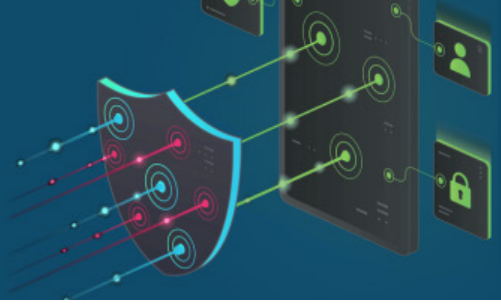
Best Practices for Secure Access of Third-Party APIs from Mobile Apps

approov

WHITE PAPER



Best Practices for Secure Access of Third-Party APIs from Mobile Apps

Contents

Introduction

Mobile apps typically communicate with several backend services over the Internet. In some cases those services may have been developed specifically for that mobile app, but in most cases the app will be dependent on services developed by a 3rd party. This whitepaper discusses how communication with those services can be properly secured, and how Approov Runtime Secrets Protection enables significantly enhanced security to be added with a minimum of development effort.

The Problem with API Keys

API keys are commonly used to secure access to a backend API. Typically, when you use a third-party API you sign up and you get allocated a key. This is essentially like a password. It authenticates the app that's making the call so that the API backend knows which particular app is making the request. It enables the blocking of any anonymous traffic and can be used to limit requests. There is normally a quota associated with a particular API key and there might also be some permissions associated with it too. So although it's an identification mechanism, in practical terms you do have to consider the API key to be a secret. If that API key becomes known then your app can be impersonated and API calls can be made by other parties, just as if your app was making them.

There's an ongoing debate about whether API keys should be used at all. Mechanisms exist which allow you to use access tokens instead. One strategy is to have an access token which is associated with a particular app and a particular user, and the access token is obtained when the app is first installed. This can also be associated with user authorization and there are ways in OAuth2 to allow this. The app performs an authorization step on installation or first use and then obtains an access token that could be used to access an API. Note, however, that most APIs do not support this approach.

This method provides some value because it ties the access down to an individual user. However, the problem is that for many applications it's actually quite easy to enroll as a user. If obtaining an access token is just based on user authorization then it's not actually authenticating the app. In that sense it is a very poor protection against abuse of the API itself. At this point you might consider using a client secret in order to protect the initial request for an access token. But then you are back to the same problem as before. Instead of having an API key you have a client secret that needs to be protected, with all the same issues.

So what can go wrong if API keys are exposed? This is in fact a general problem, it is not just related to mobile apps, it is also a risk with server to server communication. There have been various cases where developers have accidentally checked in API keys into the source code. Various tools exist that can trawl public source repositories, find those API keys and then abuse them. This is an issue that has been discussed quite a lot. There have been particular attacks associated with this and in the last couple of years we've seen more and more tools that enable you to find API keys that you might have accidentally included in source code. There are also secret management storage solutions in the market that allow you to separate secrets so that they're not stored in your source code at all.

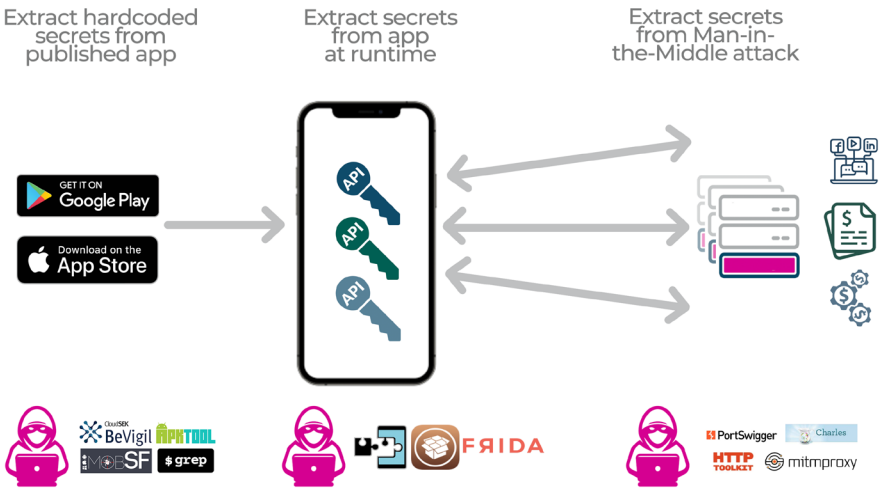
In the case of mobile apps, though, the compiled code is in the public domain. Although you may have kept your API keys out of your source code, those keys are still compiled into the release. API keys and other secrets can be reverse engineered from the code, or captured in transit from the mobile app to the backend. Any attacker has complete control of the device and the network it uses, so such attacks are relatively straightforward. Here is a recent report which highlights how easy it is to find and exploit API keys in top financial services apps using some of the techniques described in this whitepaper.

