HITB SECCONF SIN-2021

Is Attestation All We Need? Fooling Apple's AppAttest API

Igor Lyrchikov | H_D

Mobile Security Expert, Thales DIS

TRACK 1

$Whoami

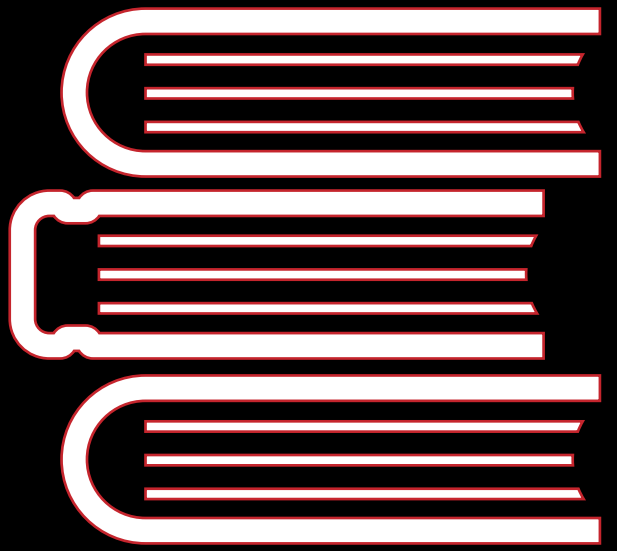
Mobile Security Expert @ Thales DIS

Penetration Tester

Information Security Researcher

Agenda

- Intro
- Motivation
- AppAttest Overview
- Pros & Cons
- Conclusion



Topic Coverage

Examples of Client-Side Protections

- Anti-Tampering
- Obfuscation
- SSL Pinning
- Root / JailBreak detection
- ETC

Application Tampering Definition

Tampering - A process of changing a mobile app or its environment to affect its behavior

https://mobile-security.gitbook.io/mobile-security-testing-guide/general-mobile-app-testing-guide/0x04c-tampering-and-reverse-engineering

Anti-Tampering - Runtime detection of the presence of an implant or binary modification

Popular Anti-Tampering Techniques

- Pre-computed Hash Verification
- Signing Certificate Verification
- Resource Integrity Check

Popular Anti-Tampering Techniques

- Pre-computed Hash Verification
- Signing Certificate Verification
- Resource Integrity Check

Problem => These checks are done on the client-side and can be disabled by the same method against which they were created.

Possible solution?

What if we can verify the pre-computed signature/hash on our backend-server?

Meet the AppAttest API... finally

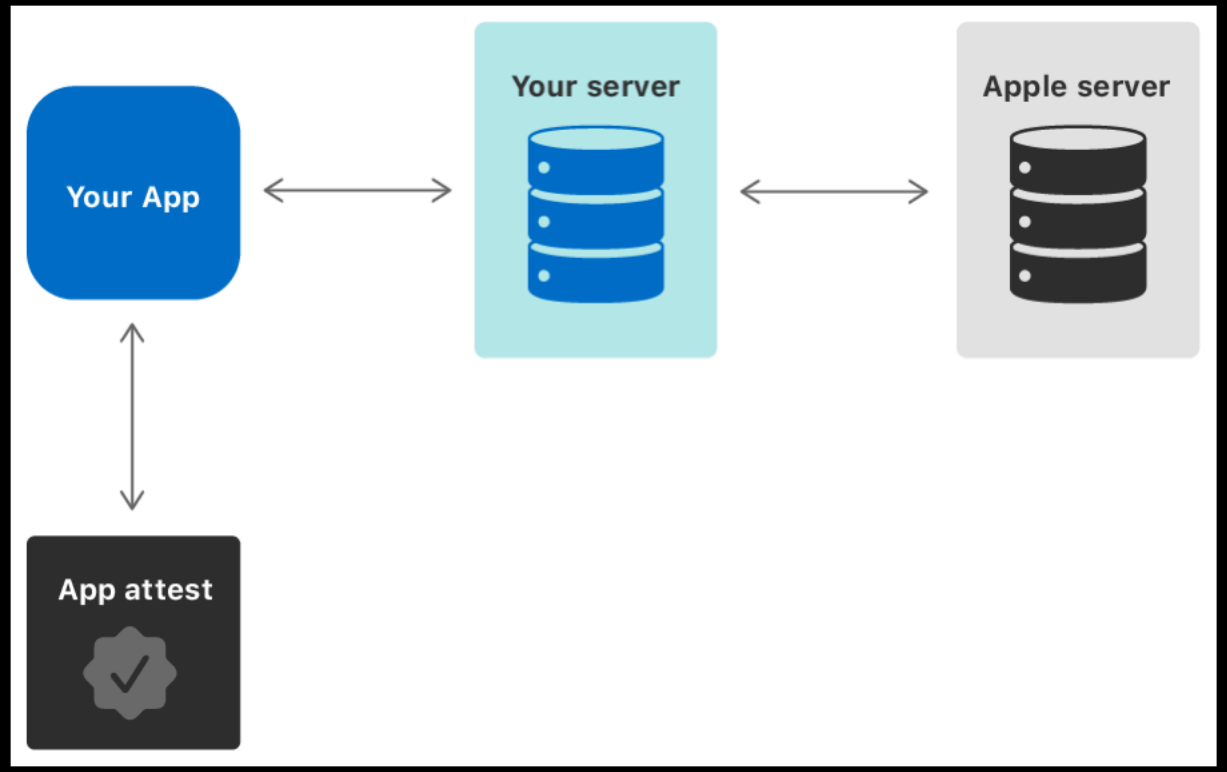
AppAttest API - Definition

Verify your app's integrity with the new App Attest API

August 3, 2020

Part of the DeviceCheck services, the new App Attest API helps protect against security threats to your apps on iOS 14 or later, reducing fraudulent use of your services. With App Attest, you can generate a special cryptographic key on a device and use it to validate the integrity of your app before your server provides access to sensitive data. - Apple

