Ensuring the Authenticity and Fidelity of Camera data

Date	2021–03–03
Author(s)	huazhe.thz@antgroup.com, jianxu.zjx@antgroup.com, jingwei.yang@antgroup.com , jushi.zf@antgroup.com
Status	EXTERNALLY PUBLISHED

蚂蚁集团
ANT GROUP

Introduction

Mobile devices, which users habitually carry along, have become the main data gateway for the majority of the online services. Cameras, microphones, accelerometers, and more recently, fingerprint scanners, are already commonplace on these devices. At the same time, modern identification techniques ask users to scan their ID documentation or do face verification authentication in order to be verified by an online service. Such techniques request from users to capture a photo through the camera of their mobile device. Data authenticity and fidelity is crucial for user identification techniques. Therefore, it is vital to address feasible attacks.
Here we will concentrate on one such attack — the injection attack. Such attacks are easy to perform, devastatingly successful, and represent a significant barrier to the widespread deployment of authentication systems that use consumer smart devices in web scenario. Instead of attacking the system in front of the capture device (e.g., the webcam on a smart device), the injection attack operates by copying a digital representation of a real biometric signal, and then injecting it into the system at some later date and/or different location. Since the copied signal is bit-for-bit an exact replica of a legitimate signal, it can completely bypass the security checks, such as liveness and anti-spoofing.

Threat Model

Injection attacks succeed if the system does not protect the digital information captured by the camera sensor. That protection must ensure obtaining the digital signal live from the actual device, and providing that signal to the decision process intact. Compromise initially may involve copying the data. During the attack, the attacker fraudulently replaces the data with false data. The attacker obtains this false data by either carefully constructing it or using previously captured data (perhaps from a different device). Even if the attack does not target the original (or derived) signal, the extracted feature template may also be subject to injection attacks. To use signals captured by these uncontrolled smart devices successfully in an authentication system, the system must validate the source of that signal and preserve the integrity of any data.

In the scenario of using a webcam on a smart device, the webcam does not currently possess any intelligent capabilities to defeat injection attacks. The communication path from the camera to the remote server is generally not protected, allowing the original stream to be intercepted or modified prior to any additional security (such as authentication meta-data) being added.

To address this vulnerability, what is required is a data integrity and timeliness test. The digital data must be tested to confirm that the capture occurred at the time the authentication system expected it to be (i.e., now), that it came from the expected device, and that it has not been tampered with.

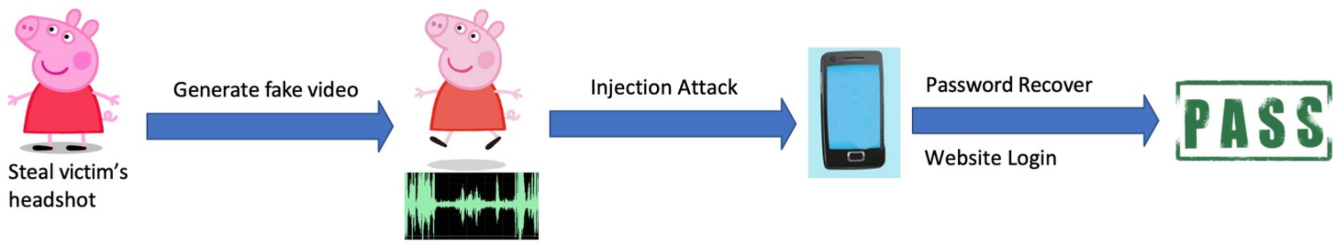

Figure 1. video injection to bypass liveness detection

Software-level Video Injection Example

- Add pre and post middleware hooks to webrtc methods to form a complete video injection attack device
- Using the above device, we can disguise the HDMI output of a PC as a video stream captured by native camera

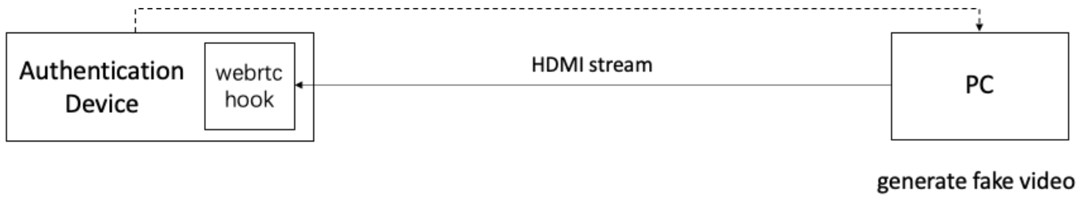

Figure 2. Software-level Video Injection Workflow

Hardware-level Video Injection Example

- Using a chip to make a hardware module that can converts HDMI stream to MIPI CSI stream
- Connecting the above module to an Android development board to form a complete video injection attack device
- Using the above device, we can disguise the HDMI output of a PC as a video stream captured by native camera