Third-Party API Access from Mobile

Consider the case where you have compiled your app and published it to the app store. It has embedded API keys or other secrets inside it and then as the app runs it will use those API keys to access whatever APIs your app is depen-

dent on. The above diagram shows examples of third-party APIs that an app may be using. This survey indicates that individual apps are now dependent on a large number of 3rd party APIs. Some 45% of apps store 3rd party API keys, with an average of 40 APIs per app for those using in-house development. Whenever it accesses that individual API, that API key is copied from the code as it runs and then transmitted over the network.

There are various vectors that can be used to extract those secrets from the mobile app,and some example tools shown in the diagram below.



One possible approach is direct extraction from the published app code. There are various tools for this, MobSF being a great example. It's a static analysis tool which can take an APK, or an iOS IPA, and reverse engineer it. You can use this for code extraction too but it integrates various analysis patterns it uses to automatically find sequences in the code that look like API keys and secrets. They tend to be quite easy to locate because they have certain well defined characteristics and high entropy. MobSF presents a list of potential API keys that it has found in the app.

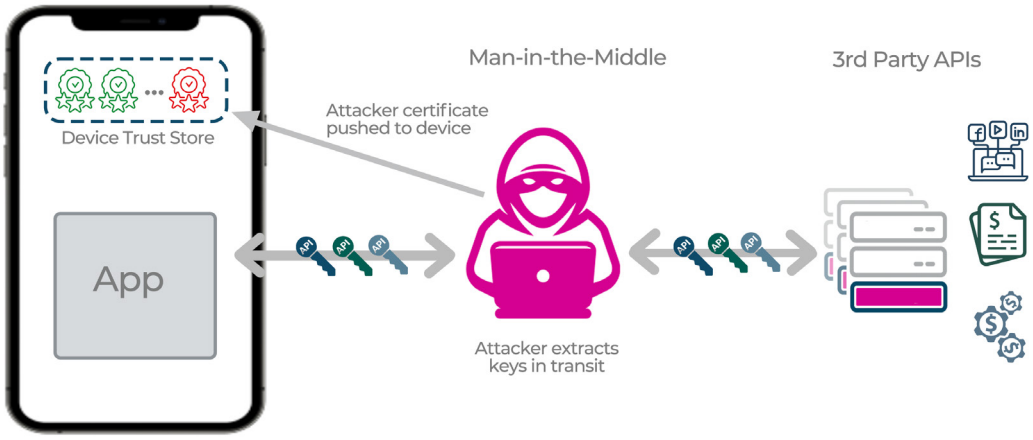
There is also a really good tool called BeVigil which is a search engine that performs a similar function. It also allows you to see API keys and other secrets. There are also command line tools you can use, like Apktool, to reverse engineer Android apps. Indeed, just running strings on binary images will often find API keys. Once you have those API keys you can then make a direct access in a script.

At runtime the API keys are going to be held in memory. There are also various code hooking tools that can be used on rooted or jailbroken devices, such as Frida. It can hook and intercept particular function calls inside a running app. This

can then be used to collect the API keys at runtime.

Finally, API keys or other secrets can be extracted by a Man-in-the-Middle attack. There are numerous tools like mitm-proxy and Burp Suite that allow you to do this.

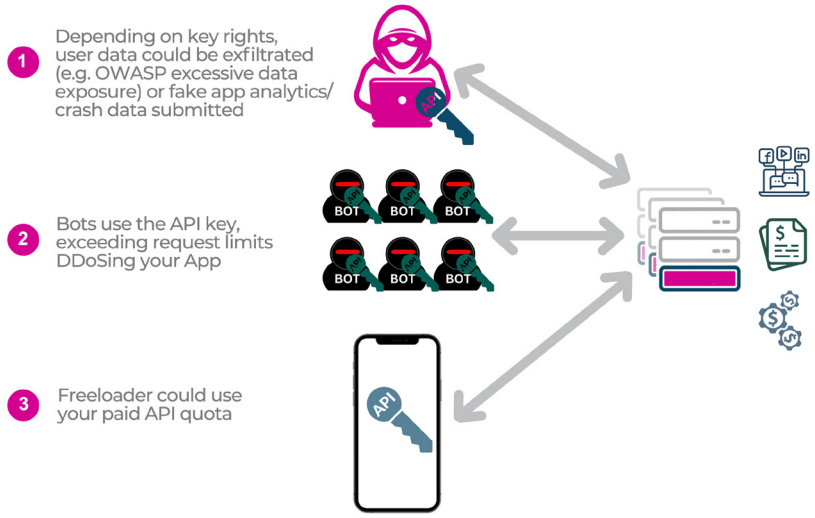
How a Man-in-the Middle Attack Works



When an app makes a connection to a backend API it does that over an encrypted TLS channel. The TLS negotiation verifies the certificate presented by the API backend. The server's certificate must be part of a chain of trust, leading to a root certificate which is trusted by the device. The device itself stores a set of root certificates and all certificate trust chains must lead to one of them. This maintains the integrity of that channel, proving that the backend identity is as expected. Once the communication is established over TLS it's not possible for a 3rd party to decrypt that traffic.