AppAttest - Definition



How it works

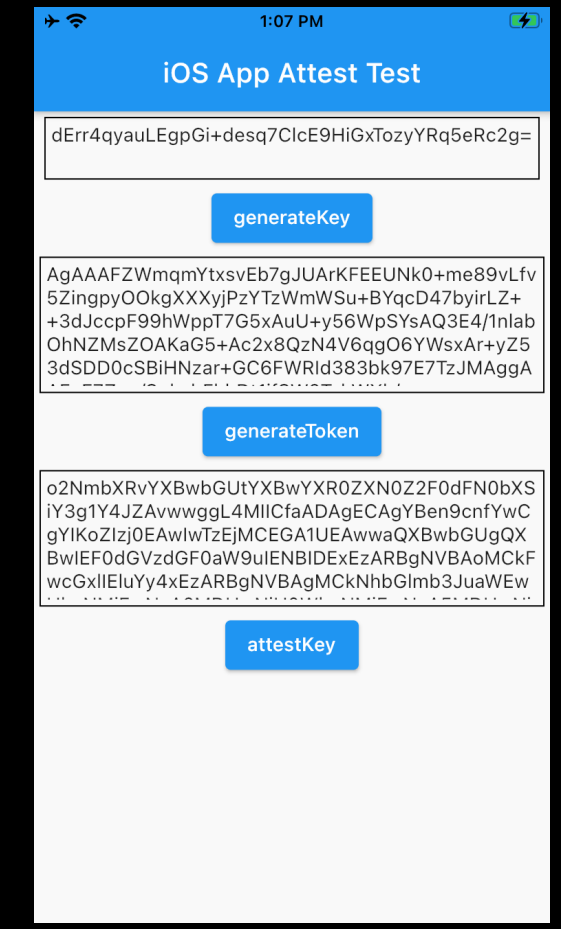
Apart from marketing bs

Sample Demo App

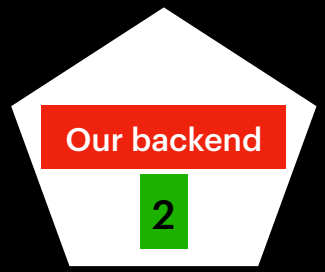
Written in Dart/Flutter

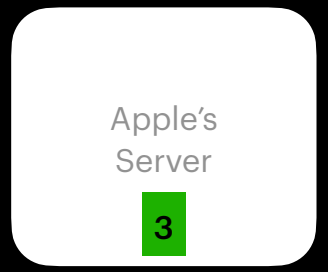
Tested on iPhone 8, iOS 14.0.1

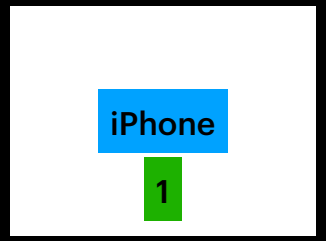
Back-end parts are hard-coded on the client side



3 Parties Involved







Step 1

Step 1

iPhone

Apple's Server

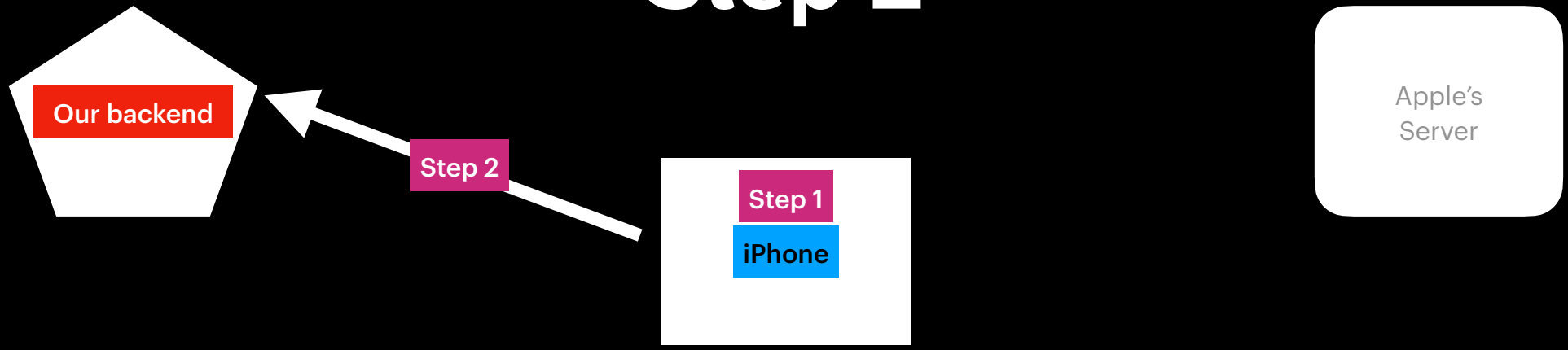
Remark from Apple: Must be done using un-compromised, trusted iOS Device on our side

1) Key generation (keyGenerate function). Happens on the iPhone using SecureEnclave

Reference

Example of generated key:
Result:
LuP7C3XHvXiqe5iVnRDENwSlSKlevcnu6FznqrOM5gw=

Step 2



Action	
1)	Key generation. Happens on the iPhone using SecureEnclave
2)	Request challenge generation on our backend

Reference

Example of challenge from our backend:
Result:
olcNbaDfflgnXbpd80scSh3WYDOwaEn2iNIFUtIU_Ex