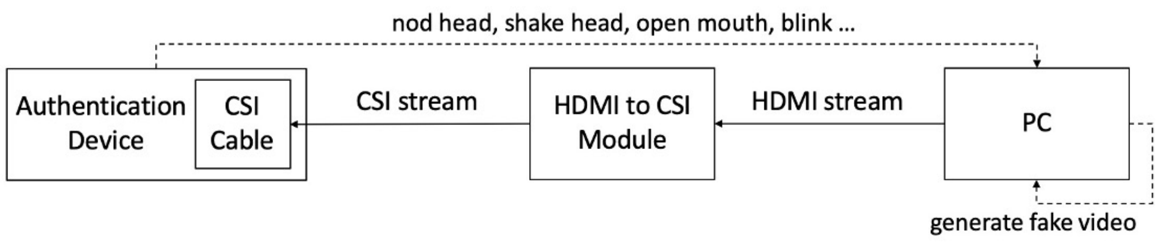

Figure 3. Hardware-level Video Injection Workflow

Our Proposal

One of the most important goal is the ability of the remote service to verify the authenticity of the received data. In our proposal, remote image attestation utilizes a modified Webauthn protocol to facilitate a public-key signatures between the client and the server, means that you have the private key and you share a public key with the verifier of the signature. When a photo has been captured, it computes and signs the hash of the photo with the private key. When the user submits the photo, the server computes the hash of the photo with the same hash function as the client side and using the public key for the corresponding user verifies that the signed hash matches the hash of the received photo. If this is successful, then the uploaded photo has not been tampered with, otherwise it is potentially dangerous to trust the client with camera data.

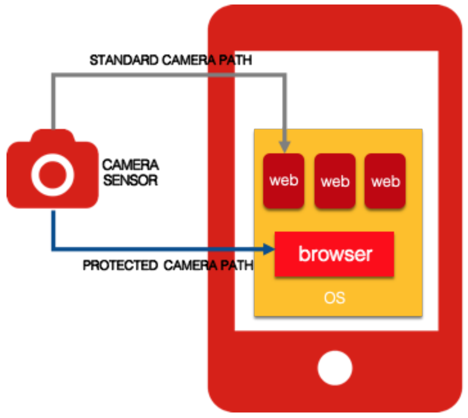

Figure 4. Photo authenticity and fidelity protection model

Using the Remote Image Attestation flow, a browser can attest to a remote entity that it is trusted, and establish an authenticated communication channel with that entity. As part of attestation, the browser proves the following:

- Its website identity
- That uploaded data has not been tampered with
- That it is running on a genuine platform with Image Attest Service enabled
- That it is running at the latest security level, also referred to as the Trusted Computing Base (TCB) level

