Device-Centric Authentication and WebCrypto

Dirk Balfanz, Google, balfanz@google.com

A Position Paper for the W3C Workshop on Web Cryptography Next Steps

Device-Centric Authentication

We believe that the key to removing passwords as the main method of authentication on the web is to embrace *device-centric authentication*. In device-centric authentication, the user authenticates to a local device (a mobile phone, a laptop computer, *etc.*), and that device then authenticates to cloud services on behalf of the user. Under the hood, the user-to-device authentication (which can take any modality, from passwords to unlock PINs or biometrics) *unlocks* a cryptographic key on the device, which is then used in a cryptographic protocol for device-to-cloud authentication.

This approach eliminates both phishing attacks and risks associated with password re-use, because the credentials that are under user control (PINs, biometrics, *etc.*), and therefore most at risk of exposure, cannot be used to log into the user's online account. They can only be used to unlock keys on a local device. Therefore, an attacker would have to obtain both something that the user *has* (the local device with the keys on it), and something that the user *is* or *knows* (the credential that the user uses to unlock those keys) in order to compromise the user's account, making device-centric authentication an effective form of (non-phishable) two-factor authentication.

Lessons from FIDO U2F

Google has been running a trial of FIDO U2F authentication, replacing its internal two-factor authentication system with FIDO U2F-compatible USB tokens. While U2F does not fully achieve the goal of device-centric authentication (the U2F USB tokens don't have a way to independently authenticate the user and tie the use of cryptographic keys to local user authentication - and are therefore used in conjunction with regular passwords), we can still draw some valuable lessons from the deployment of this system, which we list below:

WebCrypto is Not Enough

Much of webcrypto is designed to work as if the cryptographic algorithms had been implemented purely in javascript - just faster. For device-centric authentication, we want cryptographic keys to be exercised only after the user locally authenticates. While theoretically this could be done in software (only use the software key after you locally check - also in software - the user's credentials), in practice the credentials in question could be sensitive data such biometrics, which should not be exposed to applications. Therefore, the use of the keys should be controlled by a layer that is not available to the application (e.g., the key can only be used after a fingerprint is recognised, but the fingerprint data is not revealed to the application that wants to make use of

the key). This - by definition - cannot be done purely in application-provided Javascript. As a result, webcrypto needs to evolve to allow applications access to keys that function differently from “pure software” keys.

Attestation

How, then, will a server know whether the key that was used during an authentication is in fact such a key that can only be used after local user authentication, or whether it is a pure software key (i.e., a key that could have been - albeit more slowly - generated and exercised purely by application-provided javascript)? In other words: how does the server know whether the user was really successfully authenticated when it receives a cryptographic assertion from a client? The answer is attestation: when the key is registered with the cloud service, it comes with a cryptographic assertion convincing the server that this key requires local user authentication before it can be used. (Strictly speaking, U2F attestations establish that key usage requires presence of *some* user, not authentication of a specific user. U2F’s sister protocol UAF, however, does support local user authentication.)

Privacy

Strong authentication - if not done right - has the potential to be in conflict with user privacy. For example, if the above attestation reveals a permanent identifier of the certified client device, then this would allow trackability of users across key registrations (i.e., two keys registered from the same device would be identifiable as such, even if the user has otherwise taken precaution to not conflate different online identities).

Another requirement mandated by our desire to design for user privacy is that cryptographic keys should not be usable across different relying parties - something that webcrypto already provides today.

Application Identity

Users interact with a given application across a variety of platforms - they might visit paypal.com today, and use the PayPal Android app tomorrow. A companion paper submitted to this workshop submitted by FIDO addresses this topic in more detail.

User Interface

History teaches us that authentication mechanisms whose UI is not fully controlled by the relying party fail in the marketplace (examples include HTTP client auth, Cardspace, TLS client auth, and others). While we’re not entirely certain that this reflects causation (as opposed to correlation), we were reluctant to build authentication-related UI into the browser, and instead rely on the relying party to design the totality of the user experience during login.

Remember, however, that certain details of the key usage (such as what local authentication modality is used to unlock the key) might be hidden from the application, which therefore can’t show a user interface that goes into great detail about the authentication (e.g., instruct the user to “swipe their finger” versus “touch the fingerprint scanner” versus enter a PIN, etc.). It’s an

open question on how to strike the right balance between relying-party-controlled UI and UI that is specific to the local user-to-device authentication modality.