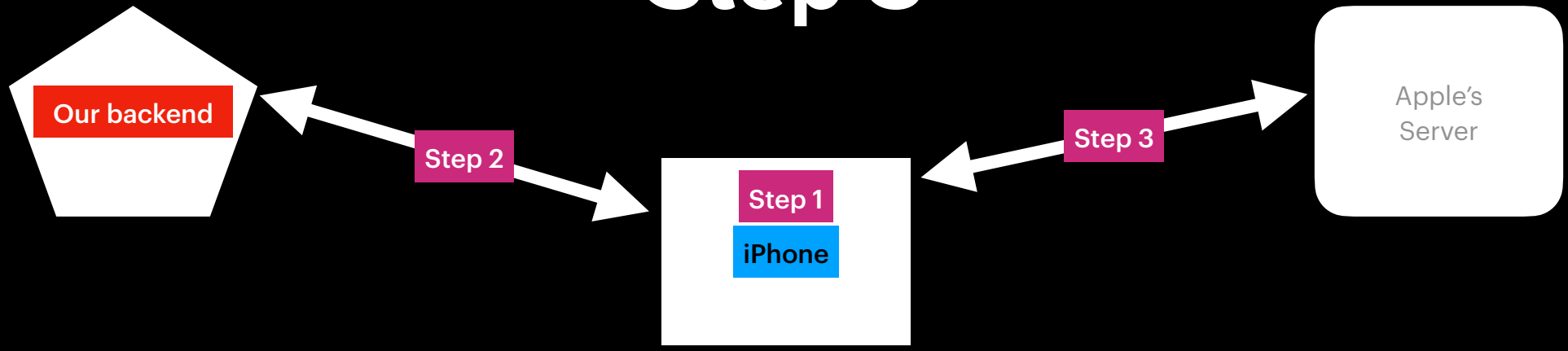
AttestKey function

```
private func handleAppAttestServiceAttestKey(_ arg: [String : Any?], result: @escaping FlutterResult) {
  guard #available(iOS 14.0, *) else {
    result(FlutterError(code: "unavailable", message: "Only available in iOS 14.0 or newer.", details: nil))
    return
  }
  let keyId = arg["keyId"] as! String
  let clientDataHash = (arg["clientDataHash"] as! FlutterStandardTypedData).data
  DCAppAttestService.shared.attestKey(keyId, clientDataHash: clientDataHash) { object, error in
    if let error = error {
      result(getFlutterError(error))
      return
    }
    result(object)
  }
}
```

Key generated at Step 1

Anti-Replay Hash from our Backend (Step 2)

Step 3



Action	
1)	Key generation
2)	Request challenge generation on our backend
3)	Attest the generated key by utilising AttestKey call on iPhone. AttestationObject returned from Apple server.

Reference

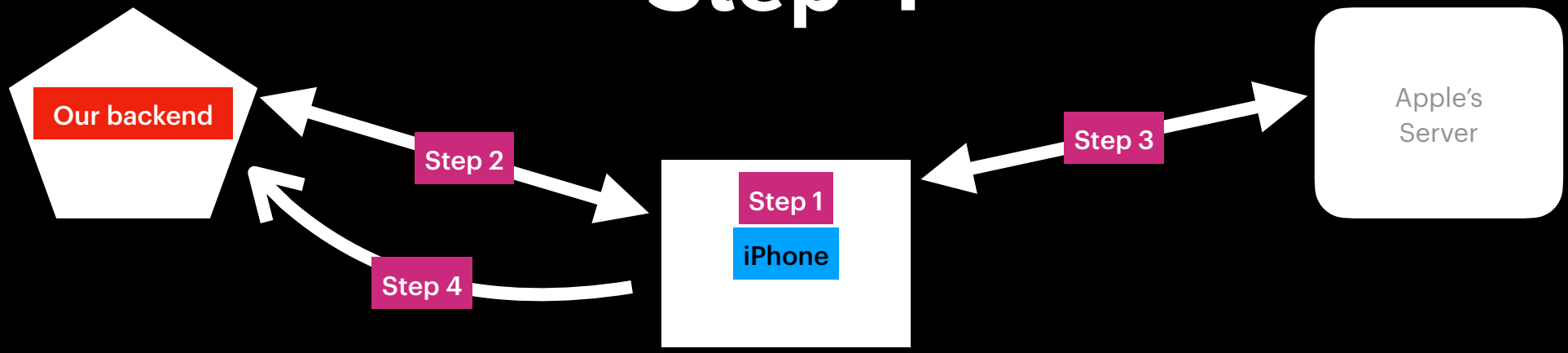
Example of AttestationObject generated by AttestKey call (quite big and won't fit here):
Result:
5ZDswwCgYIKoZIzj0EAwIwTzEjMCEGA1UEAwwaQXBwbGUgQXBwIEF0dGVzdGF0aW9uIENBIDExEzARBgNVBAoMCkFwcGxlIEluYy4xEzARBgNVBAgMCkNhb...xX

AttestationObject authenticator data

- `RP ID` (32 bytes) — A hash of your app's App ID, which is the concatenation of your 10-digit team identifier, a period, and your app's `CFBundleIdentifier` value. An attestation that an App Clip generates uses the full app's identifier, not the App Clip's identifier. For information about the difference between the two, see Creating an App Clip with Xcode.
- `counter` (4 bytes) — A value that reports the number of times your app has used the attested key to sign an assertion.
- `aaguid` (16 bytes) — An App Attest–specific constant that indicates whether the attested key belongs to the development or production environment. Apps generate keys using the former during development, and the latter after distribution, as `App Attest Environment` describes.
- `credentialId` (32 bytes) — A hash of the public key part of the attested cryptographic key pair.

CFBundleIdentifier - A *bundle ID* uniquely identifies a single app throughout the system.

