Steve Lewontin

2 November 2009

Policy Based Device Access Security

Position Paper to Device Access Policy Working Group

Steve Lewontin

2 November 2009

1. INTRODUCTION

Nokia has implemented a policy-based device access security model for its Web runtime components: Web widgets and the browser. This paper describes those elements of the model that may be suitable for incorporation into a device access policy standard.

The Nokia model describes a policy-based mechanism for making access control decisions when device APIs are invoked from Web content. As such the model specifies how policies are described and how they are used to make access decisions. In other words, the model describes policy mechanisms but does specify policies themselves, although policy requirements and possible policy defaults are clearly within the scope of a device access policy specification. The model also leaves out several other important aspects of the whole system: for example, the model provides a mechanism that can be used to map from Web content attributes, such as digital signatures, to access rights, but does not specify digital signatures. Similarly, the model provides a mechanism that can be used to get user input to access control decisions, but does not say anything about the user interface model, although it is widely recognized that the proper implementation of user interaction is essential to good security.

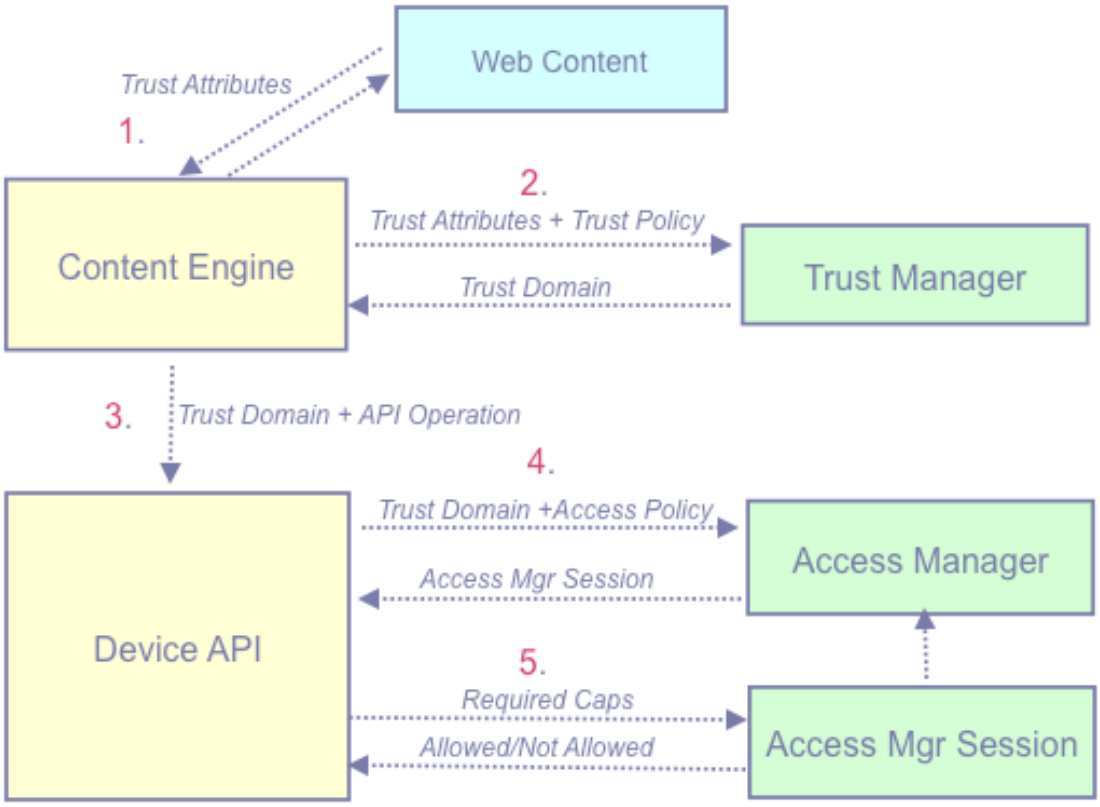
The behavior of the system is described in terms of a *policy engine* and the algorithms used by the policy engine to make decisions. The security engine is treated as an external component to the Web browser or other execution context that is rendering Web content and that accesses device services on behalf of that content. This means, for example, that the model does not cover JavaScript or HTML security—for example, the sandboxing of JavaScript contexts—although JavaScript and HTML security models are clearly important in determining correct device access policies. The model also describes how the security engine can interact with the components that render Web content—for example, the browser—but this interaction is provided as an example and is not intended to propose required behavior.