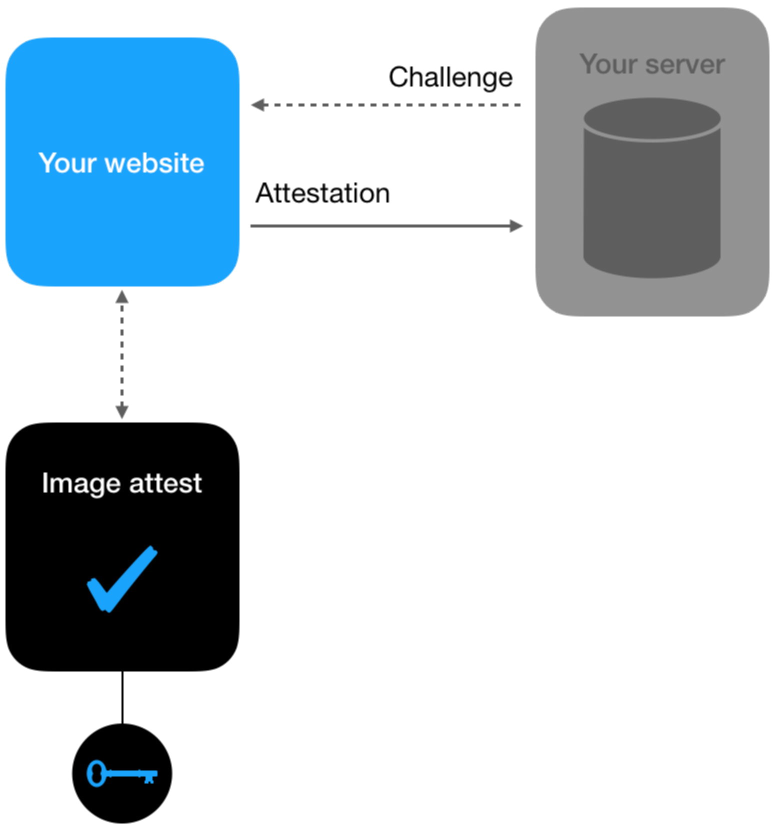

Figure 5. Remote Photo Attestation at a glance

This proposal defines the procedure enabling the creation and use of attested, scoped, public key-based webcam photo by web applications, for the purpose of ensuring the authenticity and fidelity of captured Photos. A public key credential is created and stored by a local image attest service at the behest of a Relying Party, subject to user consent. Subsequently, the public key credential can only be accessed by origins belonging to that Relying Party. This scoping is enforced jointly by conforming web browser and local image attest service. Some modifications of webauthn are proposed here to make it suitable to protect against any type of injection attack. Here we propose an alternatives to webauthn or webauthn variants to attain this objective.

Here is the comparison among Credentials Management API, WebAuthentication API and **Image Attest API.**

Credentials Management API	WebAuthentication API	Image Attest API
navigator.credentials.store	navigator.credentials.create	navigator.credentials.create
navigator.credentials.get	navigator.credentials.get	navigator.credentials .generateImageAssertion

Image Attest API is actually an extension to WebAuthentication API. In the nutshell it is an interface to talk to media service. You give it a challenge. You get media assertion back.

In order to understand how remote image attestation works, it is important to understand that they sit between two components that are outside the browser:
Server – It is intended to register new credentials on a server (also referred to as a service or a relying party) and later use those same credentials on that same server to authenticate a user.
Image attest – the credentials are created and stored in a device called a image attest service, when a photo has been captured, it computes and signs the hash of the photo with the private key.

As webauthn specification, a registration process has six steps, as illustrated in Figure 6 and described further below. This is a simplification of the data required for the registration process that is only intended to provide an overview.

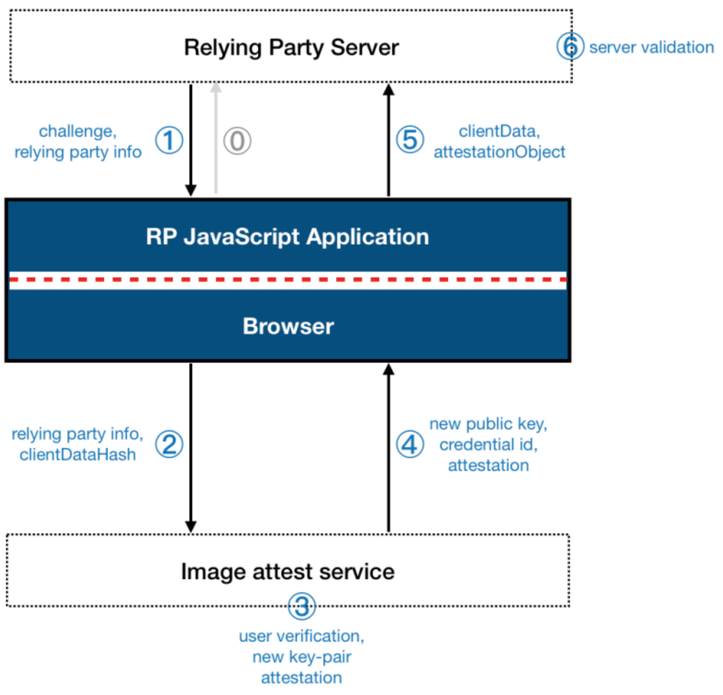
Relying Party Server
6 server validation
challenge, relying party info
1
0
5
clientData, attestationObject
RP JavaScript Application
Browser
relying party info, clientDataHash
2
4
new public key, credential id, attestation
Image attest service
3
user verification, new key-pair attestation

Figure 6. a diagram showing the sequence of actions for a Image Authenticity Attest registration and the essential data associated with each action.

The registration steps are:

0. **Application Requests Registration** – The application makes the initial registration request. The protocol and format of this request is outside of the scope.