Step 4

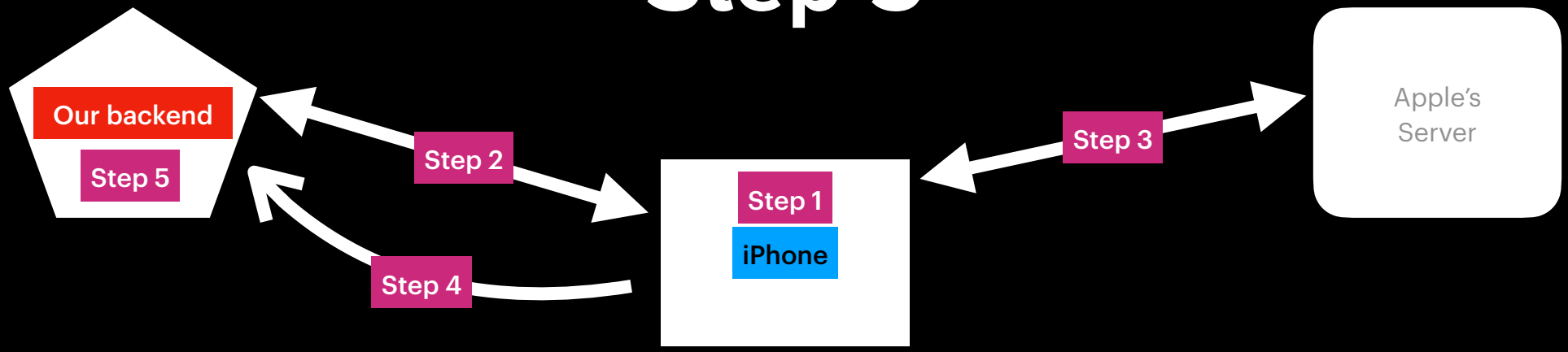


Action	
1)	Key generation
2)	Request challenge generation on our backend
3)	Attest the generated key by utilising AttestKey call on iPhone. AttestationObject returns from Apple server.
4)	Send values generated during all 3 steps back to your backend server

Reference

Example of payload to send to the backend:
"AttestationObject":"5ZDswwCgYIKoZIzj0EAwIwTzEjMCEGA1...."
"KeyID":"LuP7C3XHvXiqe5iVnRDENwSlSKlevcnu6FznqrOM5gw="
"Challenge":"olcNbaDfflgnXbpd80scSh3WYDOwaEn2iNIFUtIU_Ex"

Step 5



Action	
1)	Key generation
2)	Request challenge generation on our backend
3)	Attest the generated key by utilising AttestKey call on iPhone. AttestationObject returns from Apple server.
4)	Send values generated during all 3 steps back to your backend server
5)	Validate AttestationObject on your backend server

Reference

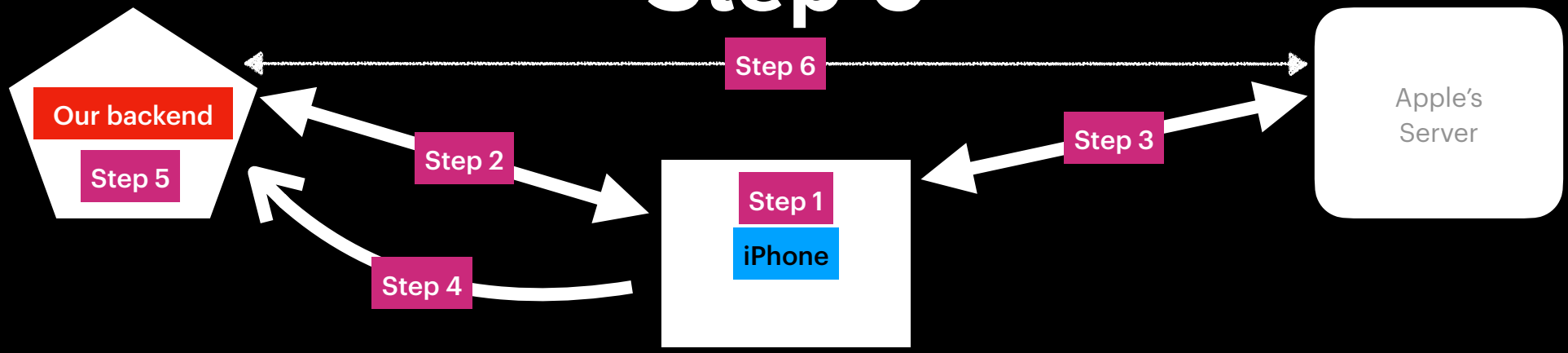
Example of payload to send to the backend:
"AttestationObject":"5ZDswwCgYIKoZIzj0EAwIwTzEjMCEGA1...."
"KeyID":"LuP7C3XHvXiqe5iVnRDENwSlSKlevcnu6FznqrOM5gw="
"Challenge":"olcNbaDfflgnXbpd80scSh3WYDOwaEn2iNIFUtIU_Ex"

AttestationObject validation

1. Verify that the `x5c` array contains the intermediate and leaf certificates for App Attest, starting from the credential certificate in the first data buffer in the array (`credcert`). Verify the validity of the certificates using Apple's App Attest root certificate.
2. Create `clientDataHash` as the SHA256 hash of the one-time challenge your server sends to your app before performing the attestation, and append that hash to the end of the authenticator data (`authData` from the decoded object).
3. Generate a new SHA256 hash of the composite item to create `nonce`.
4. Obtain the value of the `credCert` extension with OID `1.2.840.113635.100.8.2`, which is a DER-encoded ASN.1 sequence. Decode the sequence and extract the single octet string that it contains. Verify that the string equals `nonce`.
5. Create the SHA256 hash of the public key in `credCert`, and verify that it matches the key identifier from your app.
6. Compute the SHA256 hash of your app's App ID, and verify that it's the same as the authenticator data's `RP ID` hash.
7. Verify that the authenticator data's `counter` field equals `0`.
8. Verify that the authenticator data's `aaguid` field is either `appattestdevelop` if operating in the development environment, or `appattest` followed by seven `0x00` bytes if operating in the production environment.
9. Verify that the authenticator data's `credentialId` field is the same as the key identifier.

After successfully completing these steps, you can trust the attestation object.

