafetyNet ➔ UPDATE PlayServices

- SafetyNet attest() call fails
  - Nonce too short (SHOULD NOT HAPPEN TO YOU)
  - Rate limited (add API_KEY + request bigger quota)
  - **Generic error ➔ this will happen to you**

PlayServices too old

Android 4.4 no SecureBoot!



API Failures...

- **Start with retrying everything** (generic errors and network errors!)
  - Be a good citizen and use exponential backoff!

- attest()
  - Inspect attestation result on the client to determine if JSON error field is present
    → base64 decode → parse json → error field present?
    - YES → retry

- If everything fails report to your backend ... app specific behavior :-(
  - Have a plan for handling this otherwise I'll just "report an error and bypass your check"

Howto: App/APK Integrity

apkDigestSha256 and **apkCertificateDigestSha256**

- hash of the APK binary and the hash of APK signing Certificate

Easy mode:

- **APK Certificate Digest** is always the same (if always signed with same cert)
    - Can hard code into your backend (you only have one data point)

If you have this you have a form of application binary integrity via SafetyNet

Howto: App/APK Integrity

apkDigestSha256

Advanced mode:

- Collect all APK Digests and compare against database

Features:

- Your devs can sign apps but don’t control APK digest database → you control what versions are allowed to speak to your backend
- Revoke APK versions by digest

WARNING: Need to have total control over your release process!

Implementation & Deployment Summary

Client

- Check error conditions and retry, report failure codes to backend

Backend

- Validate signature and attestation data
- Check all fields including timestamp and nonce
- Tie your APIs to valid attestation responses

Make decision for failures that prevent attestation to happen (important!!!)

- Ask user to update PlayServices, have whitelisting mechanism for customers

Attacks

Can we Trust SafetyNet Attestation?

I wanted to know how far we can trust this system

- Limitations (e.g. Android versions)
- Attacks & Bypasses

You really want to know how well your security system works!

SafetyNet vs. Android Versions

- Android 4 - Android 5
  - Can’t detect boot state (secure vs insecure)
  - roots/attacks that require an unlocked bootloader work
    - With limitations…

- Android 6 and up
  - Detect boot state and fail CTS on in-secure boot!

Android 4

- No dm-Verity ➜ root can remount and write files in /system

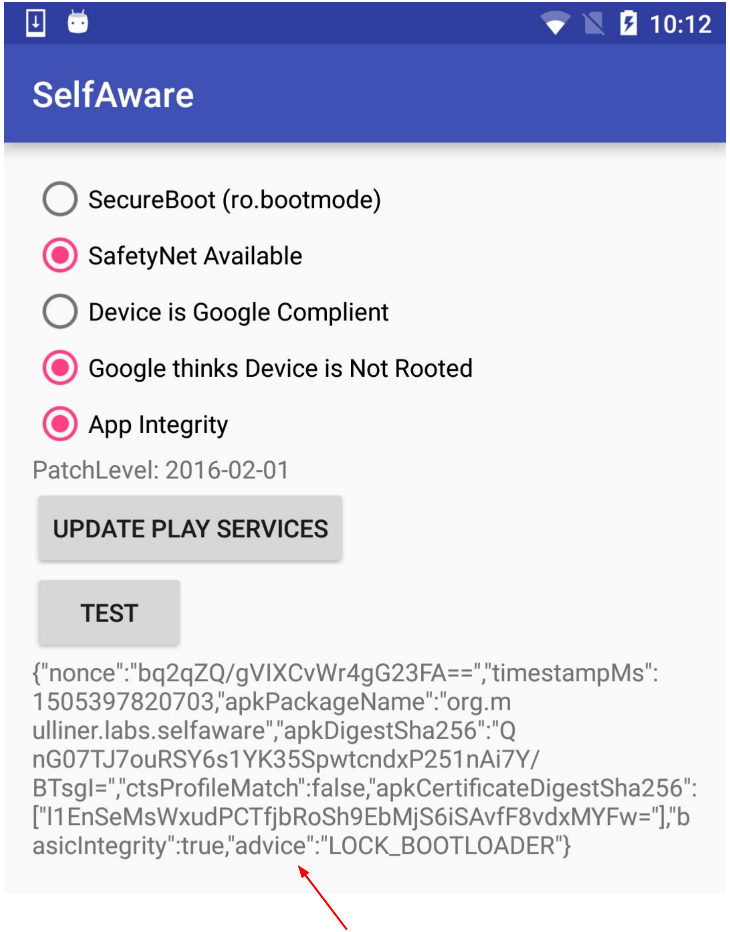
- SafetyNet Attestation inspects filesystem not running processes
  - Temp. move files such as “su” is enough to bypass it
    - Move /system/xbin/su to /data/local/tmp, run app (pass attest), restore su