However, using a proxy tool it is possible for an attacker to insert themselves between the app and the backend API. If the device is rooted or jailbroken then they can push a self-signed certificate from the proxy tool onto the device as if it is one of those trusted root certificates. The app will then happily connect to what it thinks is the actual backend API where it's actually a Man-in-the-Middle (MitM) impersonating that backend API. The requests communicate through the MitM to the actual backend API, but now the attacker can see all the traffic including any API keys. These can be easily extracted and then used outside of the app.

The Risks of Stolen API Keys



The risk obviously depends on the capabilities of the particular API whose key has been compromised. It's possible that having access to an API outside of the app allows some data exfiltration; either directly or indirectly through vulnerabilities like OWASP excessive data exposure. This is a classic API implementation blunder whereby an API may present more information than it really should, but relies on the client to filter what is made available. Once you have the API key you can access the API directly, perhaps running it in Postman or similar, and therefore see all the data.

Another possibility is that there are API keys associated with crash or installation analytics. You can use that to impersonate the app and post incorrect and misleading data. That undermines the analytics, or causes support teams to waste time tracking down seemingly widespread phantom bugs.

Generally if you are using a 3rd party API then there is going to be some kind of quota associated with it so you can't make so many API calls that the quality of service of others users is degraded. This might be expressed as the total number of APIs calls that may be made per day, hour or minute associated with a particular API key. If that API key is extracted and distributed bot scripts start making API calls rapidly then your quota will be quickly exceeded. In effect this is a Distributed Denial of Service (DDoS) attack. The cost of doing this for the attacker is very low. Calls from the real app itself will be rejected and it won't be able to function because it will be unable to get access to backend APIs.

If you use some 3rd party APIs that are on a pay-per-use basis then the attacker's motivation might be to be able to make API calls without paying. But you will be paying for them!

Existing Protection Techniques

1 Obfuscation

Code obfuscation and hardening makes it difficult to reverse engineer and extract secrets statically.

But doesn't protect them against runtime/MitM extraction.

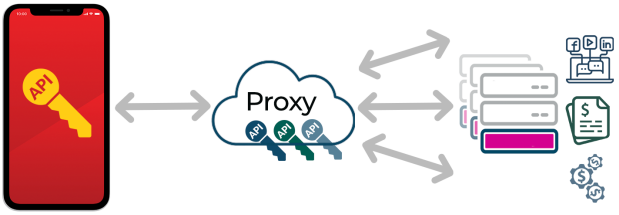
2 Pinning

Sometimes used in conjunction with obfuscation.

But pinning can be defeated with hooking tools, so ineffective without very strong RASP defense.

3 Proxy

App uses a different API Key to communicate via proxy so 3rd party API keys not exposed.



Adds latency costs and resilience risks and doesn't solve issue as the key to the proxy can itself can be extracted and abused.

Obfuscation or code hardening provides protection against static reverse engineering. The first key extraction technique we discussed was to download the app from the store and then use tools to statically reverse-engineer keys out of the app. If the app code and configuration are obfuscated then it becomes much more difficult to do that because those keys should be protected at rest. So this provides good protection against the first line of attack. But the problem is that it doesn't protect it directly against the runtime or Man-in- the-Middle extraction attacks. Unfortunately, protection is only as good as the weakest link.

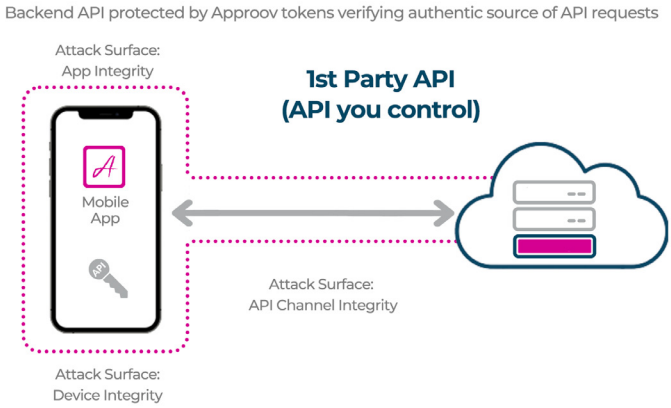
Pinning is also employed in apps as a way of locking down connections to specific known certificates associated with the backend API servers. This prevents Man-in- the-Middle extraction being used easily. However, even if your app uses pinning, then it is still possible to defeat it if you use certain hooking tools running inside the app on rooted or jailbroken devices. Therefore you also need to have Runtime Application Self Protection (RASP) or App Shielding protections running inside the app to prevent this.