Step 6



Action
1) Key generation
2) Request challenge generation on our backend
3) Attest the generated key by utilising AttestKey call on iPhone.
4) Send values generated during all 3 steps back to your backend server
5) Validate AttestationObject on your backend server
6) Use the receipt that was extracted from AttestationObject during Step 5. Send the receipt to Apple Server to get the metric

Reference

Example of request to the Apple's endpoint:
curl -i --verbose -H "Authorization: <JWT>" -X POST --data-binary "Receipt_Base64" https://data-development.appattest.apple.com/v1/attestationData

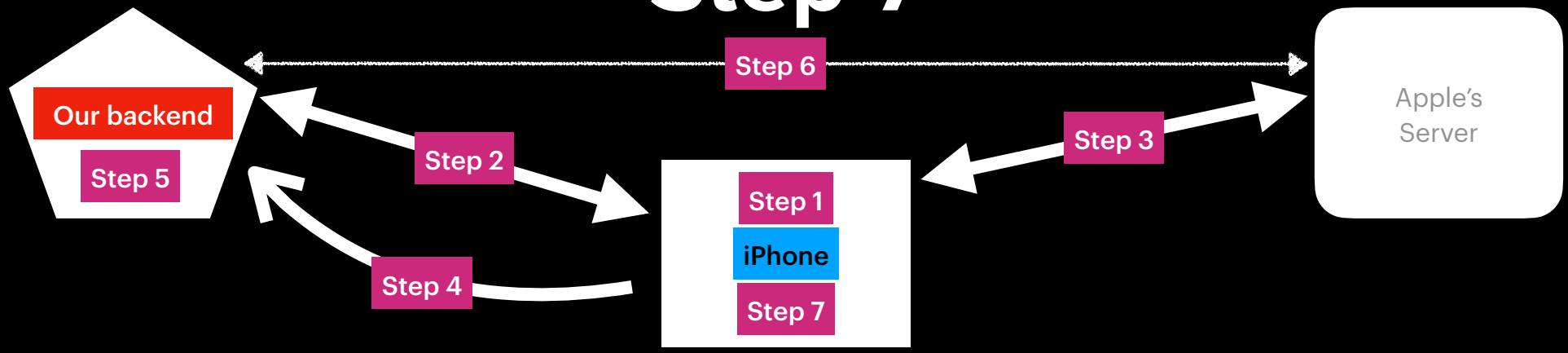
Server-to-Server interaction

Interpret the Metric

Field 6 of the receipt contains either the string `ATTEST` for the receipt that comes with an attestation object, or the string `RECEIPT` for receipts that you request using your server. Only the latter provide the risk metric in field 17. The receipt represents the metric as a string that indicates the number of attested keys associated with a given device over the lifetime of the device. Look for this value to be a low number.

Note that the metric can grow if a user reinstalls your app, restores from a backup, or transfers a device to another user. For privacy reasons, App Attest keys stored on device don't survive these events, forcing your app to generate a new key on the same device. This growth should be modest, but you'll have to tune your risk assessment logic based on the typical numbers that you see over time. You can help to keep the number small by only generating new keys when absolutely necessary.

Step 7

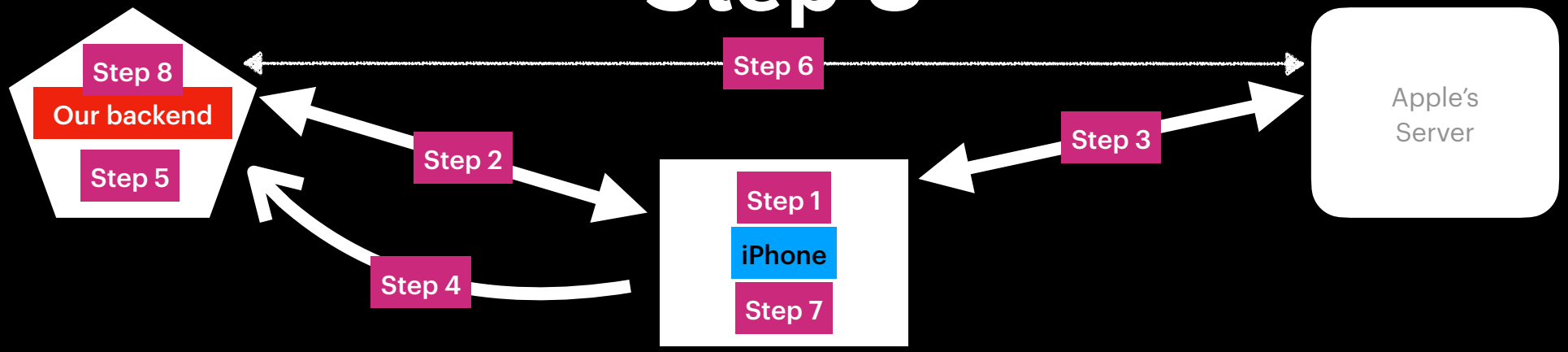


Action	
1)	Key generation
2)	Request challenge generation on our backend
3)	Attest the generated key by utilising AttestKey call on iPhone
4)	Send values generated during all 3 steps back to your backend
5)	Validate AttestationObject on your backend server
6)	Use the receipt that was extracted from AttestationObject during Step 5
7)	Now you can use generateAssertion method to sign requests to your backend with attested key

Reference

Example of an assertionObject:
CBOR Object

Step 8



Action	
1)	Key generation
2)	Request challenge generation on our backend
3)	Attest the generated key by utilising AttestKey call on iPhone
4)	Send values generated during all 3 steps back to your backend
5)	Validate AttestationObject on your backend server
6)	Use the receipt that was extracted from AttestationObject during Step 5
7)	Now you can use generateAssertion method to sign requests to your backend with attested key
8)	Verify the assertion object

Reference

Check 'Verify the Assertion' part:
https://developer.apple.com/documentation/devicecheck/validating_apps_that_connect_to_your_server

Assertion Object Validation

To verify the assertion, use the decoded assertion, the client data, and the previously stored public key, and follow these steps:

1. Compute `clientDataHash` as the SHA256 hash of `clientData`.
2. Concatenate `authenticatorData` and `clientDataHash`, and apply a SHA256 hash over the result to form `nonce`.
3. Use the public key that you store from the attestation object to verify that the assertion's `signature` is valid for `nonce`.
4. Compute the SHA256 hash of the client's App ID, and verify that it matches the `RP ID` in the authenticator data.
5. Verify that the authenticator data's `counter` value is greater than the value from the previous assertion, or greater than `0` on the first assertion.
6. Verify that the embedded challenge in the client data matches the earlier challenge to the client.

When the assertion meets all of these conditions, you can trust it. Store `counter` to use in step 5 when verifying the next assertion.

What's next?