User Identity and Cryptographic Assertions

Another lesson we have learned from earlier failed attempts in this space is that user identity is something that belongs in the application layer, and that lower layers (TLS, HTTP) are poorly equipped to deal with user identity. Consider, for example, multi-login, a feature at Google that allows multiple accounts to be logged in at the same time. Such a feature cannot be implemented if the user identity is tied to, say, a TLS client certificate (thus forcing a TLS connection between a browser and a server to be associated with exactly one user identity). Consider another example: in addition to allowing multiple accounts to be logged in at the same time, Google also considers users to be “logged in as” Google+ Pages or YouTube Channels that the user has permission to manage. This means that there is a many-to-many relationship between user identity and user credential: A single user credential (say, a password) can represent multiple user identities (e.g., a gmail account and one or more Google+ Pages and YouTube channels); and one Google+ Page can be associated with multiple user credentials (the passwords of the various users allowed to manage that Page).

FIDO U2F doesn’t get in the way of these various permutations: It simply deals with keys and signed statements produced by these keys - nothing in the protocol refers to user identities. This served us well when integrating U2F with the various Google services, and we believe this should also be the model going forward.

Recommendations for Future Web Crypto APIs

To achieve device-centric authentication, webcrypto needs to evolve in a couple of ways:

Local User Authentication

Web applications should be able to require that certain private keys cannot be used unless the user was locally authenticated. A minimal viable implementation of this requirement would be simply a bit at key-creation time asking the underlying system to gate future key usage by local user authentication.

Attestation

While a simple bit requiring key usage to be gated by local user authentication might be enough, more ambitious implementations could employ key attestation. This would allow relying parties to get more detailed information about key storage (hardware versus software) or the particulars of the local user authentication modality. If we employ attestation, it needs to be anonymous, either by employing batch certification (the approach used in the current FIDO specs), or an anonymising “Privacy CA”, or by taking advantage of the latest advances in speeding up pairing-based direct anonymous attestation (DAA) protocols. We have implemented pairing-based DAA on our U2F hardware, and achieved attestation performance of less than one second - we therefore think that DAA is the right choice going forward.

Cryptographic Key Discovery

Users will likely carry keys created this way with them on mobile devices (smart phone or perhaps a dedicated portable device). An application invoking a webcrypto API on one device should be able to use keys resident on a different (nearby) device. In order to do this, webcrypto needs a mechanism to discover that such keys on nearby devices exist. This discovery mechanism must respect same-origin policy and user privacy.

What We Don't Need

In order to keep the changes to a minimum, we also list here potential additions we believe we can do without:

- *A special API for user authentication:* key generation and signing is enough, as long as the relying party knows that the signature can't get generated without local user authentication.
- *A new user interface*: webcrypto doesn't need a browser-based UI today, and it shouldn't need one for device-centric authentication. The only UI triggered by this use of webcrypto should be that required for local user authentication (i.e., asking the user to enter a PIN, swipe a finger, etc.). This UI typically will be rendered by the O/S (in case of PINs) or by software associated with the biometric verifier.

A Use Case Scenario

Bob has a smartphone with a fingerprint scanner. On the phone, there is a key for example.com that can only be used when Bob's fingerprint is recognized.

Bob walks up to a desktop computer that he hasn't used before, and navigates to login.example.com. The Javascript on the login page calls the key discovery API to discover nearby keys (either locally or on nearby devices), and is informed about the key on Bob's phone.

The Javascript on the login page then calls the sign API, specifying the key on Bob's phone. Since that key can only be used when Bob's fingerprint is recognized, Bob has to swipe his finger across the fingerprint reader on his phone.

Because the key on Bob's phone is associated with Bob, the relying party has now authenticated Bob. It also realizes that the computer Bob is logging in from is new - Bob has never logged in from that computer before. It therefore offers to Bob to generate a keypair on that new computer.

When Bob agrees, example.com's Javascript calls the key generation API, asking for a key that is protected by local user authentication. The call returns an error, indicating that no local user authentication mechanism has been set up yet. The relying party informs Bob about this error, who then proceeds to set up local authentication (for example, he sets up a screen lock with a PIN). Now, when the relying party repeats the call for key generation, it succeeds.

The next time Bob wants log into example.com from this computer, he again navigates to login.example.com. Key discovery now reveals two keys: one on Bob's phone, and one locally on the computer. Preferring the local key, the Javascript on the login page calls the sign API specifying that key. Bob has to enter the screen unlock PIN (he doesn't need to use his phone this time), and is logged into example.com.