Boot Loader Unlocked

Nexus 5x with Android 6

Note the advice field:

LOCK_BOOTLOADER



Client-side response validation?

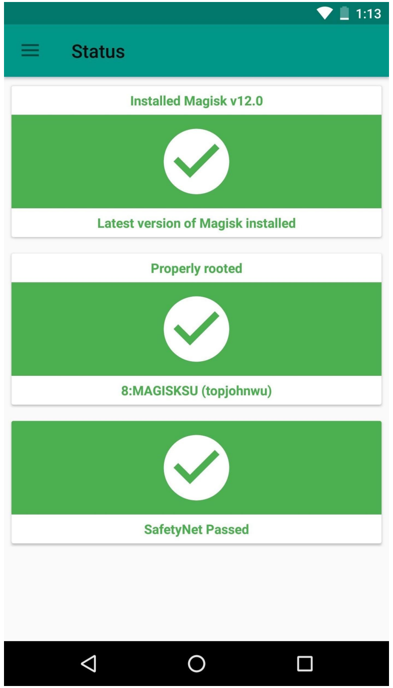
- Very easy to directly bypass
- variety of dynamic methods, xposed, frida etc
- Example: http://repo.xposed.info/module/com.pyler.nodevicecheck

```
XposedHelpers.findAndHookMethod(JSONObject.class, "getBoolean",
        String.class, new XC_MethodHook() {
            @Override
            protected void beforeHookedMethod(MethodHookParam param)
                    throws Throwable {
                String name = (String) param.args[0];
                // Modify server response to pass CTS check
                if ("ctsProfileMatch".equals(name)
                        || "isValidSignature".equals(name)) {
                    param.setResult(true);

                    return;
                }
            }
        });
```

SuHide and Magisk

- SuHide was the first attempt to hide root from SafetyNet
  - *Reference: https://koz.io/hiding-root-with-suhide/*
- Magisk is the modern root that will bypass SafetyNet
  - Based on “systemless root” (namespace hacks)
  - Cleans up filesystem namespace for specific processes like Play
  - Unlocked bootloader, selinux policy patch ➔ all this is hidden
  - https://github.com/topjohnwu/Magisk
- Need custom detections for those!
  - Google plays Cat'n Mouse
  - End-game (?): trusted hardware attestation



SafetyNet's Application Integrity Checks

apkDigestSha256 and apkCertificateDigestSha256

- Calculated on the APK file on disk

Android doesn't execute the APK

- APK contains DEX files
- Until Android 4 DEX files are converted into ODEX (optimized byte code)
- Android 4.4/5 and later DEX files are compiled into native code

This can be attacked!

(Hiding behind ART by Paul Sabanal 2014 - rootkit via odex modification)

Running Code on Android

Android 4.4 and 5

- APK: /data/app/sa.apk
- Data: /data/data/org.mulliner.labs.selfaware/
- Code: /data/dalvik-cache/data@app@org.mulliner.labs.selfaware-1.apk@classes.dex
  - Owned by system

Android 6 and later

- APK: /data/app/org.mulliner.labs.selfaware-1/base.apk
- Data: /data/app/org.mulliner.labs.selfaware-1/
- Code: /data/app/org.mulliner.labs.selfaware-1/oat/ARM/base.odex ← native code
  - Owned by system and writable by installd

Running Code on Android

Android 4.4 and 5

- APK: /data/app/sa.apk
- Data: /data/data/org.mulliner.labs.selfaware/
- Code: /da ... sses.dex
    - Owned by sy...

App can't read its own code on the disk.
Zygote loads it into memory.