2. SECURITY MODEL

Access control decisions are needed when Web content—for example, a Web widget—accesses local services and data. The security engine does not itself control access, rather it acts similarly to what in XACML and SAML terminology is known as a *Policy Decision Point* (*PDP*). The security engine takes data about a service access request and uses this to compute an access decision based on trust and access policies. When accessing a local service, it is assumed that trusted code executing in the content engine context calls the security engine to get a decision about whether to allow access, but it is the trusted code that actually enforces the access decision, acting, in XACML or SAML terms, as a *Policy Enforcement Point* (*PEP*).This model implies that the security engine has nothing to say about how or when (or even if) access control is applied. This is up to the implementation of the Web content engine and/or of the device API implementation.

An example of the interaction between Web content, the content engine (for example, the browser), the device API implementation, and the policy engine when making an access control decision is as follows:

Steve Lewontin 2 November 2009



1. The content engine loads the content and gets any needed content trust attributes, such as the origin URL or the digital signature.

2. The content engine queries the trust manager to get the *trust domain* of the content, passing the relevant trust attributes, and the path to the appropriate trust policy to the trust manager.

3. The content engine makes an API request, passing the trust domain to the device API implementation.

4. The device API implementation creates a *security session* with the access manager, passing the path to the appropriate access policy and the content trust domain.

5. The device API implementation asks the security manager for an access decision (via the security session) passing the required capabilities. Based on the result of the access control decision, the service invokes the requested operation or throws a security exception.

This sequence may vary depending on the application launching the content and how the service is implemented. For installed content such as widgets, steps 1 and 2 may also be carried out by the installer, which stores the trust domain in the application registry where it can be retrieved by the Widget engine. Similarly, session state may be persisted so that, for example, user granted blanket permissions can be kept for use when the installed content is instantiated again. Also note that steps 4-5 can be carried out by the Web content engine rather than the service API implementation.

3. POLICY

The security engine is policy driven so that trust and access control policies can be customized to a specific device and content engine instance instance. For example, a mobile operator may want to customize trust and access policies for their phones. Also, Web and widget content may use different policies.

Policies are described in XML format via policy files, which are stored in protected locations so that they can only be modified by the policy "owner" (for example, the mobile operator). Policies have two components: trust policies and access control policies.

3.1 Trust Policies

A simple trust policy file is:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<trustpolicy>
  <defaultdomain name = "Untrusted"/>
  <domain name="NokiaService">
    <origin url = "http://www.nokia.com/services"/>
    <origin url = "http://www.nokia.com/services/music"/>
  </domain>
  <domain name="NokiaPublic">
    <origin url = "http://www.nokia.com"/>
  </domain>
</trustpolicy>
```

The trust policy provides mappings between *properties of content artifacts* and *trust domains*. In this case, the policy file provides mappings for several origin urls: Each domain is named in a `<domain>` tag containing one or more `<origin>` property tags. The policy also names a default domain for properties that are not mapped in any domain section. Policies can also name other properties, such as certificate fingerprints.

Note that the mappings specified in a trust policy file are raw mappings that may not correspond to the property values associated with a code artifact. For example, content from *http://www.nokia.com/services/maps* to the *NokiaService* domain while *http://www.nokia.com/products* would be mapped to *NokiaPublic* by same-origin best matching rules. Property-specific matching rules are applied by the trust manager to translate actual content artifact properties to the mappings specified by trust policies.

3.2 Access Policies

A sample access control policy is:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<policy>
  <!--  an alias groups a set of capabilities under one name -->
  <alias name="UserDataGroup">
    <capability name="ReadUserData" />
    <capability name="WriteUserData"/>
    <!-- capability name="Location"/-->
    <capability name="UserEnvironment"/>
  </alias>
  <alias name="NetworkGroup">
    <capability name="NetworkServices"/>
    <capability name="LocalServices"/>
  </alias>
  <alias name="DeviceResourcesGroup">
```

```
    <capability name="MultimediaDD"/>
    <capability name="ReadDeviceData"/>
    <capability name="WriteDeviceData"/>
    <capability name="CommDD"/>
    <capability name="SurroundingsDD"/>
    <capability name="NetworkControl"/>
  </alias>

  <domain name="Untrusted">
    <!-- always granted capabilities for this domain -->
    <capability name="UserDataGroup"/>
    <capability name="NetworkGroup"/>

    <!--  user-grantable capabilities for this domain -->
    <user>
      <defaultScope type="session" />
      <scope type="oneshot" />
      <scope type="permanent" />
      <capability name="DeviceResourcesGroup"/>
      <capability name="Location"/>
    </user>
  </domain>
  <domain name="OperatorSigned">
    <capability name="UserDataGroup"/>
    <capability name="NetworkGroup"/>
    <capability name="DeviceResourcesGroup"/>
    <capability name="Location"/>
  </domain>
</policy>
```