Another approach is to move the API keys out of the app entirely by communicating with the backend via a proxy rather than directly to the API. The app authenticates itself to that proxy and then the requests get forwarded on to the actual backend API with the necessary API keys being added at that point. Only the proxy itself knows the real third-party API keys and relies on the authentication of the app to enable them to be used in the forwarded requests. This has its issues though.

Firstly, you are adding some complexity and latency to all the requests. Secondly, you are adding costs because all the requests are going through your proxy. Thirdly, you have to maintain and worry about the resiliency of the proxy because if the proxy goes down then nothing works.

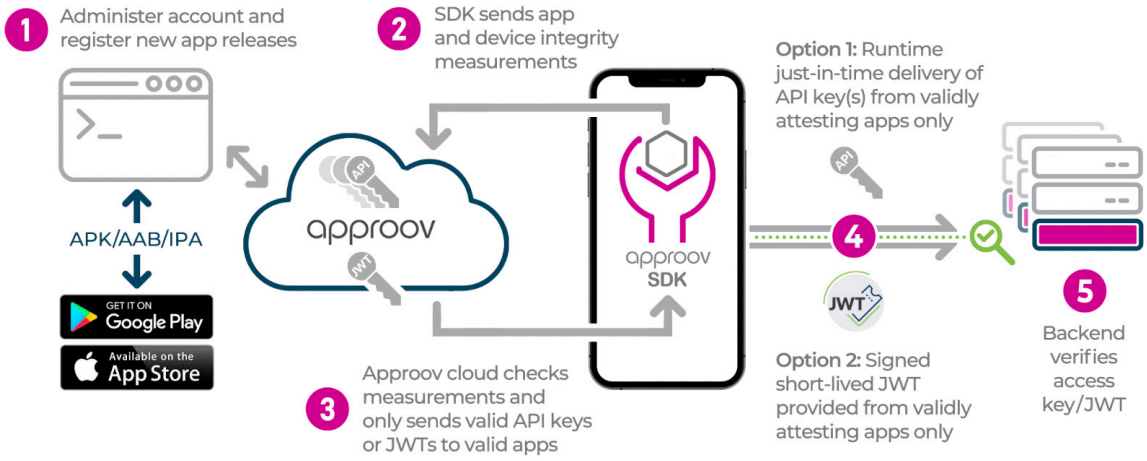
But perhaps the single biggest shortcoming of the proxy approach is that it simply moves the problem rather than really solving it. How do you authenticate the app against the proxy? You're back to the original problem of needing some kind of API key or secret in the app that can be extracted. With this an attacker can simply access the 3rd party APIs via the proxy, and DDoS it by using up its quota if they wish.

First-Party API Protection with Approov



Over the last few years Approov has been focused on protecting first-party API - i.e. the APIs our customers control directly. Approov provides protection for the various attack surfaces that can be used against your app. It provides proof of app integrity. Rather than using an API key to prove the presence of a particular app, it uses a short lifetime cryptographically signed token that is unique to each running app instance. The Approov SDK agent running inside an app on a user's device also checks that nothing nefarious is happening, in terms of function hooking that may be exfiltrating data from your app. It also hardens the attack surface of the API channel itself, ensuring that no MitM attack can occur.

Approov Architecture



The above diagram illustrates the five-step flow of how Approov API Protection works for first-party API. The app is developed as normal but at some stage the Approov SDK will be dropped into it. When a new app is released to the appropriate app store the Approov command line registration tool should be used to add that particular version of the app so that Approov recognizes it as a valid version of your app. This tool also allows various other account parameters to be set, such as your chosen security policy.

The other major component of the Approov solution is a cloud service. The Approov SDK in your app integrates into the networking stack that you are using. When your app runs and makes an API call this will automatically make a call into the Approov SDK. This will then send heavily defended measurements about the app environment and device environment to the Approov cloud. Those measurements are evaluated against your criteria, making sure it is the valid registered app and examining the data for the characteristics of rooting, jailbreaking, or other evidence indicating the app and/or device may be compromised. The Approov cloud will then make a decision and will send a short-lived JWT (JSON web token) back to the Approov SDK. It may either send a valid JWT back if it's happy and everything seems okay. If the criteria are not met then the JWT sent back will not be correctly signed.

The JWT is automatically added as an additional header to all the networking requests being made. Moreover, those requests are made over a pinned version of TLS to make sure that there is no Man-in-the-Middle and communication really goes to the backend API.

