Steve Lewontin

12 November 2008

Web Runtime Policy Based Security

Steve Lewontin 12 November 2008

1. INTRODUCTION

The *Nokia Web Runtime (WRT)* provides a common engine for rendering Web content—HTML, JavaScript, CSS, and other resources—across all Nokia platforms. The WRT is based on the WebKit open source browser engine with many extensions to support mobile browsing. The WRT is used in a variety of contexts: for rendering Web pages in the browser, for executing Web widgets and Web applications, and for displaying Web content embedded in other applications. This means that the WRT may be present in multiple instances in a device. For example:

- Nokia's Web browser applications embed the WRT within platform-specific browser UIs.
- Web Widgets and Web applications are launched within a “chromeless” instance of the WRT
- Other Nokia applications and services embed the WRT within their UIs.

One of the key facilities provided by the WRT is to give Web content access to local device services—location, camera, sensors, etc.—via JavaScript. This is a very powerful extension to the capabilities of Web content running in the WRT, but it raises serious security concerns since access to local services can compromise user data and privacy. To deal with this the WRT provides a security engine for controlling access to local services.

Local service access is potentially available from any context that instantiates the WRT. Because of this, the security engine must be able to assess the capabilities that allowed to specific content running in a specific WRT instance and make an appropriate access control decision.

2. SECURITY MODEL

The WRT security engine provides APIs for making access control decisions. Access control decisions are needed when content executing in a WRT instance—for example, a Web widget, a Web application, or a Web page—accesses local services and data. Such access control could be provided by extending and modifying the built-in sandboxing provided by the JavaScript runtime, but both the mechanisms and the methods are highly browser-specific and not easily suited to the task of controlling access to local services and data. Instead, the approach has been to define the security engine as an external component that can be invoked from the WRT by a variety of mechanisms.

The model used by the WRT security engine is that the WRT itself and any local service invoked from the WRT are trusted. In addition, any application that creates a WRT instance is also trusted. “Trusted” in this case means that these elements are native code whose capabilities are controlled by the native operating system. The WRT security engine operates on behalf of these trusted elements, and cannot control access to local services by these trusted elements themselves. Instead, the WRT security engine is used by these trusted elements to determine what access to grant to Web content that is running within a WRT instance. The security engine therefore acts as a “throttle” that the WRT can use to limit access to device services to less than or equal to the access granted by the OS to the WRT and other native elements.

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, trusted code 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 WRT itself (specifically of the WRT local service framework) and/or of the local service implementation.

An overview of the basic steps to make an access control decision are shown in Figure 1. These operations are described in more detail in Section 4.2.