Android 6 and later

- APK: /data/app/org.mulliner.labs.selfaware-1/base.apk
- Data: /data/app/org.mulliner.labs.selfaware-1/
- Code: /data/app/org.mulliner.labs.selfaware-1/oat/ARM/base.odex ← native code
    - Owned by system and writable by installd

ODEX Code Modification Attack: Overview (Generic)

- Actual code modification
  - Use apktool to unpack; MODIFY SMALI CODE; apktool to build APK; jarsigner to sign
    - Modified APK with wrong signature (but signature is not part of the ODEX file)

- Compile DEX code to ART code
  - Dex2oat --dex-file=sa.apk --oat-file=sa.odex
    - ODEX file based on <u>modified APK</u>

- Prevent the Android VM from re-compiling (aka patching the CRC32)
  - ODEX file contains CRC32 of DEX files it was generated from
  - Patch CRC32 in ODEX file to match the DEX code from the original DEX files in original APK
    - Made a tool for this!!!

Attacking ODEX files: all Android Versions

- Need to write ODEX files
  - Root device… any way to write those files will enable this attack!
- Overwrite ODEX files in dalvik cache
  - Android 4.4 /data/dalivk-cache
  - Android 6+ /data/app/APPNAME/oat/ARCH/base.odex
- Stop and start app ➔ WIN
  - Tested on bunch of 4.4 and 6 devices
- Modification persists across reboots
  - Remove root (unroot)

Attacking ODEX files: all Android Versions

- Need to write ODEX files
  - Root device... any way to write those files will enable this attack!
- Overwrite ODEX files in dalvik cache
  - Andro
  - Android 6+ /data

SafetyNet AppIntegrity is bypassed as checks are run on the APK!

- Stop and start app ➔ WIN
  - Tested on bunch of 4.4 and 6 devices
- Modification persists across reboots
  - Remove root (unroot)

Attacking ODEX files **without** Root (Android 6)

Goal: overwrite /data/app/org.mulliner.labs.selfaware/oat/arm/base.odex

Who can write?

- Users: system and installd (basically: installd and zygote)

Attacking ODEX files **without** Root (Android 6)

Goal: overwrite /data/app/org.mulliner.labs.selfaware/oat/arm/base.odex

Who can write?

- Users: system and installd (basically: installd and zygote)

Who else can write?

- Kernel ➔ **dirtycow** (CVE-2016-5195)
  - Linux kernel bug that ultimately allowed writing ANY file that you can read

ODEX file Attack via Dirtycow

Same exact procedure as before!

File size is the only issue (dirtycow can’t write past file boundary, not append!)

- Patching the APK might add code
  - Remove code? ➜ No!

Dex2Oat optimizes native code for the specific CPU
“--instruction-set=arm --instruction-set-variant=cortex-a53”

- **Trick: just don’t optimize the OAT file to make it small!**
  - I just run: dex2oat --dex-file=bad.apk --oat-file=patched.odex

ODEX file Attack using Dirtycow

BLU device with Android 6 (also tested on Nexus 5x with Android 6)

- Works on every Android device with a kernel that is vulnerable to dirtycow
    - Should be plenty of Android devices

Overwrite the odex file via:

dirtycow base.odex /data/app/org.mulliner.labs.selfaware/oat/arm/base.odex

Remember: no root required!

Attack Impact

Limited to Android devices that are still vulnerable to dirtycow

- Likely many (I don’t have numbers)

Attack obviously goes beyond SafetyNet Attestation

- Android 7 devices will not be vulnerable since dirtycow patch is required!

Notified Google over a year ago (about the generic attack), was told this is known!

CopperheadOS - hardened Android clone (www.copperhead.com)

- Mitigates by re-compiling apps before each start (can be slow)

Fun time

- SafetyNet includes DalvikCacheMonitor
- monitors cache modifications
- Iterates over dalvik cache dirs
- Finds cache files, stores hashes and timestamps, in sqlite on device
  - gms_data /snet/dcache.info sqlite
- Part of “idle” mode SafetyNet checkers
  - Runs at intervals, compares results
- Doesn’t influence attestation results
- Doesn’t check /data/app/package.name/oat/

Summary

SafetyNet Attestation improves over time

- basicIntegrity (added mid-2016)
  - Presence of su binaries in well known locations
  - Unexpected SELinux states