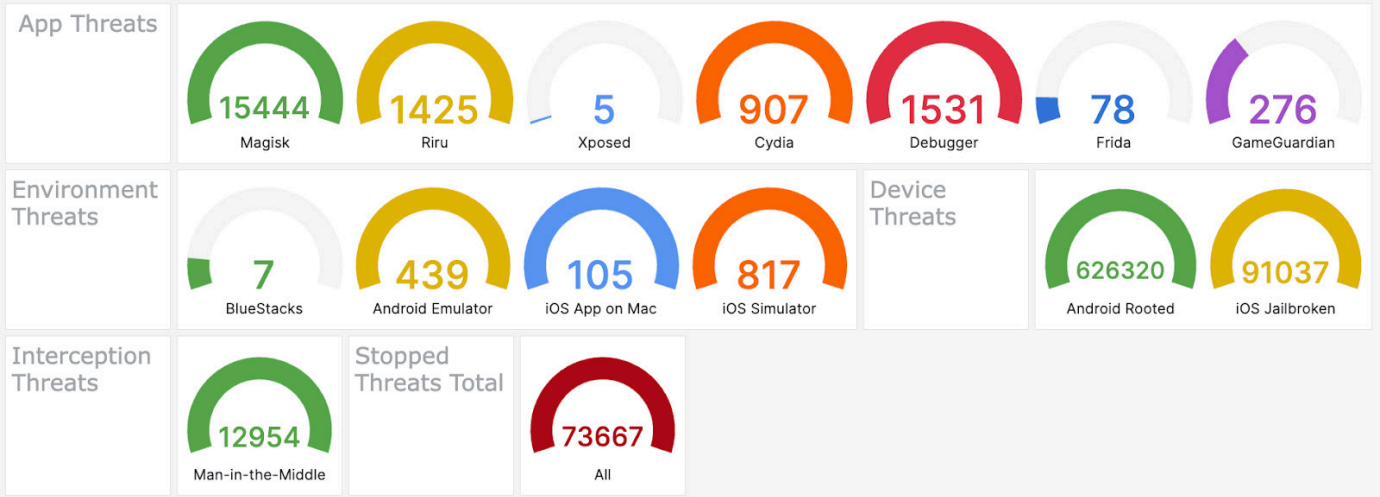
The backend API needs to check that the received JWT is valid. Only requests with a valid Approov token, signed correctly and not expired, will be let through. This is done by checking its signature using the unique signing key for the Approov account. This can either be a symmetric signature, or an asymmetric one with a public and private key. Note though that the key is never published inside the app. The only way to get a valid Approov token is via successful interaction with the Approov cloud.

The JWTs are short lived, typically with a maximum lifetime of 5 minutes. If the app continues to make requests beyond this then the integration automatically goes through the attestation process again to obtain a refreshed token.

App and Device Environment Checks

Approov performs a wide range of app and environment checks as part of the attestation process. Moreover, we are always adding to these as Approov is under continual development against the latest attacks.

The diagram above illustrates the breadth of these, including detection of rooting, jailbreaking, hooking frameworks, emulators, simulators, and various things that might be going on inside the memory map that indicates that data exfiltration may be occurring, whether a debugger is attached or checking for cloned apps.



The interaction with the Approov cloud has a side benefit. Approov sees all the connections from different devices and account holders get live and cumulative metrics showing what's actually happening in the app environment. This illustrates levels of activity and also failures and the reasons why particular devices are being rejected.

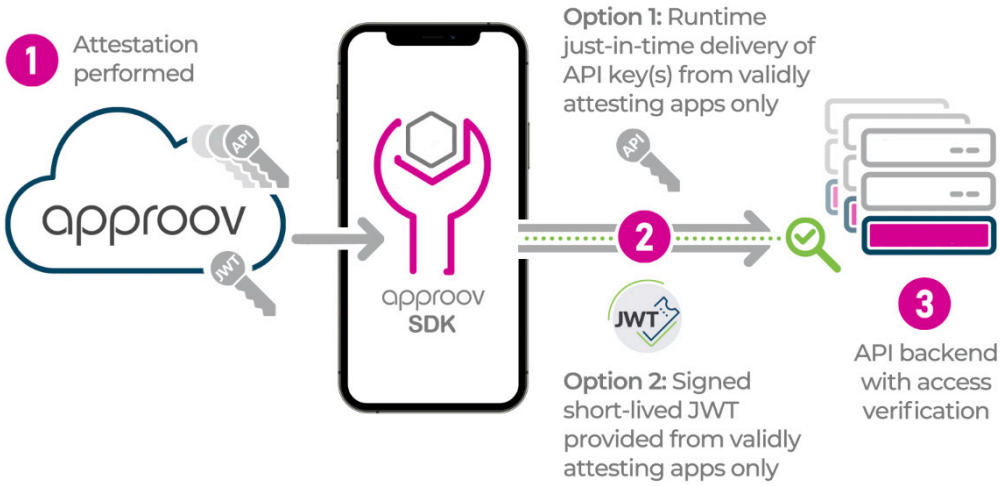
But What About 3rd Party APIs?

