Android Proof: Authenticated Data Gathering using Android Hardware Attestation and SafetyNet

Overview

The Android Proof leverages new security guarantees offered through the use of SafetyNet Software Attestation and Android Hardware Attestation, for the provision of a secure and auditable environment whereby authenticable data can be fetched.

The Android Proof is based on a service application, which is running on a physical Android device. The Software Attestation provided by SafetyNet will be leveraged to guarantee integrity of the device - specifically, it will demonstrate that the list of root certificate authorities have not been modified. If this claim holds true, then the service application can start a secure HTTPS connection to the URL provided by the user and return the HTTP response to the Oraclize infrastructure along with the attestation result.

Service Applicability

The Android Proof service finds application in a wide range of use cases - it might be useful either for notarizing webpages or to certify data served by HTTPS APIs. A specific application of the latter is for the connection between blockchain protocols and the wider Internet - web-data might be delivered to smart contracts by attaching the authenticity proof to the data itself.

Android Proof Process

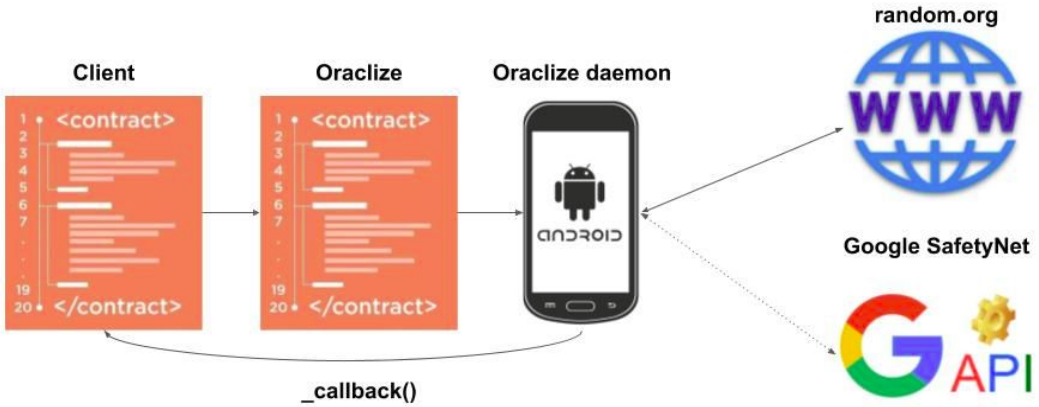
The Android Proof setup phase starts the first time the application is launched. If the device's Android KeyStore is empty, the application creates a NIST-256p key pair. The certificate chain of the created key is retrieved and sent to Oraclize's infrastructure. The leaf certificate contains the Hardware Attestation Object, which is needed to prove that the key has been generated inside the Android Hardware KeyStore. The full chain of certificates must be verified against publicly available Google-owned Certificate Revocation Lists. The Attestation Object and the certificate chain needs to be retrieved once more only after OS upgrades or security updates.

The Android Proof main operating mode follows the steps outlined below:

- As soon as an Android Proof request is made, the URL provided by the user is forwarded to the Android device. An HTTPS connection is established and the entire HTTP response is retrieved. The service application signs the SHA256 hash of the response with the hardware attested key pair present on the device.
  Note: An Android Oreo specific feature is that when the device is establishing an HTTPS connection to a server that incorrectly implements TLS protocol-version negotiation, HttpsURLConnection no longer attempts the workaround of falling back to earlier TLS protocol versions and retrying.

- The application issues a call to SafetyNet's API. The API nonce parameter is set equal to the SHA256 hash of the HTTP response, the signature and the requestID. SafetyNet then returns an AttestationResponse in the JSON Web Signature format (JWS).

- The application sends the entire raw JWS response, the HTTP response, its signature and the requestID to Oraclize's infrastructure. After a full validation of the proof, the data in the HTTP response is parsed and distributed to the user together with the SafetyNet AttestationResponse and the Hardware Attestation Object.



Oraclize's Verification and Proof Process

The actual authenticity proof consists of both the SafetyNet AttestationResponse and the full chain of certificates of the key pair used for signing the HTTP response. The Hardware Attestation Object contained within the leaf certificate is of specific interest and utility.