The basic function of the access control policy is to define the capabilities assigned to a set of trust domains. Each **<domain>** tag defines the capabilities of one domain. Within each domain section the capabilities are listed by **<capability>** tags. Capabilities listed within the domain section itself are granted unconditionally. Capabilities that can be granted based on user choice are listed in one or more **<user>** sections under the domain.

Each user section defines the scopes for which a user is allowed to grant access, via one or more **<scope>** tags. The possible scopes are *one-shot*, which allows a single access, *session,* which allows access for the duration of an application session, and *permanent*, which allows access for the current and future sessions until revoked by the user.

The optional **<defaultScope>** provides a hint as to which scope should be offered as the default to the user (Note that **<defaultScope>** is also a scope, so no scope tag with the same value is required.)

User choice is typically implemented by prompting, but the security engine does not control how user choice is offered. Instead, the security engine supports a *userConditionHandler* callback interface so that the calling code can implement user choice in whatever style it wants. For example, the user can be prompted for a yes/no decision on the default scope or can be offered a list of all the allowed scopes with the default scope set as the default choice. Similarly, the user can be prompted only once for all of the capabilites listed in a user section, or can be prompted individually for each one.

User choices are an example of *conditional capabilities* (similar to conditions in the OSGI R4 security model). These are capabilities that are are granted based on some runtime condition—in this case user choices made in response to prompts. User capabilities are the only conditional capabilities currently implemented, but the underlying architecture supports a generic condition model so that other condition types can be easily implemented (for

example, a capabilities granted based on network connectivity could limit high-bandwidth, expensive network operations to cases where a device has LAN connectivity).

From the point of view of the security engine, capabilities are simply names: they have no inherent semantics. Access checking simply compares the names of required capabilities against the names granted by policy. As a matter of convenience, policies can also define **<alias>** tags that create new names for sets of capabilities, but these are subject to slightly different processing rules when computing access decisions.

3.3 Access Computation

3.3.1 Capability tests

The security engine calculates access decisions by implementing an *isAllowed()* operation that uses three parameters: a *list of required capabilities*, a *trust domain*, and a *policy*. Clients of the security engine access this operation via a *security engine session object* which specifies the trust domain and policy. The trust domain is supplied by the client when creating the session. The security engine maintains an internal representation of the policy. The security engine creates this by reading a *policy file* or by recreating a policy from a *persisted policy.* (See Section 3.3.4) The client supplies either the policy file or a *persisted policy key* when creating the session. The client invokes the *isAllowed()* operation, passing the list of required capabilities as a parameter.

1. The security engine generates an *allowed capability list* from the input *trust domain* and *policy.* The allowed capability list is an unordered list of *unconditional* and *conditional capabilities*. Unconditional capabilities contain a capability name. Conditional capabilities contain a capability name and a reference to a *condition*. Conditions implement an *isMet()* operation that the security engine uses to decide whether to grant the capabilities associated with a condition. For user conditions, each condition contains an unordered list of *allowable scopes* and any *default scope hint*. User conditions also hold the *current grant state of session and permanent grants*. Each allowable session and permanent grant can be marked as one of *untested*, *granted*, or *denied*. Finally, user conditions contain a reference to *user condition handler callback operation* implemented by the client. User conditions are shared among all of the conditional capabilities listed in a **<user>** section of the policy, so that all capabilities in a section share the same *current grant state*.

2. For each capability in the required capabilities list:

   - If the required capability is found in the allowed capability list and is *unconditional*, processing continues to the next required capability.

   - If the required capability is found in the allowed capability list and is *conditional*, the security engine calls the condition's *isMet()* operation (see 3.3.3). If this returns *false*, processing stops and *isAllowed()* returns *false.* If *isMet()* returns *true*, processing continues to the next required capability.

