WebCrypto++

Combining Traditional PKI with WebCrypto

Background: In many parts of the world proprietary browser “plugins” have been deployed in order to provide missing browser PKI-functionality like the ability to sign transactions etc. Although indeed working, these plugins share a number of weaknesses like:

- Highly platform-dependent making them expensive to roll out to consumers
- Typically run with the same privileges as local applications
- Requiring users to perform explicit installs which may not be allowed on enterprise-managed computers

The W3C WebCrypto [http://www.w3.org/TR/WebCryptoAPI] WG has (among numerous of other things) taken on this as a target.

The following *conceptual specification* outlines a foundation for creating lightweight JS/HTML5-based “plugins” having similar functionality as the proprietary plugin schemes, but without the mentioned disadvantages.

Preconditions

This specification depends on that the user has one or more keys supplied in a smart card, “soft token”, or in an embedded hardware-based security solution. One possibility which has been mentioned is that the user in some way grants unknown web-code (or sites) access to specific keys. Unfortunately this method has major security and usability disadvantages.

To facilitate a more *manageable* access control model, this specification builds on the current WebCrypto API specification which *indirectly* mandates that keys residing in platform-wide keystores are *inaccessible* from WebCrypto, while introducing a mechanism that enables such keys to *optionally ”transcend”* through the use of additional security constructs.

The Foundation – Keys Trusting Code

In the described scheme, it is the *issuer* of a key that (more or less) unilaterally decides the policy for a key; the user’s only “right” (and obligation) is protecting it from theft etc. This conforms to most existing payment-, government- and enterprise-keys and may also be a prerequisite for *certification*.

To make this possible, *issuers* selectively grant code *they trust* the permission to access *“their”* keys by applying an application-oriented ACL (Access Control List) to such keys during the key provisioning phase.

The K2C (Key to Code) trust-arrangement enables *custom “plugin” code to run with the same (limited) privileges as other web-code*. Users should not need to explicitly grant privileges to code or keys, but rather be able aborting the entire operation which should be a part of any well-designed application.

Virtual Domain for SOP “Emulation”

A core WebCrypto feature is the reliance on SOP (Same Origin Policy) for protecting keys from unauthorized access. To facilitate cross-origin-operations `postMessage()` operations can be used.

The described scheme depends on that key-using applications are *digitally signed* for creating cryptographically provable links between applications and associated key ACLs. Each signed application should internally (in the browser), be represented as a unique “Virtual Domain” which is how cross-domain protection is maintained.

To make this scheme reasonably straightforward implementing in a browser, signed applications should probably be restricted to window objects such as IFRAME.