1. **Server Sends Challenge and Relying Party Info** – The server sends a challenge and relying party information to the JavaScript program.
2. **Browser Calls on Image attest service** – Internally, the browser will validate the parameters and fill in any defaults. One of the most important parameters is the origin, which is recorded as part of the clientData so that the origin can be verified by the server later.
3. **Image attest service Creates New Key Pair and Attestation** – Before doing anything, the authenticator will typically ask for some form of user verification to prove that the user is present and consenting to the registration. After the user verification, the authenticator will create a new asymmetric key pair and safely store the private key for future reference. The public key will become part of the attestation, which the authenticator will sign over with a private key that was burned into the secure environment during its manufacturing process and that has a certificate chain that can be validated back to a root of trust.
4. **Image attest service Returns Data to Browser** – The new public key, a globally unique credential id, and other attestation data are returned to the browser where they become the attestationObject.
5. **Browser Creates Final Data, Application sends response to Server** – The PublicKeyCredential is sent back to the server using any desired formatting and protocol.
6. **Server Validates and Finalizes Registration** – Finally, the server is required to perform a series of checks to ensure that the registration was complete and not tampered with. These include:
   a. Verifying that the challenge is the same as the challenge that was sent
   b. Ensuring that the origin was the origin expected
   c. Validating that the signature over the clientDataHash and the attestation using the certificate chain for that specific secure environment

After a user has registered with webauthn, they can subsequently capture photo with the service. The authentication flow looks similar to the registration flow, and the illustration of actions in Figure 7 may be recognizable as being similar to the illustration of registration actions in Figure 6.

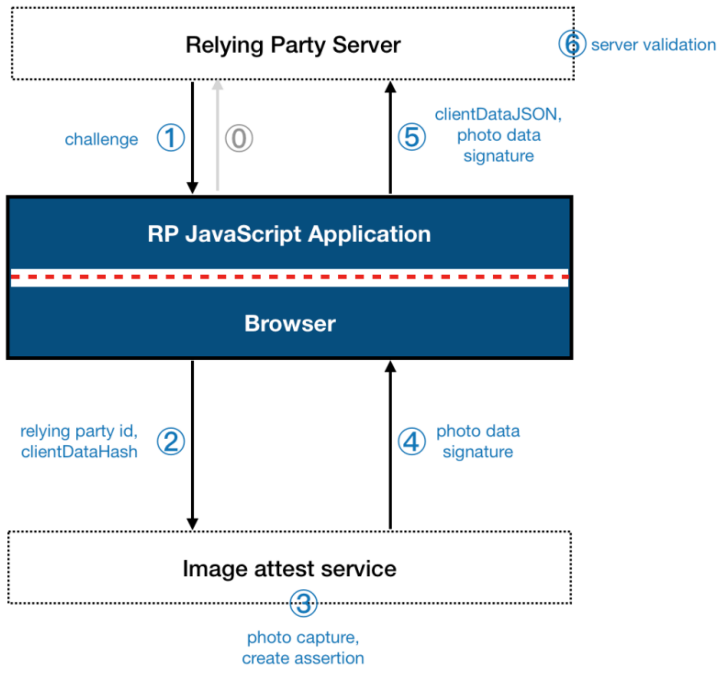
Relying Party Server
⑥ server validation
challenge ①
⓪
⑤ clientDataJSON, photo data signature
RP JavaScript Application
Browser
relying party id, clientDataHash ②
④ photo data signature
Image attest service
③
photo capture, create assertion

Figure 7. similar to Figure 6, a diagram showing the sequence of actions for Image Authenticity Attest authentication and the essential data associated with each action.

0. **Application Requests Authentication** – The application makes the initial authentication request. The protocol and format of this request is outside of the scope.
1. **Server Sends Challenge** – The server sends a challenge to the JavaScript program.
2. **Browser Calls Image attest service** – Internally, the browser will validate the parameters and fill in any defaults. One of the most important parameters is the origin, which recorded as part of the clientData so that the origin can be verified by the server later.
3. **Image attest service Creates an Assertion** – The image attest service finds a credential for this service that matches the Relying Party ID and prompts a user to capture photo. Assuming both of those steps are successful, the image attest service will create a new assertion by signing over the clientDataHash and photo data with the private key generated for this website during the registration call. The assertion signature could be embedded in the EXIF data of the photo itself.
4. **Image attest service Returns Data to Browser** – The image attest service returns the authenticatorData and assertion signature back to the browser.
5. **Browser Creates Final Data, Application sends response to Server** – It is up to the JavaScript application to transmit this data back to the server using any protocol and format of its choice.
6. **Server Validates and Finalizes Authentication** – Upon receiving the result of the authentication request, the server performs validation of the response such as:

a. Using the public key that was stored during the registration request to validate the signature by the image attest service.
b. Ensuring that the challenge that was signed by the authenticator matches the challenge that was generated by the server.
c. Checking that the Relying Party ID is the one expected for this service.