At this point AppAttest API is correctly implemented and works fine. Are we finally protected from hackers, crackers, modders and other guys who want to mess with our App?

Some Fun From Apple

Same article

You can't rely on your app's logic to perform security checks on itself because a compromised app can falsify the results. Instead, you use the `shared` instance of the `DCAppAttest`

Before using a key pair, ask Apple to attest to its origin on Apple hardware running an uncompromised version of your app. Because you can't trust your app's logic to verify the attestation result, you send the result to your server. To reduce the risk of replay attacks during

How can I say that app is uncompromised if you're saying that I can't rely on my app's logic?

Not all devices can use the App Attest service, so it's important to have your app run a compatibility check before accessing the service. If the user's app doesn't pass the compatibility check, gracefully bypass the service. You check for availability by reading the `isSupported` property.

Meh...

Some Fun From Apple

Your app uses the App Attest service to assert its authenticity. A compromised version of your app running on a genuine, unmodified Apple device can't create valid assertions. However, an attacker that modifies the device's operating system might bypass restrictions. Although modifying the operating system is difficult and an unlikely source of widespread fraud, you might need to guard against an attack that uses a single compromised device to serve assertions to many subscribers.

What?

Alright. Let's assume Checkra1n or other modern JailBreaks for iOS 14 doesn't exist

While it isn't possible to detect fraudulent activity with absolute certainty, App Attest does provide a metric to assess its likelihood. Specifically, you can get an approximate count of unique attestations for your app on a particular device. A count that's higher than expected might be an indication of a compromised device that's serving multiple compromised instances of your app. You can use this information to assess your risk.

Meh #2

Some Fun From Apple

And last one from Apple developer's forum. Question:

What stops a compromised/fake app instance to pretend to run on 'not supported' device, report that to the app server and that way circumvent App Attest completely?

Is there any way for the app server to verify this 'not supported' claim received from the app instance?

Security DeviceCheck

Brilliant answer - please go figure it out on your own... I wish I saw this answer before starting this research

Accepted Answer

As you note, a compromised app may remove the call to the App Attest service, preventing the service from being used.

The absence of the attestation when your service expects it may be used by the service as a risk signal.

The App Attest framework is supported on iOS/iPad OS 14 and later for devices that have a SEP. As adoption of iOS 14 increases the absence of the attestation will provide an increasingly strong risk signal.

It is also critical to follow the full verification procedure on your service to ensure any attestation received has not been manipulated.

Posted 1 year ago by Frameworks Engineer

Bypass-related Scenarios

AppAttest can't detect if Device is

- JailBroken

AppAttest can't detect if App is

- Already Tampered prior installation

AppAttest can't detect if App is

- Modified in runtime (hooking, swizzling)

Bypass-related Scenarios. Case 1

Drop outgoing http connection to Apple's server

```
The client failed to negotiate an SSL connection to gateway.icloud.com:443: Remote host closed connection during handshake
[7]  The client failed to negotiate an SSL connection to register-development.appattest.apple.com:443: Remote host closed connection during handshake
```

Done either by hooking or MITM proxy

If the method, which accesses a remote Apple server, returns the `serverUnavailable` error, try attestation again later with the same key. For any other error, discard the key identifier and

Possible because of incorrect handling of serverUnavailable error

https://developer.apple.com/documentation/devicecheck/dcerror/3585178-serverunavailable

Bypass-related Scenarios. Case 2

Return that device is not supported or current iOS version is <14

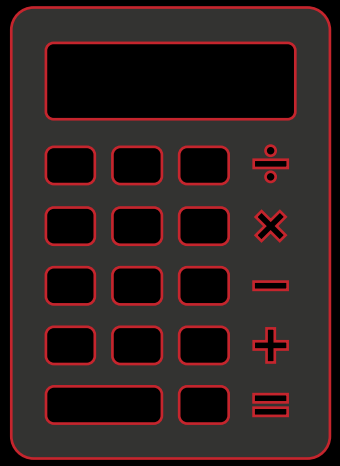
Interesting fact:

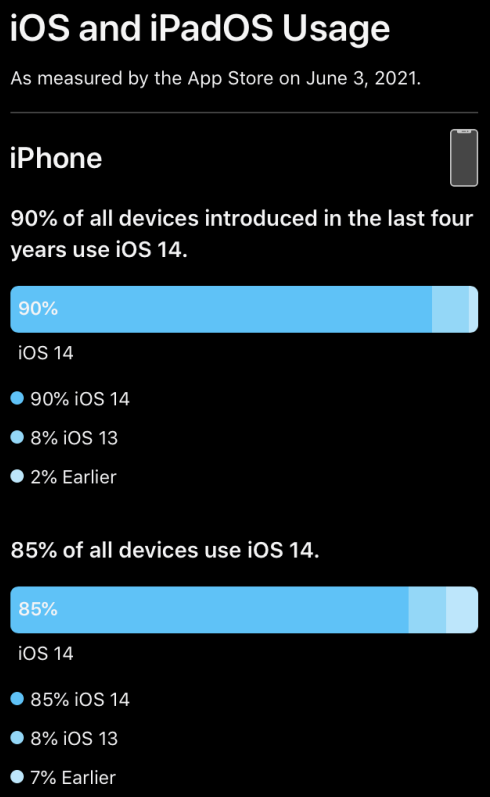
All supported iOS devices are always return true on isSupported call, so this check might be not implemented if the Application is released for mobile-only systems

Hook Platform API to return version less than iOS 14

```
[VERBOSE-2:ui_dart_state.cc(199)] Unhandled Exception: PlatformException(unavailable, Only available in iOS 14.0 or newer., null, null)
#0      StandardMethodCodec.decodeEnvelope (package:flutter/src/services/message_codecs.dart:597:7)
#1      MethodChannel._invokeMethod (package:flutter/src/services/platform_channel.dart:158:18)
<asynchronous suspension>
#2      _applicationState._generateToken (package:device_check_example/main.dart:36:11)
<asynchronous suspension>
[VERBOSE-2:ui_dart_state.cc(199)] Unhandled Exception: PlatformException(unavailable, Only available in iOS 14.0 or newer., null, null)
#0      StandardMethodCodec.decodeEnvelope (package:flutter/src/services/message_codecs.dart:597:7)
#1      MethodChannel._invokeMethod (package:flutter/src/services/platform_channel.dart:158:18)
<asynchronous suspension>
#2      _applicationState._generateKey (package:device_check_example/main.dart:50:11)
<asynchronous suspension>
```