This is all great for first-party APIs where you control the backend API. Due to the use of the JWT it is necessary to update the backend API to do the token verification.

However, there remains a challenge around applying this to third-party APIs. If your mobile app is communicating directly then the access is probably authorized via an API key. You can't modify it to add an Approov JWT check. Furthermore, certificate pinning can't be added easily because your app will then require the presence of a very specific certificate on the backend. Unfortunately if the API is not controlled by you then that certificate could be changed at any point, preventing your app from making connections.

To circumvent these limitations, Approov is used by some customers as part of a proxy solution as well protecting a first-party APIs. The Approov token can be used to verify that the access is really coming from the authentic app and then requests can be made out to the third-party APIs using the secret API keys. Of course this introduces the latency, cost and resiliency concerns alluded to earlier, but these are customers who already have a dependence on their own backend service and an engineering team to support it, so this is a lesser concern for them.

We are aware though that there are numerous mobile apps which have no custom backend at all, they are only using third-party APIs. Thus the mobile app developer doesn't necessarily want to deal with the backend aspect at all. Approov Runtime Secrets Protection elegantly addresses this problem.



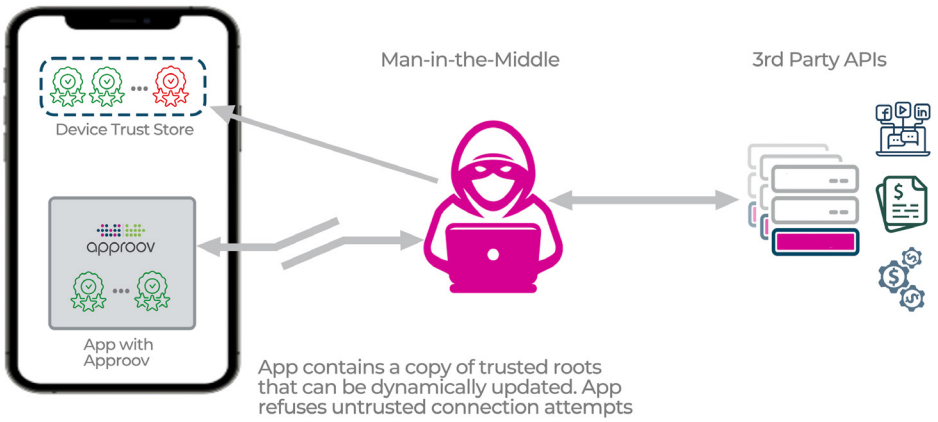
Runtime Secrets Protection provides a slightly different model of using Approov, providing all the protections discussed previously, while adding third party API protection. You must still go through the same process of including an Approov SDK in your app and registering your app when you release it. You must also register your app secrets and API keys with the Approov cloud, where they are securely stored. You can then easily migrate them out of your app build environment, so that they are no longer present in the built app code at all. Thus the app is no longer subject to any reverse engineering risk since there are no keys to steal from the published form of the app.

The Approov integration operates in the same way. When a network request is made it performs app and integrity measurements and sends the results to the Approov cloud. It is only then, after the app has passed its required checks, that the API keys or other secrets are transmitted back to it. Moreover, they are sent in a heavily obfuscated form and only stored in memory in this protected fashion. They are then automatically added to the network requests that require them, which are also further protected against any possibility of them being stolen in transit.

If the app doesn't pass its checks then the keys are never sent to it. An optional mechanism is provided to inform the user that this isn't actually a proper instance of the app or that it's running on a compromised device.

The bottom line is that with Runtime Secrets Protection in place only valid app instances running in uncompromised environments can access the API keys and secrets stored in the Approov cloud. No changes to the backend APIs themselves are necessary to achieve this.