The proof can be independently verified by any third party by verifying all the conditions expressed in the following list:

JWS Verification

- The AttestationResponse JWS must be valid: the JWS Header contains the certificate of the key used to sign the concatenation of the encoded header and encoded payload.
- The JWS Header contains a valid certificate chain consisting of a Google issued certificate and its parent certificate issued by a known root certificate authority. Both must be checked against their respective Certificate Revocation Lists to verify that they are still valid.

SafetyNet Authenticity Verification

- Verify that the leaf certificate contained within the JWS Header was issued to the SSL hostname *attest.android.com.*
- Validate the JWS against the Google Device Verification API, which will return true if and only if the JWS has indeed been generated by Google.

SafetyNet Response Verification

- The SHA-256 of the application package included within the JWS Payload needs to match the SHA-256 hash that can be generated by compiling the released source-code.
- The SHA-256 of the certificate used to sign the application has to match the publicly known Oraclize developer certificate.
- The "*basicIntegrity"* and "*cstProfileMatch"* flags must both be true.
- The SafetyNet nonce has to equal the SHA-256 digest of the HTTP response, signature and requestID.
- The timestamp included into the JWS Payload must indicate a point in time successive to the user's request of the proof.

Hardware Attestation Verification

- The attestation certificate chain regarding the key used to sign should be valid, meaning that none of the certificates have been revoked and that every certificate is verified against its direct parent.
- The first leaf certificate public key verifies the signed SHA-256 hash of the HTTP response.
- The root certificate of the certificate chain is one of the known Google root certificates for hardware-attestation use.
- The security parameters of the Hardware Attestation Object, such as the boot integrity and patch version specified, should be the latest and most secure versions available.

Design Rationale

The Android Proof is based upon different components of the Android OS and the Google development platform. Specifically:

- Android Hardware Keystore and Hardware Attestation, both implemented in a Trusted Execution Environment (TEE).
- Google SafetyNet Software Attestation APIs

The following paragraphs will analyse these features in further detail.

Android Hardware Attestation

Android Nougat is the first version offering support for hardware attestation [5], which enables developers to generate an attestation object with a number of details regarding a precise key stored in the Android Hardware Keystore. The attestation object is an X509v3 extension present in the key certificate, and it is signed by a special attestation key kept on the device. The root certificate regarding that key should be a known root Google certificate.

The attesting key is provisioned (i.e. injected in the TEE) during the manufacturing process. For the purpose of Android Proof, a valid attestation object for the signing key should have the following features present:

- Keymaster and Attestation Security Level should have a *TrustedEnviroment* level.
- Key details should be in the *teeEnforced* authorization list.
- OS version and security patch level should be the most recent available.
- A locked and verified boot state with the correct certificate hash, corresponding to a known OEM root certificate.
- Cryptographic characteristics such as the key size, algorithm used, key purpose, creation and expiration date should be the expected ones for the key pair.

It's important to know that since Android Marshmallow, Android features a *Root-Of-Binding-Trust (ROT)*. During device start-up, the TEE is initialized with the public key used to sign the partition and the current device state: locked or unlocked. Any change to those parameters will lead to a different initialization value and a different Hardware Keymaster ROT; making all previously generated keys useless.

If the keys are still usable at any point in time, the presence of the Hardware Keymaster ROT can guarantee that the device bootloader is still in a verified and locked state. Consequently, there is no need for the Android Proof to reproduce the attested key pair certificate chain each time it receives a request, being the one produced during the key pair creation enough. In other words, if the application is able to produce an HTTP response signed by the correct key, this means that the device is still in a secure state.

With Android Nougat, the Root-Of-Binding-Trust (ROT) has been enhanced by including the OS version and security patch level. Therefore, neither security updates nor OS upgrade rollbacks can happen without invalidating and rendering the keys stored within the TEE unusable. Furthermore Android Oreo does not even allow the device to boot up from such a downgraded and less secure version of the operating system.