Collin Mulliner
@collinrm

Discovered new element "basicIntegrity: true/false" in Android's SafetyNet Attestation. Need to investigate what this indicates. #android

3:03 PM - 6 Jul 2016

- advice (added ca. mid-2017)
  - LOCK_BOOTLOADER
  - RESTORE_TO_FACTORY_ROM

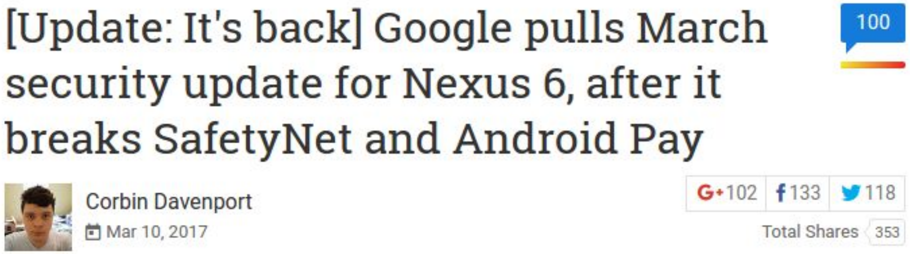
{"nonce":"bq2qZQ/gVIXCvWr4gG23FA==","timestampMs":
1505397820703,"apkPackageName":"org.m
ulliner.labs.selfaware","apkDigestSha256":"Q
nG07TJ7ouRSY6s1YK35SpwtcndxP251nAi7Y/
BTsgI=","ctsProfileMatch":false,"apkCertificateDigestSha256":
["l1EnSeMsWxudPCTfjbRoSh9EbMjS6iSAvfF8vdxMYFw="],"b
asicIntegrity":true,"advice":"LOCK_BOOTLOADER"}

SafetyNet Attestation “Outage”

- Attestation is based on CTS data
    - CTS is run by manufacturers (including Google) for each OS release and patch

- Missing or false data ➔ Attestation believes device is modified

- Google broke Attestation briefly for Nexus devices
    - I found Attestation was broken for YotaPhone with a specific security update (~1 year ago)


[Update: It's back] Google pulls March security update for Nexus 6, after it breaks SafetyNet and Android Pay

100

Corbin Davenport

Mar 10, 2017

G+102 f133 118

Total Shares 353

Proposed Improvements

- Include key & ID hardware TEE attestation
- Disassociate attest request with data collection / data send
- Increased privileges could help Snet
- Collect info via more elaborate methods
- Some more obfuscation wouldn’t be a bad idea, or using native code
    - Droidguard is much more difficult to RE
    - No reason to include original class names in debug info of renamed classes

```
.class Lcom/google/android/snet/h;
.super Ljava/lang/Object;
.source "AutoValue_SdCardAnalyzer_SdCardAnalysisInfo.java"

.implements Lcom/google/android/snet/bb;
```

Conclusion

- SafetyNet is a good and “free” way to perform device integrity detection
  - Developers who used to rely on home-rolled or library provided root detection should use it
- As is the case with all client-side security systems, it can be bypassed
  - Current bypasses are not always practical in attack scenarios
- Using it for application binary integrity isn’t ideal
  - There are better frameworks (commercial) for anti-debug & binary protection
- It’s only good if implemented securely
  - Verify result at backend, not on-device,
  - Verify crypto, nonces, check all fields
  - Don’t just run one attestation on app start, tie result to API response

Thank you - Questions?

References

Google documentation

- SafetyNet training article
- SafetyNet API SDK docs

John’s blog posts

- Inside SafetyNet part 1 – koz.io (17 Sept 2015)
- Inside SafetyNet part 2 – koz.io (20 Mar 2016)
- Inside SafetyNet part 3 – koz.io (13 Nov 2016)
- Using SafetyNet securely – cigital (09 Oct 2015)
- Using SafetyNet securely – koz.io (12 Oct 2015)

Collin’s presentation / tools

- Inside Android's SafetyNet Attestation: Attack and Defense
- https://www.mulliner.org/android/

Google SafetyNet sample app

- app & server source - github (28 Oct 2016)

Cigital SafetyNet Playground app (09 Oct 2015)

- Play Store
- Client-side source - github
- Server-side source – github