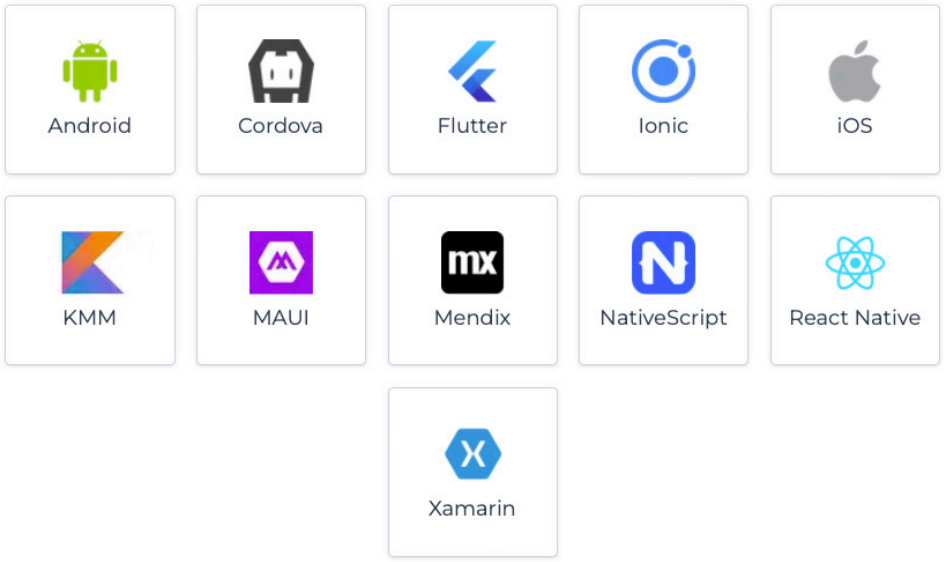
Managed Trust Roots



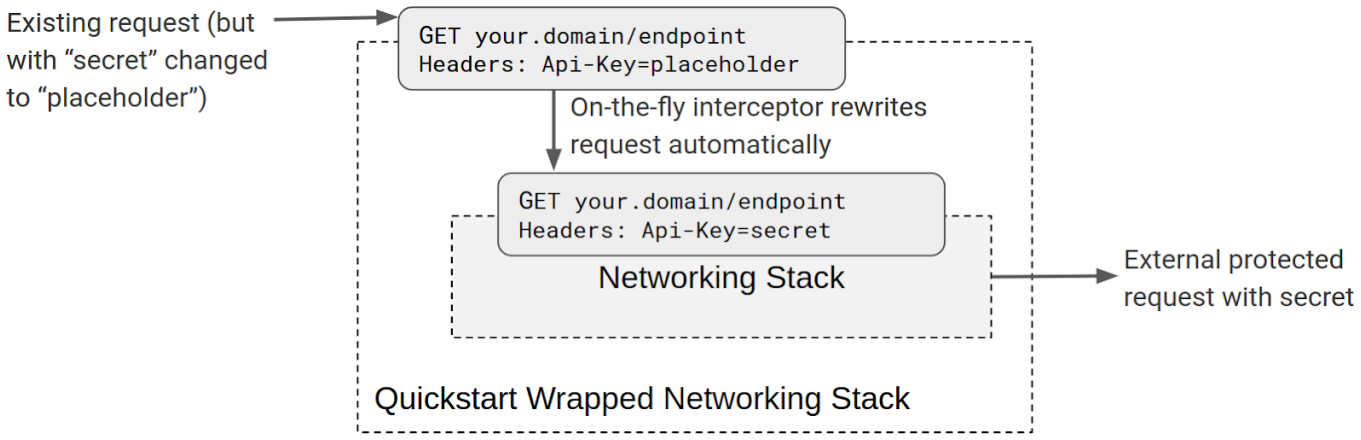
As discussed, the device trust store is potentially subject to compromise. But we can't directly pin against a particular certificate in a 3rd party backend API because they are subject to change at any point. The way Approov deals with this problem is to provide the capability for another trust store held within the app itself. This approach is termed "Managed Trust Roots". With this in place the app will only make the connection if the certificate chain root is in the device trust store ***and*** it also in Approov managed trust roots inside the app. Therefore the Man-in-the-Middle attack doesn't work because there is no way to push a self signed certificate into the app's managed trust roots, as there is with the device trust store.

The managed trust roots are dynamic so they can be changed as needed inside the app by Approov. If particular certificates need to be removed or newly issued certificates added then this can all be done without any need to update the app itself. Crucially though, these updates are only accepted if signed by Approov so an attacker cannot use this mechanism to subvert the trust store.