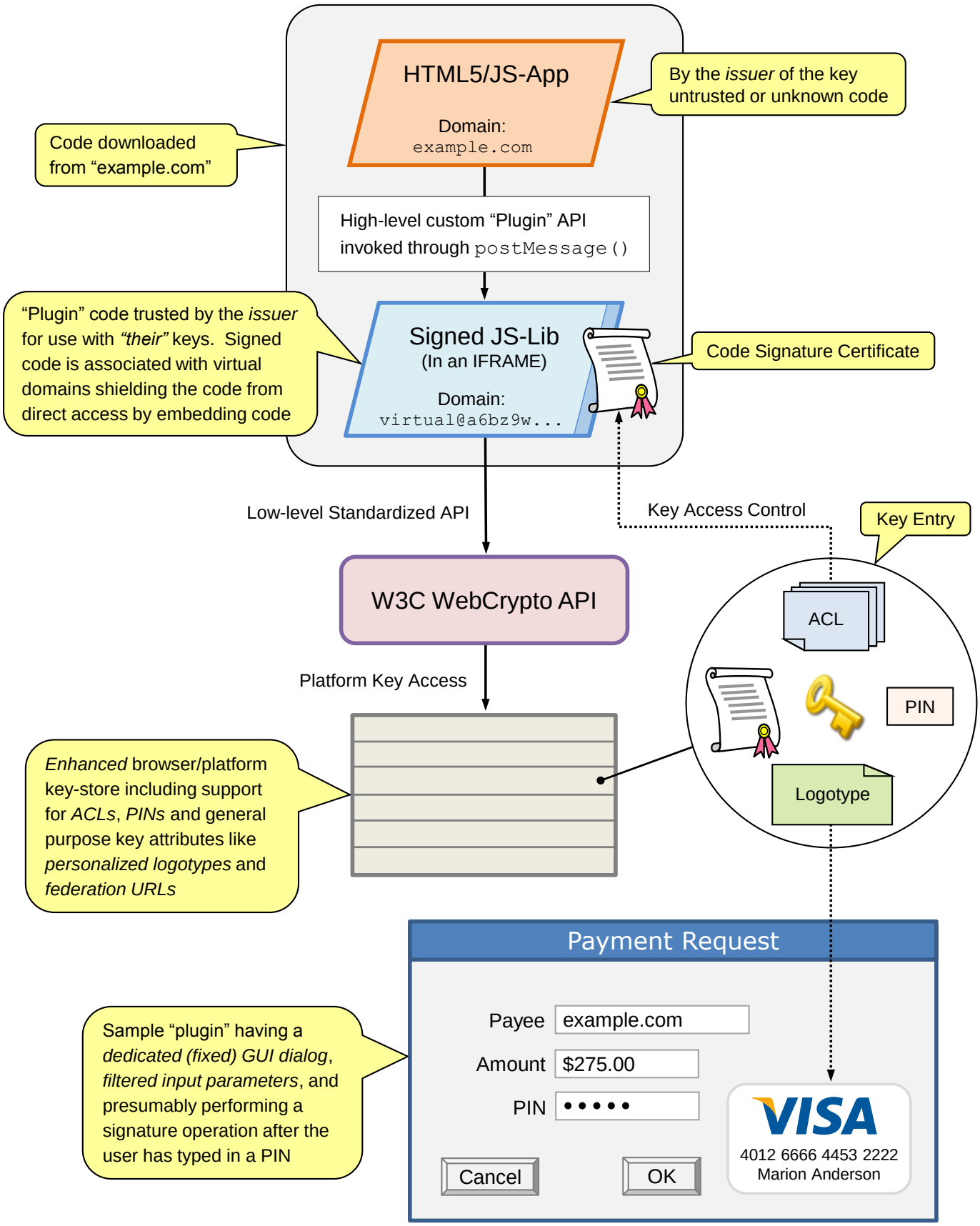
Elimination of Issuer Web Support in Cross-Origin Operations

Since signed code can be securely distributed through arbitrary means, *there is no longer a requirement on issuers running specific web-sites for supporting cross-origin operations*; relying parties only have to package WebCrypto “plugins” with their own application code *like they already do with other libraries*. In fact, the ACL-scheme does not have to map to an issuer; an ACL may equally well identify one or more providers of trusted third-party software.

Federation without the “NASCAR” Syndrome

The library capability can also be used to make *federated* authentication and payment systems less confusing to use as well as more scalable by supplying the WAYF (Where Are You From) information (URL, logotype, etc.) in specific *key attributes* rather than through myriads of *statically configured* federation partner logotypes. Only *personalized* logotypes matching the user’s resources *and* the service provider’s requirements need to be displayed and selected from.

Architecture Overview



The above is an attempt catching the entire concept in a single picture

API Examples

The following sections contain a few examples on how the WebCrypto API could be extended to support the described "plugin" scheme. Here supplied in JavaScript notation rather than Web IDL. The new elements are marked with **blue**.

Finding Keys

Assuming that the keys we are interested in reside in the browser/platform keystore we need a platform-independent way of finding them.

```
// The following promise should return with an array of "CryptoKey" objects
window.crypto.subtle.KeyStore.enumerateKeys(window.crypto.subtle.KeyStore.PLATFORM)
.then (function (foundKeys) {
    // Successful call, any keys?
    if (foundKeys.length == 0) {
        fail and exit…
    }
    // Select a key to use, here just the first one
    var key = foundKeys[0];
```

One could imagine supporting a discriminating argument like SMARTCARD or a bit-field with attributes like CONNECTED, EMBEDDED, HARDWARE etc.

The call to **enumerateKeys()** will only return the keys that the invoking application (Signed JS-Lib in the Architecture Overview) through the keys' ACL declarations has acquired the permission to access.