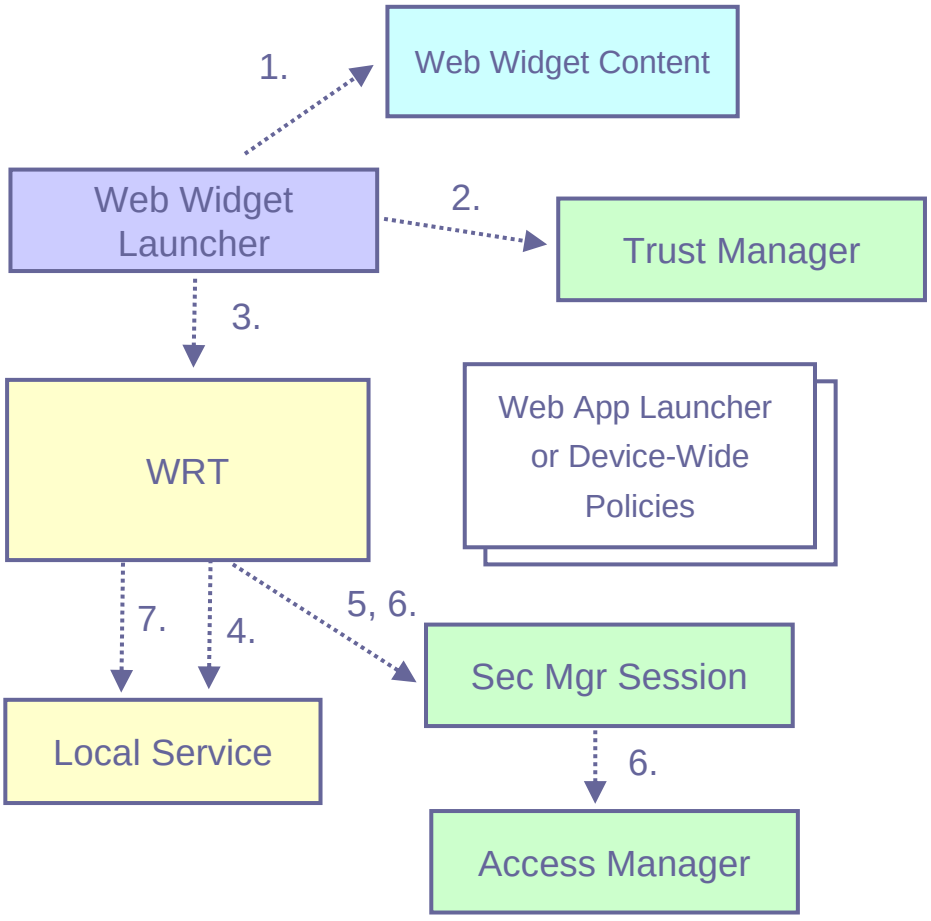

Figure 1: WRT- Security Engine Interaction

The diagram shows how this happens when the WRT is instantiated by the Web widget launcher to run widget content, but the basic steps are similar in other cases.

1. The launcher loads the content and gets any needed content properties, such as the origin URL or the digital signature.

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

3. The launcher instantiates the WRT to render the content, passing the trust domain and the path to the appropriate access control policy.

4. When access to a local service is requested by the content, the WRT gets the *required capabilities* from the local service implementation.

5. The WRT creates a security session with the access manager, passing the path to the appropriate access policy and the content trust domain.

6. The WRT asks the security manager for an access decision (via the security session) passing the required capabilities.

7. Based on the result of the access control decision, the WRT loads the service and invokes the requested operation.

This sequence may vary depending on the application launching the content and how the service is implemented. For example, for Web pages launched from the browser or for Web apps that are launched directly from the Web, step 2 can be carried out directly by the WRT based on the URL. 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 launcher.

Steps 4-6 require cooperation between two trusted elements: the WRT and the invoked service. The WRT must supply the trust domain and the invoked service must supply the required capabilities. But the actual interaction with the security engine can take place from either component. For example, the WRT can query the local service implementation about required capabilties or read these capabilities from service meta-data and then call the security engine. Or the WRT can pass the trust domain to the local service and the service can interact with the security engine.

Having the WRT interact with the security engine is convenient for service implementations and allows consistent access control across all services. On the other hand, having the service interact with the security engine allows the service to control how access decisions are made. For example, when an access control is applied at the granularity of the service interface, then it is simple for the WRT framework to make a decision before instantiating the service object. But when access control is applied at the granularity of individual service methods, it may be more convenient for the service to make decisions.

Not shown in the diagram is the mechanism used to save session state on behalf of content that may persist access control state from one session to another (for example, storing user-granted capabilities.) This requires the launcher to provide a path to protected storage where persisted sessions for that launcher can be kept.

3. POLICY

The security engine is policy driven so that trust and access control policies can be customized to a specific device and WRT instance. For example, a mobile operator may want to customize trust and access policies for their phones, and a service provider may want to provide policies tailored specifically to their service model (This requires the service provider to invoke services via a provider-specific instance of the WRT.)

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. Trust policies will be described in a later version of this document. 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, although these are subject to slightly different processing rules when computing access decisions.

3.1 Access Computation

3.1.1 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 3.1.3) 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, the security engine checks whether the allowed capability list contains the required capability. If not processing stops and *isAllowed()* returns false. If the required capability is found in the allowed capability list and is unconditional, processing continues to the next required capability. If a required capability is found in the allowed capability list and is conditional, the security engine calls the condition's *isMet()* operation (see 3.1.2). If this returns *false*, processing stops and *isAllowed()* returns *false.* If *isMet()* returns *true*, processing continues to the next required capability. 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.

3.1.2 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.)

3.1.3 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.

4. SECURITY ENGINE DESIGN

This section provides a more detailed overview of the security engine design.

4.1 Components

Figure 2 shows the major security engine components.