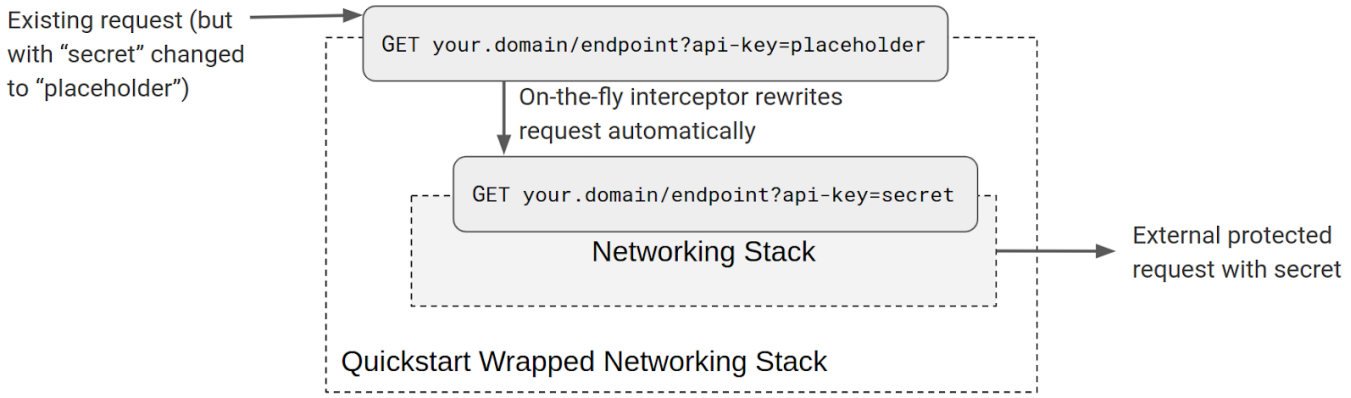
Quickstart Integrations



In order to make the integration Approov straightforward we have a range of app quickstart integrations. Within native Android we support a range of different networking stack options, such as OkHttp, Retrofit, Volley, GRPC and plain Http URL connection. For iOS native we support NSURLSession, URLSession, AlamoFire, GRPC and Async Http. Please contact us if you would like support for something else.

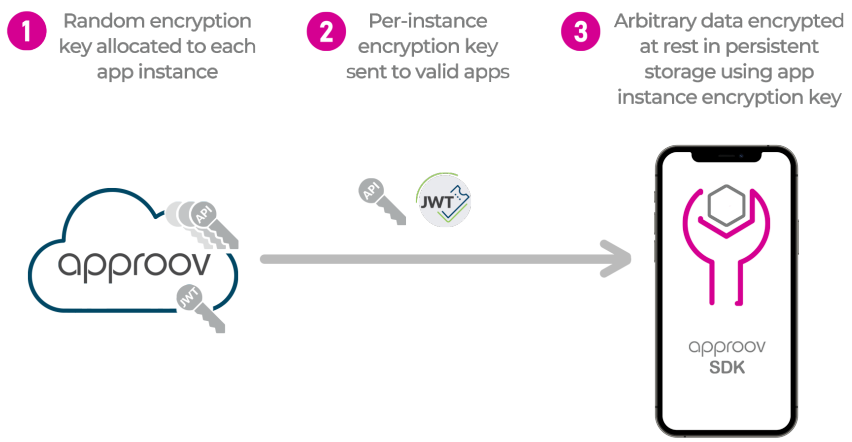


We make integration as simple as possible by providing a networking stack that has exactly the same interface as the standard one, except that it adds runtime secrets substitution. If you currently have a secret or API key then you simply need to swap it out for a placeholder value instead. This is just some innocuous string that's of no direct value. The Approov networking stack performs interception on the fly. If you are providing an API key as a header then it will detect the placeholder API key and then make an authenticity check with Approov. If the app passes then it makes the replacement automatically, replacing the placeholder with the secret for the actual network request. The secret is only available transiently on the network request. It is normally held in a protected form in memory. Every request causes Approov to make some authenticity checks, and a complete integrity check is performed every five minutes if the app is still being used.



Some APIs provide API keys as a query parameter on the URL rather than as a header, and this is also supported. The placeholder value can be provided as a query parameter and this will be rewritten on the fly for valid apps.

App Instance Secure Strings



There's also a further facility that Runtime Secrets Protection provides. It is also possible to store per-app instance secrets. Each app instance is randomly allocated an encryption key that can be used to secure locally persisted data. This key is only securely transmitted to the app if it passes the Approov checks at runtime. Primitives are provided in the integration to read and write strings that are encrypted using this key. This means that if the device becomes compromised then it will no longer be furnished with its decryption key by Approov, so any data persisted locally will be safe from compromise. This capability can be used if you have a long-lived access token that needs to be persisted across app restarts, and can't therefore just be held in the memory.

Summary

Approov Runtime Secrets Protection frees you from needing to hard code API keys or other secrets in the mobile app itself. Concerns of reverse engineering and abuse are removed. Strong app attestation protection ensures that only an authentic app running on an uncompromised device can access them, and thus the APIs that they protect. The TLS communication channel is protected with managed trust roots, providing an additional trust store within the app. Moreover, this is all enabled without any need to change the backend APIs and with a straightforward drop-in app integration approach.

A further significant advantage is that because the secrets themselves are migrated into the cloud they can actually be updated at any point. Static secrets, that could only be changed by republishing the app, become dynamic. If they need to be changed for some reason or they are breached in some other way then they can be changed on the fly. Literally five minutes later all the running apps that are running will be using the new version of the secret.