SafetyNet Software Attestation

SafetyNet is a Google service for Android developers. The main use of the service is to discover whether an application is running on a device that is in a *tampered state*, including being rooted, and act on the SafetyNet result accordingly to the application's security policies. An example is given by AndroidPay, which uses SafetyNet to determine whether the device is rooted and disables the application in case of a positive result.

To recap, SafetyNet is used in the Android Proof to attest:

- The SHA-256 hash of the application package running on the device. The main goal is to demonstrate that the application sending the SafetyNet request executes the same code implemented by the open-source application that is publicly available and distributed. This is to be able to guarantee that no alteration have occurred on the HTTP response before it is signed and its hash used in the SafetyNet request.
- The SHA-256 hash of the developer certificate used to sign the application package.
- That the system is not in a tampered state, and specifically that the root CAs have not been modified. SafetyNet performs also as a basic root detection system.
- That both the HTTPS request and the signing process using the attested key have taken place in the application issuing the SafetyNet request.

According to Kozyrakis ([3], [4]), SafetyNet performs a series of checks on the device and it uploads to Google servers for analysis purposes. The particularly relevant checks are in regards to the HTTPS connection (*ssl redirect*, *ssl handshake*) and *SystemIntegrityChecker,* which calculates a HashTree over */system*. The hash is checked against a hash table of known devices configuration on Google servers. The device root certificate authorities are contained in the folder */system/etc/security/cacert*s, so that any modification to the certificate authorities themselves would trigger a failure of the *SystemIntegrityChecke*r. In the JWS Payload of the SafetyNet Response, this check is returned as the value of the variable *basicIntegrity*.

Additionally, Android Nougat standardizes the local list of root CAs across devices and forces the applications needing custom CAs to explicitly include them in their configuration files.

SafetyNet API requires a nonce to be set for each issued request. Android Proof will use the SHA256 of the HTTP response, signature and requestID as the SafetyNet nonce. The obtained hash should be unique, as recommended by SafetyNet guidelines.

After having performed the analysis, SafetyNet API will return a result in the form of an AttestationResponse object. The format chosen for the result is the JWS Compact Serialization format [8]. The JWS is represented as the concatenation of the *Base64URLSAFE* encoded:

- JWS Header
- JWS Payload
- JWS Signature

The JWS Header is a JSON Object containing two items: an "alg" and a "x5c".

The first is a key-value item, indicating the signing algorithm used by the JWS. In the case of SafetyNet, the value is "RS256", which refers to RSA and SHA256 based signing algorithms.
The second item is a JSON Array, having as elements the leaf certificate (the one corresponding to the key used to digitally sign the JWS) and its parent certificates.

The JWS Payload is the *Base64URLSAFE* encoded Safety Response. The decoded payload has the following structure:

```
{
	"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”
}
```

The JWS Signature is the signature generated by the RSA signing algorithm on the SHA256 hash of the encoded JWS Header concatenated with the encoded JWS Payload.

All proofs and attestations can be verified by using Oraclize-developed open source verification code.

Security

While designing the architecture of the Android Proof, the following assumptions related to security were made:

- The Trusted Execution Environment (TEE) used on the device is secure: private keys cannot be leaked even in the event of a complete kernel compromise.
  Therefore, the Hardware Attestation enables us to prove that:
  - The private key of our key pair is truly generated and contained in a TEE.
  - The device bootloader is verified and locked.
  - The device is updated to the most recent security patch level and OS.
- SafetyNet is sophisticated enough to offer good guarantees that the device is not rooted or, at the very least, to attest that the root certificate authorities have not been modified and that a secure HTTPS connection with the datasource is possible.
- The Android App Sandbox model is secure, and applications cannot tamper with memory or data, which do not belong to them. This is one of the cornerstones of the Android OS.
- Every time the device is updated with a security patch or an OS update, the Hardware Attestation Object needs to be re-extracted in order to prove that the device is running the most up-to-date and secure software.

In the last year a number of high-profile vulnerabilities regarding the Android platform have been discovered by security researchers. Some of these vulnerabilities enabled escalation of privileges by an attacker and full compromise of the device kernel. The use of these critical vulnerabilities alone does not lead to attackers to gain control of the device's TEE, which is separated and independent from the main system. Unfortunately, not all implementations of TEE are strong and safe: CVE-2015-6639 and CVE-2016-2431 are two vulnerabilities found in the Qualcomm TEE implementation (QSEE) [6], [7].