Bypass-related Scenarios. Case 2





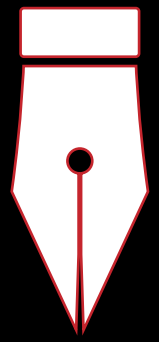
10% of all iOS and iPadOS devices is

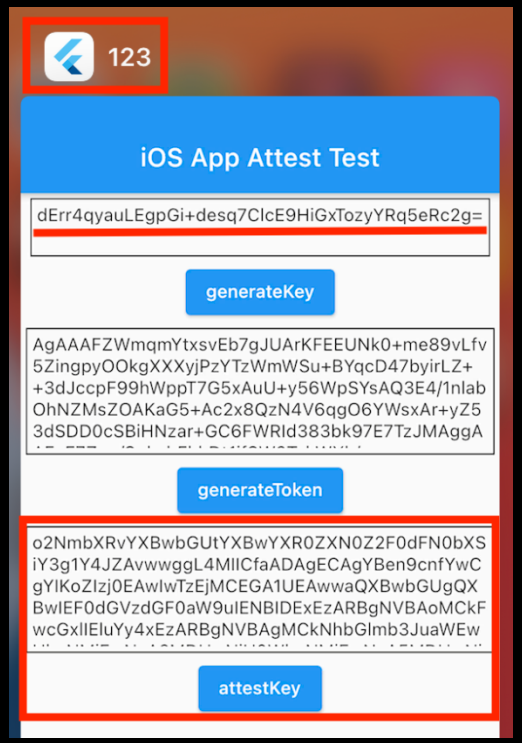
~100.000.000 unsupported devices

This version-related bypass will be most efficient for some time untill iOS <14 is EOL

Bypass-related Scenarios. Case 3

Abuse incorrect parsing of AttestationObject on back-end





Nothing stops you from patchnig and re-signing the Target App

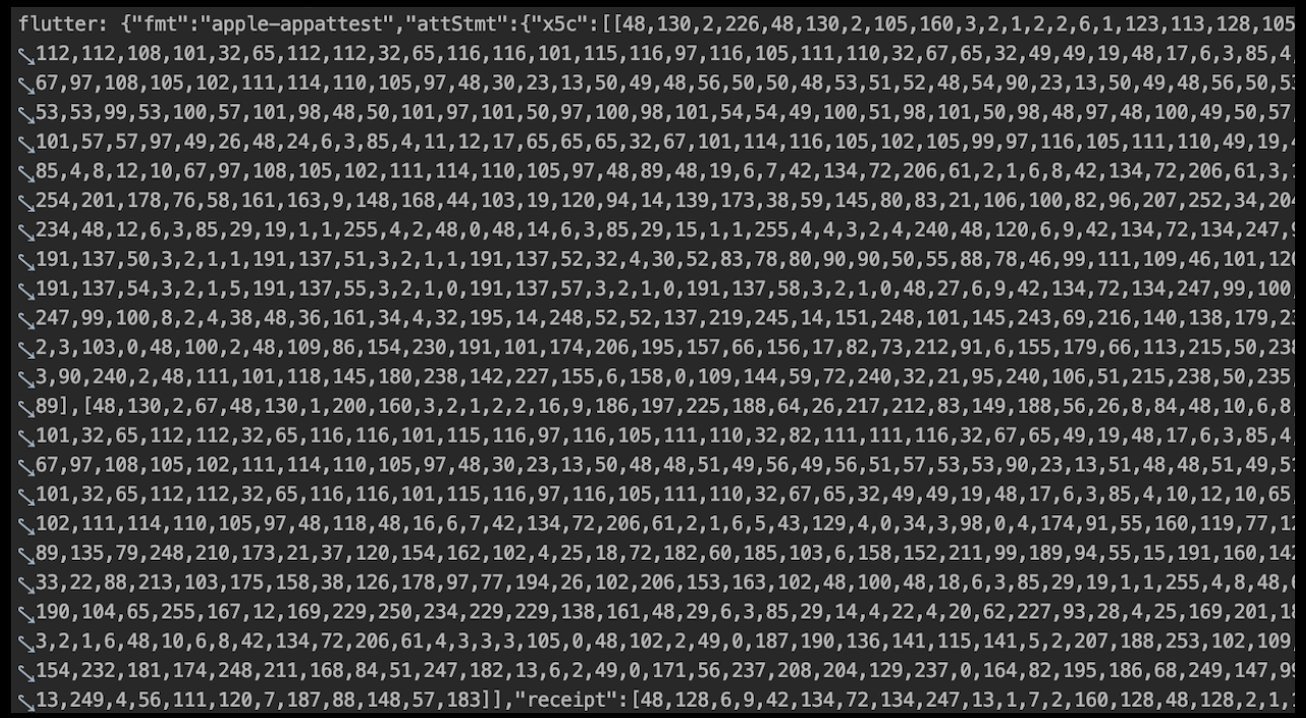
Apple doesn't check the bundle identifier on their side nor validates the sandbox state (JailBroken or not)

AttestationObject will be generated anyway

Bypass-related Scenarios. Case 3

Abuse incorrect parsing of AttestationObject on back-end

AttestationObject consist of multiple byte arrays and different fields



Must be properly parsed and validated following Apple's guidelines

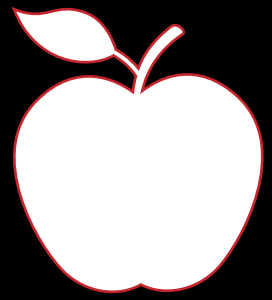
Worth it to implement or not?