   - If the required capability is not found in the allowed capability list, the security engine checks for any aliases that name the required capability. If none are found, then processing stops and *isAllowed()* returns *false*. If aliases for the required property are found, then the security engine checks to see if at least one of them is unconditionally or conditionally allowed. If one of them is allowed,

processing continues to the next required capability. If none of them is allowed, then processing stops and *isAllowed()* returns *false*.

If processing reaches the end of the required capability list without returning false, *isAllowed()* returns *true*. Note that the required capability list is treated as an unordered set and that all required capabilities must be granted for *isAllowed()* to return true.

3.3.2 Ambiguities

Policies allow a capability and one or more aliases that name the capability to be listed in a trust domain. According to the rules specified in 3.3.1 if a required capability is explicitly named in the allowed capability list, then its aliases will not be checked. This means that if a conditional check of an explicitly named capability fails, then the capability will not be allowed even if some alias would have allowed this capability. Also note that an alias named in the required capability list is checked in the same way as any other required capability. If both a capability and an alias that names that capability are specified in the required capability list, both the capability and the alias capability checks must succeed for *isAllowed()* to return true. These rules insure that there will be no ambiguity when both a capability and an alias that names this capability are both listed in a trust domain.

Policies do not allow a capability or an alias to be listed more than once within a trust domain. For example, a capability cannot be listed both as a conditional and unconditional capability or be listed in more than one condition within a domain. If a capability or alias is listed more than once, then the results of condition evaluation are not defined.

Finally, each condition section groups together a set of capabilities that will be evaluated together under the control of a single condition. This means that if the condition is met for any of the capabilities, then it is met for all of the capabilities. In some cases, it may be desirable for capabilities with the same conditional test (for example, user conditions that name the same default and allowed scopes) to be evaluated separately. In such a case, the capabilities should be named in different conditional sections.

3.3.3 Condition Evaluation

Each condition supports an *isMet()* operation. For user conditions, this is calculated as follows:

1. If the the allowable scopes contains session scope and the current grant state marks session scope as granted, *isMet()* returns *true*.

2. If the the allowable scopes contains permanent scope and the current grant state marks permanent scope as granted, *isMet()* returns *true.*

3. If the condition contains a non-null reference to a user condition handler callback the user condition calls the handler. If the handler returns *false*, *isMet()* returns *false*. If the handler returns *true*, *isMet()* returns true. If the user condition handler reference is null, *isMet()* returns false. The user condition calls the user condition handler with three *in* parameters—the list of capability names associated with the condition, and the list of allowable scopes, and the default scope hint—and one *in/out* parameter—the *current grant state*. The user condition handler typically uses these parameters to prompt the user, but it is up to the handler to decide what prompting behavior to implement. The user condition handler is expected to set the current grant state to reflect any granted scopes. (Note that marking a scope as

denied is meaningful only to the condition handler: this is not used by the condition's isMet() logic which only checks for grants. The denied flag can be used by the user condition handler to decide whether to prompt again for a grant that was previously denied during a session.)

3.3.4 Persistence

The security engine can serialize the current state of user grants during a session. When a session is persisted, the current grant state of any *user permanent grants* is persisted with the session. Neither session grants nor denials are persisted. When a session is recreated from persistent state, an persisted permanent grants are marked as granted in the current state of user grants. All other scopes are marked as untested.

3.4 Capabilities

Policies are written in terms of *capabilities* but the policy model says nothing about the semantics of capabilities. For example, the policy model would allow capabilities to be treated simply as opaque strings. However, a specification of device APIs must include the capabilities required to access each operation or service in order to make it possible to write meaningful policies and to successfully deploy content. For example, content might need to be signed, based on policies and the capabilities required to access the APIs invoked from the content.

Furthermore, in order to create a device API ecosystem that is consistent across a wide variety of platforms, a device API standard probably needs to say something about capability semantics: for example, to specify a set of capability names and to describe how they are intended to be mapped to the kinds of operations and data relevant to device APIs. In other words, capability semantics are clearly relevant to device access policy standard specification.

Nevertheless, capability semantics are not specified here because it seems advantageous to define a policy model that is independent of specific capability models. For example, capability models may be abstract, defining capabilities for generic operations like reading and writing, or they may be concrete, specifying capabilities for specific interfaces or operations on those interfaces, such as *sendMessage.* In our opinion, concrete capability models are easier to specify, easier to interpret and more extensible, but the Nokia policy model was specifically designed to support both abstract and concrete capability models since we needed support existing abstract capability models native to some Nokia devices.