The use of Android Nougat, SafetyNet, and newer devices therefore provide us with additional security guarantees due to the following reasons:

- Android Nougat and Oreo have introduced a number of additional characteristics that protect the device's security [1], [2], [9].
- Exploiting the aforementioned vulnerability still wouldn't allow an attacker to bypass SafetyNet, which gathers data locally and analyzes it remotely on Google's infrastructure.
- Newer devices, such as the Nexus 5X and Nexus 6P were not vulnerable to recent TEE attacks. A Google Pixel 2 phone is currently being used for the production environment.
- Lastly, if a TEE implementation is compromised and attestation keys are leaked, Google can revoke the attestation key by placing the corresponding certificate in a revocation list. This consequence will happen as soon as those violations are intercepted by Google, through SafetyNet or other channels.

Scalability

Currently the main limitation for the scalability of the system is Google's SafetyNet API which has a default quota of 10.000 requests per day for each API key [10]. Google offers the possibility to increase this quota through a form request available on their platform, to avoid blocking of the system in the occasion of an unexpected spike or getting notifications before reaching that limit, a few strategies to

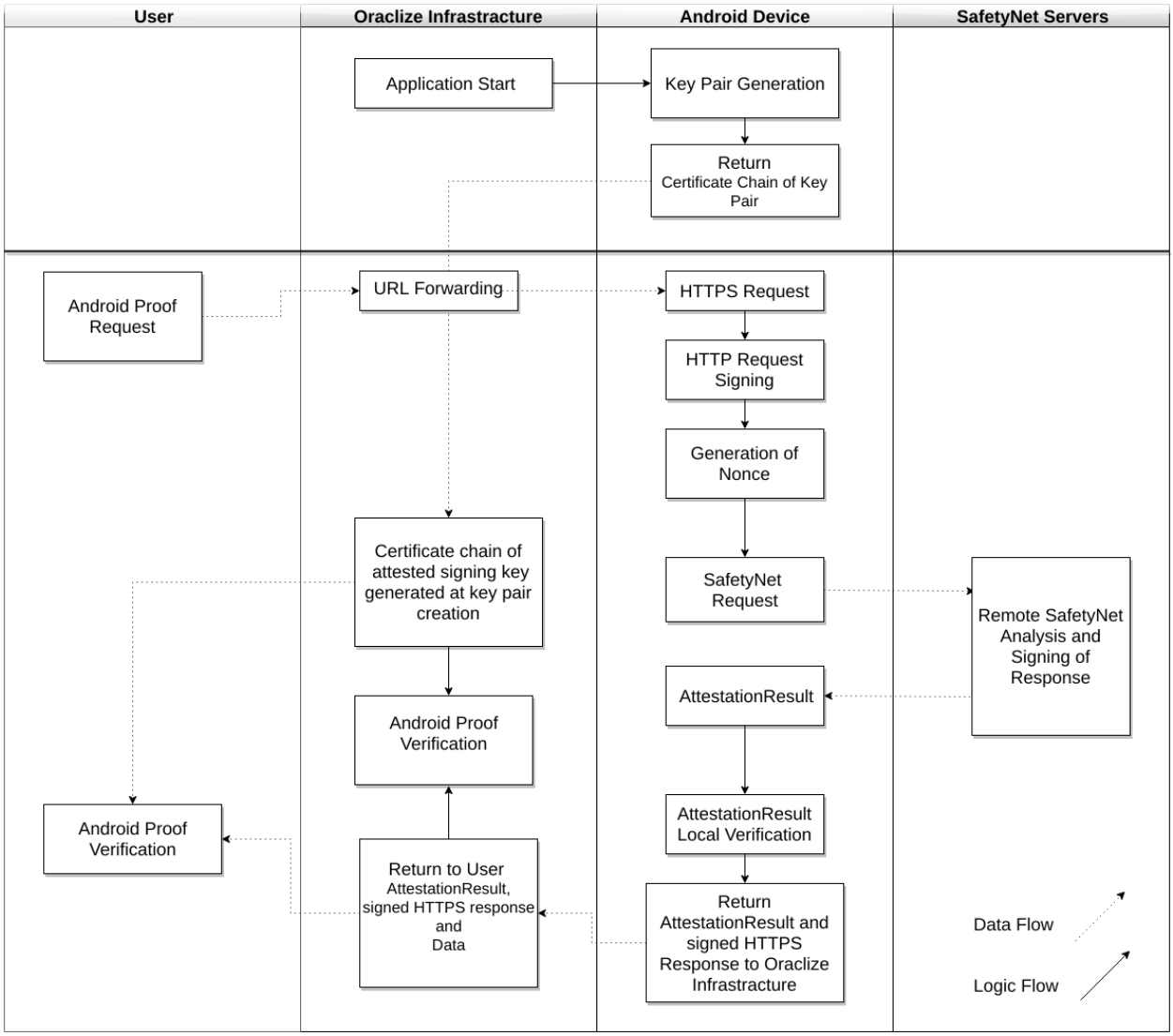
prevent and handle these situations are being considered.