PROS

CONS

Pros

1) Looks promising against replay attacks
2) Certain companies can benefit from implementing Fraud Metric analysis, especially if Apple expand the list of metric data
3) Could increase anti-tampering protection as additional layer of security by leveraging bypass complexity for already implemented RASP checks
4) Might become industry standard once supported by 100% of iOS devices



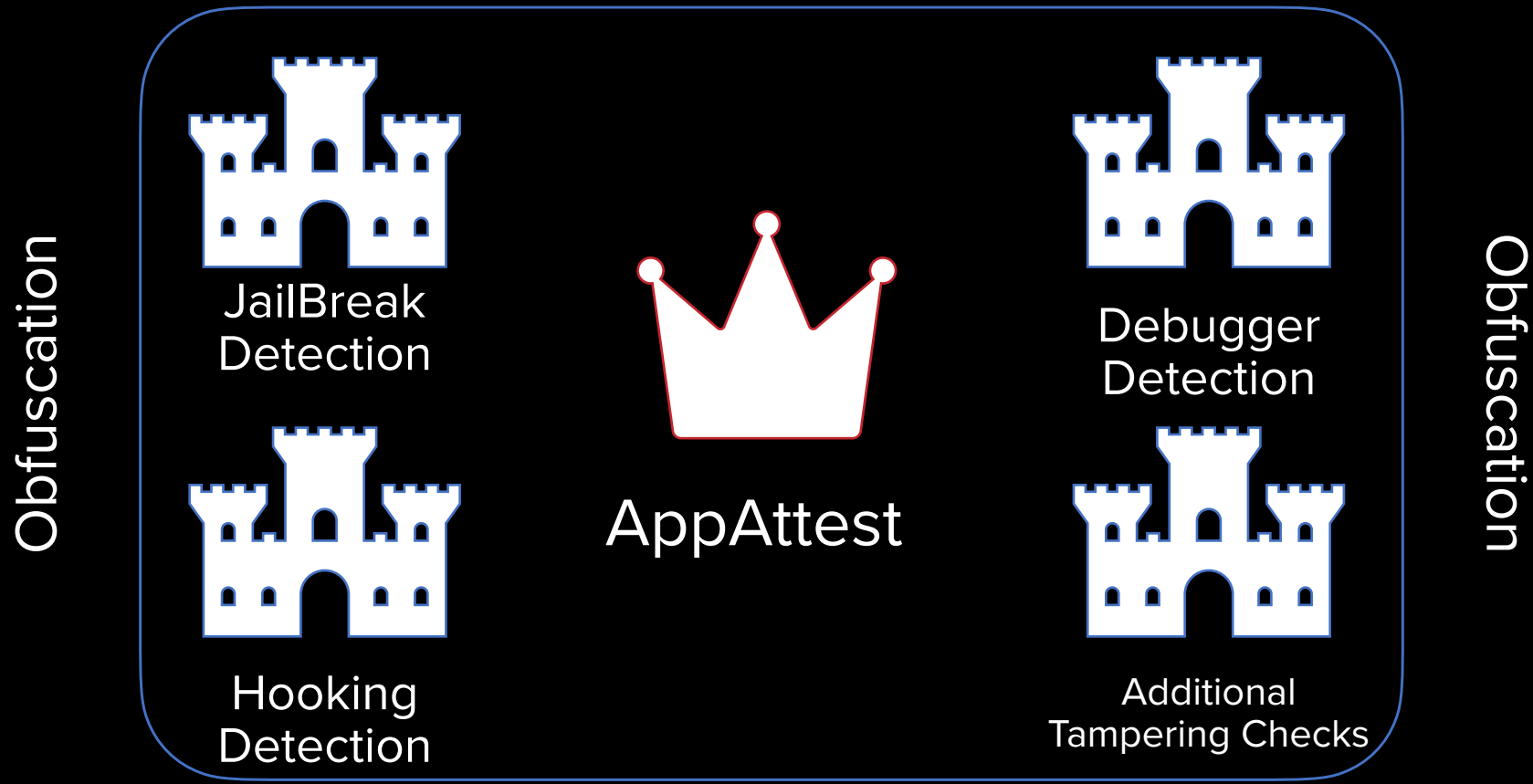
Cons

1) Not possible to validate if Application is already running on compromised device
2) If device has been JailBroken after the key attestation, all new assertions are tampered - not possible to detect it even using Apple’s fraud metric
3) Useless against application's behaviour modification with Runtime Instrumentation Frameworks/Debugger
4) Multiple design issues - success of the implementation strongly depends on integrator’s implementation
5) App extensions doesn’t support App Attest
6) Dependency on 3rd party - Apple's back-end server
7) Number of unsupported devices is still big

Worth it to implement or not?

Its necessary for the project team to discuss all potential risks and identify the impact and threat model.

How to do it well?



Conclusion

AppAttest

- Backup solution, yet not ready to replace traditional Anti-Tampering mechanisms
- Might evolve in the future
- Bypass complexity is relatively easy due to multiple logical issues in its implementation
- Recommended for integration only in experimental mode as additional source of knowledge

References

- Sample Project used in this talk, written in Dart/Flutter: https://github.com/HD421/iOS-AppAttest-Playground
- Example of server-side implementation, written in Kotlin: https://github.com/veehaitch/devicecheck-appattest
- Comprehensive tampering description and techniques review: https://mobile-security.gitbook.io/mobile-security-testing-guide/general-mobile-app-testing-guide/0x04c-tampering-and-reverse-engineering
- AppAttest Service documentation: https://developer.apple.com/documentation/devicecheck/dcappattestservice
- Framework related articles worth to read:
  - https://developer.apple.com/documentation/devicecheck/establishing_your_app_s_integrity
  - https://developer.apple.com/documentation/devicecheck/validating_apps_that_connect_to_your_server
  - https://developer.apple.com/documentation/devicecheck/assessing_fraud_risk

HITB SECCONF SIN-2021

Thank You for Joining Us

Join our Discord channel to discuss more or ask questions

https://discord.gg/dXE8ZMvU9J