Conclusion

In the present situation, injection attacks can be extremely simple in web scenario. In contrast, the attacker must specifically customise the spoofing attack to target an individual victim. This makes injection attacks a significant risk, and a barrier to the successful deployment of enhanced authentication.

We propose camera security in web scenario is engineered to isolate the data received from the camera and securely sign it, in HW, safeguarding against untrusted applications and processes. It is designed to empower the remote service with the ability to determine about the authenticity of the received web image, helping to deliver a protected authentication method for making payments and conducting safe transactions.
Moreover, The Trusted Web Application will be able to provide authenticity to a remote service, not only for the captured photos, but also to various sensitive data captured from the sensors of a device. This can be achieved by slightly modifying our current implementation.

Appendix

How does Image Attest work?

In the existing Image Attest proposal, you adopt Image Attest to check if clients send the original data captured by the camera sensor to your server. Your website uses the `makeCredential` call to create a cryptographic key on device, and then attest to the key's validity. This produces an attestation object that your website passes to your server, along with the corresponding key identifier. Your server verifies the attestation object, and then extracts the embedded public key and other information. Later, your server uses the key to verify assertion objects with camera data sent from your website. Once verified, camera data authenticity and fidelity is maintained. Let's dive into the specifics of how this flow works.

Check for Availability

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

```
// 🆕
if(PublicKeyCredential.isImageAttestAvailable()) {
  // Perform key generation and attestation.
}
// Continue with server access.
```

You change the behavior of both your website, as shown above, and your server — which can no longer require assertions — when you find that the service isn't supported.

Provide a Challenge

Every time your website needs to communicate attestation data to your server, the website first asks the server for a unique, one-time challenge. Image Attest integrates this challenge into the objects that it provides, and that your website sends back to your server for validation. This makes it harder for an attacker to implement a replay attack.
When asked for a challenge, provide your website with a randomized data value, and remember the value for use when verifying the corresponding attestation or assertion objects sent by the client. How you use the challenge data depends on the kind of object that you need to validate.

Create a Key Pair

For each user account on every device opening your website, generate a unique, hardware-based, cryptographic key pair by calling the `navigator.credentials.create` method:

```
var challenge = <# A string from your server #>

var publicKey = {
    'challenge': challenge,

    'rp': {
        'name': 'Example Inc.'
    },

    'user': {
        'name': 'the@example.com',
        'displayName': 'Theo'
    },

    'pubKeyCredParams': [
        { 'type': 'public-key', 'alg': -7  },
        { 'type': 'public-key', 'alg': -257 }
    ]
}

navigator.credentials.create({ 'publicKey': publicKey })
    .then((newCredentialInfo) => {
        console.log('SUCCESS', newCredentialInfo)
    })
    .catch((error) => {
        console.log('FAIL', error)
    })
```

On success, the method's completion handler returns attestation response contains credential info, such as user public key, credential identifier, counter, as well as authenticator info, such as certificates, signature, aaguid and other information. The device automatically stores the associated private key in the Secure Enclave or TEE, from where the Image Attest service can use it to create signatures, but from where no process can ever directly read or modified it, ensuring its security.

Verify the Attestation

The Image Attest service creates an attestation object composed of authenticator data and an attestation statement, as specified by the Web Authentication specification. The following authenticator fields are of particular interest for Image Attest:

- `RP ID` (32 bytes) — A hash of your website's eTLD.
- `counter` (4 bytes) — The number of times your website used the attested key to sign an assertion.
- `aaguid` (16 bytes) — An Image Attest—specific constant that indicates whether the attested key belongs to the different browser environment. Different browser may have different Image Attest root certificate.
- `credentialId` (32 bytes) — A hash of the public key part of the cryptographic key pair being attested.