Google offers a Dashboard on their platform to monitor API calls, it's attached to the API Key sent as a parameter for the SafetyNet API. The use of a system with rotation of multiple API keys could be an option if there is no limitation based on IP address or Key owner. If such limitation exists there's the possibility of requesting from the client to provide an API Key to be used on the SafetyNet request, in this case an internal monitoring system can be used to manage the quota.

Another possible bottleneck in the system is the dependency on a single Android device to handle all the SafetyNet calls. On internal stress testing with one device and sequential processing, the system was able to produce on average 7000 AndroidProofs per day. Considering that there are many conditions that can cause an attestation to fail, such as network connection, server availability and other similar problems, the configuration was optimized to produce the maximum positive results with an average of 8s to complete each request.

Future works might consider multiple devices integration and parallel processing of Android Proofs but no known issues have been identified that could prevent this configuration.

Appendix A: Graph



References

1. Austin D. and Vander Stoep J. (2016, May 05). "*Hardening the media stack*". Retrieved November 22, 2016, from https://android-developers.blogspot.it/2016/05/hardening-media-stack.html
2. Xiaowen Xin (2016, September 06). "*Keeping Android safe: Security enhancements in Nougat'*. Retrieved November 22, 2016, from https://android-developers.blogspot.it/2016/09/security-enhancements-in-nougat.html
3. John Kozyrakis (2015, September 17) "*SafetyNet: Google's tamper detection for Android"* ~ blog. Retrieved November 22, 2016, from https://koz.io/inside-safetynet/
4. John Kozyrakis (2016, March 20) "*Inside SafetyNet - part 2"* ~ blog. Retrieved November 22, 2016, from https://koz.io/inside-safetynet-2/
5. Key Attestation. (n.d.). Retrieved November 22, 2016, from https://developer.android.com/training/articles/security-key-attestation.html
6. L. (2016, May 02). "*QSEE privilege escalation vulnerability and exploit (CVE-2015-6639*)". Retrieved November 22, 2016, from https://bits-please.blogspot.it/2016/05/qsee-privilege-escalation-vulnerability.html
7. L. (2016, June 15). "*TrustZone Kernel Privilege Escalation (CVE-2016-2431)"*. Retrieved November 22, 2016, from https://bits-please.blogspot.it/2016/06/trustzone-kernel-privilege-escalation.html
8. *"RFC 7515 - JSON Web Signature (JWS)".* (n.d.). Retrieved November 22, 2016, from https://tools.ietf.org/html/rfc7515
9. Gian G Spicuzza (2017. December 20). *"Double Stuffed Security in Android Oreo".* Retrieved March 4, 2018, from https://android-developers.googleblog.com/2017/12/double-stuffed-security-in-android-oreo.html
10. Oscar Rodriguez (2017. November 13). "10 things you might be doing wrong when using the SafetyNet Attestation API", from https://android-developers.googleblog.com/2017/11/10-things-you-might-be-doing-wrong-when.html