Using Keys

Now we are ready using the "regular" WebCrypto API!

```
window.crypto.subtle.sign(AlgorithmIdentifier, key.privateKey, Data2Sign)
.then (function (etc….
```

A WebCrypto implementation must distinguish between keys associated with **KeyStore** *and keys having "web-storage" since access control must be enforced for every call*

ACL Contents

To accept signed code the origin of the signature must be established. Since it is not the platform (*for the described "plugin" concept NB*) that trusts code-signatures, but individual keys, ACLs must as a minimum contain a CA certificate or the actual signature certificate itself. To make the key protection scheme more universal, an ACL could also indicate support for "system" applications like browser-, email- and VPN-clients.

Key Attributes

Many "plugins" presumably need to find out various and usually *composite* key attributes like associated X.509 certificates, supported algorithms, etc. In the Architecture Overview sample, a logotype element was also included. The definition of key attributes is yet TBD. The extension method itself could be like the following:

```
// The following call should return an attribute-specific object
// or null if there is no matching attribute
var keyAttr = key.getKeyAttribute(AttributeTypeURI);
```

Key Authorization

In the depicted payment scenario a specific GUI element was used for gathering key authorization data (PIN). When the user clicks OK, the authorization data would be transferred to the target key through the use of an additional method implied by this specification:

```
key.authorize(AuthorizationData);
```

If an application provides no explicit authorization data for a key that needs authorization, the browser should automatically

request the user for key authorization through a system-specific pop-up dialog launched immediately before the actual crypto-operation is to be performed.

Privacy Considerations

In spite of building on a technically quite different foundation, the described concept doesn't appear to depart privacy-wise from the original WebCrypto API scheme. *Nothing prevents malicious issuers from subjecting their clients to various attacks or misfortunes*.

If an issuer wants to limit cross-origin usage of its keys, a JavaScript "plugin" can discriminate caller domains exactly as much as it considers appropriate, like only accepting requests from domains like "**`*.gov.uk`**".

Security Considerations

Although signed code protects against modified code, *actual security* is neither worse nor better than for WebCrypto since a "broken browser" could provide the proper signature to the platform but run an entirely different piece of code.

This limitation applies to most other systems well. In an unhealthy environment, code-signatures add little or no value.

However, unlike schemes relying on SOP, *signed code and ACLs do protect against DNS poisoning or fake TLS certificates trying to circumvent the key access control provided by SOP*.

Protection of private and secret key-material can be as strong as is technically possible since this part is not bound to the browser/web level.

Note that *relying party code never gets direct access to key*s, it can only **`postMessage()`**.

Third-party software may require additional validation by issuers.

Mobile Device "App" Support

The devised key access control method should be equally applicable for Apps as featured in Android, iOS, or Windows 8.

Key Provisioning

Due to the fact that this specification relies on non-standard key attributes, key provisioning (certificate enrollment) becomes crucial. A related project, SKS/KeyGen2 [https://code.google.com/p/openkeystore] supports the all the necessary features including ACLs, Logotypes, PIN-codes, etc.

Alternative to "Trusted Applications" Running in a TEE?

The described scheme exhibits similarities with concepts based on "Trusted Applications" running in a TEE (Trusted Execution Environment).

Security-wise the primary difference is that a TEE is supposed to be less vulnerable to malware attacks than code running at the application layer.

Implementation-wise however, *TEE-based concepts are usually crippled by code-size and deployment restrictions not to mention a much more limited and often platform-specific programming environment*. It is also quite tricky bypassing the "regular" I/O system of an operating system which is necessary in order to achieve a "Trusted UI".

What's more, even a "Trusted UI" is open to spoofing attacks since a *typical* user cannot really distinguish between a "genuine" and "fake" PIN UI to take a concrete example. In the sample scenario keys would only accept a payment authorization from trusted (known) code which means that a successful PIN spoof attack would be ineffective unless the attacker has direct access to the device itself.

> *This proposal can be regarded as a "TEE-light", enabling the development of "Trusted Web Apps" based on standard web-technology including transiently downloaded code, not requiring any specific installation-step*

Author

Anders Rundgren, anders.rundgren.net@gmail.com

IPR

An earlier version of this document has been filed as a defensive publication at: http://ip.com/IPCOM/000229430