The attestation statement uses a custom Image attestation statement format with the following syntax:

```
$$attStmtType //= (
                      fmt: "image-attest",
                      attStmt: StmtFormat
                  )

StmtFormat =      {
                      x5c: [ credCert: bytes, * (caCert: bytes) ],
                  }
```

To verify and decode the attestation object, first check that it has the Concise Binary Object Representation (CBOR) data format with the expected syntax. The decoded object should look like this:

```
{
  fmt: 'image-attest',
  attStmt: {
    x5c: [
      <Buffer 22 33 42 de ... >,
      <Buffer 22 33 52 aa ... >
    ],
  },
  authData: <Buffer 42 c3 1e 23 ... >
}
```

Use the decoded object, along with the key identifier sent by your website, to perform the following steps:

1. Verify that the `x5c` array contains the intermediate and leaf certificates for Image Attest, starting from the credential certificate stored in the first data buffer in the array (`credcert`). Verify the validity of the certificates using browser's Image Attest root certificate depend on the authenticator data's `aaguid` field.
2. Create `clientDataHash` as the SHA256 hash of the one-time challenge sent to your website 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.xxxx.xxx.x.x`, 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 website.

6. Compute the SHA256 hash of your website's eTLD, and verify that this is the same as the upload data’s `RP ID` hash.
7. Verify that the authenticator data’s `counter` field equals `0`.
8. 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.

Store the Public Key

Store the verified public key from `credCert` on your server and associate it with the user for the given website. You’ll use this key to check assertions later. As an added protection against replay attacks, make sure that the public key isn’t already associated with another user.

Assert the Authenticity and Fidelity of Captured Photos as Needed

After successfully verifying a key’s attestation, your server can require the website to assert its authenticity for any or all future server requests. The website does this by signing the request. In the website, first obtain a unique, one-time challenge from the server. You use a challenge here, like for attestation, to avoid replay attacks. Use challenge from server and the key identifier that you generated earlier to create an assertion object by calling the `navigator.credentials.generateImageAssertion` method:

```
var publicKey = {
    challenge: <# A string from your server #>,

    allowCredentials: [
        { type: "public-key", id: credentialId }
    ]
}

// 🆕
navigator.credentials.generateImageAssertion({ 'publicKey': publicKey })
  .then((getAssertionResponse) => {
      alert('SUCCESSFULLY GOT AN ASSERTION! Open your browser console!')
      console.log('SUCCESSFULLY GOT AN ASSERTION!', getAssertionResponse)
  })
  .catch((error) => {
      alert('Open your browser console!')
```

```
        console.log('FAIL', error)
    })
```

It uses HTML5 getUserMedia to capture photos or videos from user's camera. The camera data will be passed to assertion statement as a receipt that you can use later to be displayed in your web page, rendered into a canvas, or submitted to server. On success, you can pass the completion handler's assertion object, along with the client data, to the server. If the assertion object fails verification, it's up to you to decide how to handle the request.

Verify the Assertion

After successful attestation, your server can require the associated client to accompany server requests with an assertion object. Each verified assertion reestablishes the authenticity of the client.
The client creates the assertion by packaging the request as `clientData` and asking the Image Attest service to sign the data with the attested private key. Along with the signature, Image Attest includes a receipt that you can use later to extract the camera data,and a simplified authenticator data instance in the assertion object, similar to the one in the attestation object but containing only the first few fields, including `RP ID` and `counter`.
After receiving the client data and the assertion, you'll need to verify and decode the assertion to ensure it uses the CBOR data format has has the expected contents. The decoded object should look like this:

```
{
  signature: <Buffer 22 33 42 de ... >,
  authenticatorData: <Buffer 30 23 43 1d ... >
  // 🆕
  receipt: <Buffer 30 80 06 09 ... >
}
```

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` , `receipt` and `clientDataHash` and apply a SHA256 hash over the result to form `nonce`.
3. Use the public key that you stored from the attestation object to verify that the assertion's `signature` is valid for `nonce`.
4. Compute the SHA256 hash of the website's eTLD, 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 challenge embedded in the client data matches the earlier challenge to the client.

When the assertion meets all of these conditions, you can consider the assertion verified. Store `counter` for use in step 5 of verifying the next assertion. When assertion succeeds, independently handle the receipt right away.

Limitation

- Suggested countermeasures to injection attacks are to operate in a controlled environment and challenge-response operations may limit the widespread adoption. This feature is available only in secure contexts (HTTPS), in some or all supporting browsers.
- Since proposal security depends on security of key, the security that the private key store has been more important for this purpose. Generally, it should be a hardware-based, cryptographic key.
- The data received from the camera is essentially insecure, requiring browsers to provide additional restrictions for security purposes, safeguarding against untrusted applications and processes. To mitigate hardware-layer injection, the browser should add identity authentication for native camera.

蚂蚁集团 ANT GROUP | 支付宝