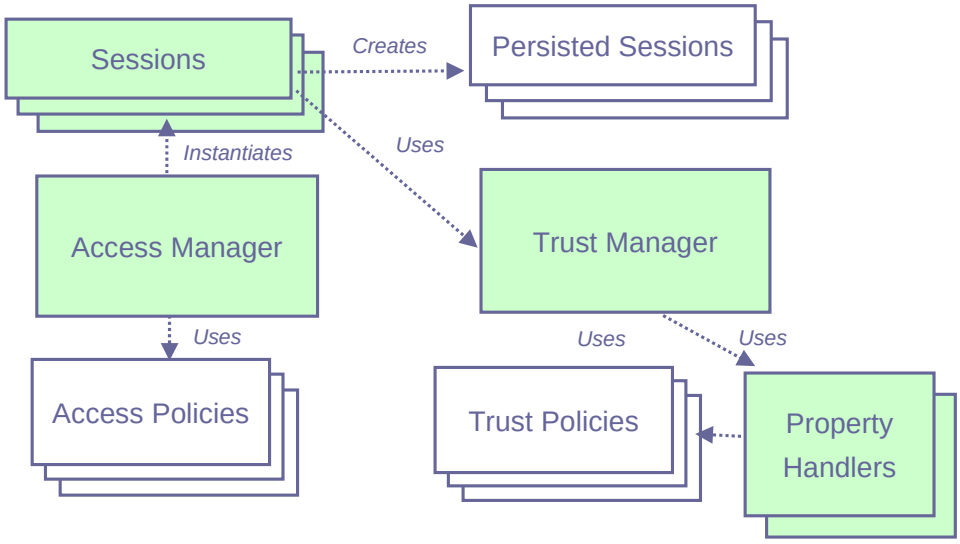

Figure 2: Security Engine Components

Access Manager	Computes access decisions based on a *trust domain*, *access policy* and *required capability set*.
Trust Manager	Maps content properties such as signatures and origin to *trust domains*. These mappings are carried out by *property handlers*, based on *trust policies*.
Session	Holds state for a client access control session.

4.2 WRT and the Security Engine

The WRT uses the security engine for two operations: getting the trust domain for content, and requesting access control decisions when the content accesses a protected interface. Access control decisions are requested from a security engine client session, so a preliminary step to requesting access control decisions is to create a session.

4.2.1 Getting the Trust Domain

The trust engine maps content properties such as content origin and signatures to trust domains. The trust engine dispatches handling of specific properties to property handlers, which know how to map from a specific property to a trust domain. Property handlers may

use trust policies to carry out these mappings. For example, a policy can specify the set of origin URLs corresponding to a trust domain.

Trust domains may be assigned when content is installed or when content is run by the WRT. In the first case, the installer asks for a trust domain from the trust engine and saves this securely to the installed content registry. When content is run, the WRT can retrieve the saved trust domain from the registry. In the second case, the WRT gets the trust domain directly from the trust engine when the content is run. In both cases, the WRT must maintain a reference to the trust domain associated with the content in such a way that the trust domain can be specified when requesting an access control decision.

4.2.2 Creating a Session

Security engine sessions embody access control state for an access control policy, trust domain, and content context. The WRT can construct a session in two ways:

- The first time the WRT creates a session it supplies a trust domain and access control policy. At the end of a session, the WRT can save the session state to persistent storage and retrieve a key that can be used to reinstate the session with any persistent session state.
- The WRT can create a session based on a previously saved session by by supplying a session key. To maintain persistent sessions on behalf of content, the WRT saves the session key to the installed content registry.

An access control policy is set for the WRT when a WRT instance is created. This policy may be a common policy for the device, or it may be tied to a specific WRT instance. The WRT uses the specified policy when creating all sessions.

The WRT can get a trust domain as described above or, it can get a saved session key from the installed application registry.

4.2.3 Asking for an Access Control Decision

Access control decisions are requested from a security engine session, by passing a list of required capabilities to the session *isAllowed()* operation. (See Section 3.1). Required capabilities are typically specified for a service interface as a whole, but some services may specify capabilities per operation. Service implementations can specify required capabilities as meta-data. In this case, the WRT service framework reads the required capabilities and requests an access control decision before instantiating the service object.

In some cases, a service implementation may want to control access on a per-operation basis so that content that does not have the capabilities to invoke all of the operations of the interface can nevertheless invoke some operations. In this case, the WRT delegates access control decision making to the service implementation. This is supported by passing a reference to the current access control session to the service implementation.

4.2.4 User Prompting

Policies may grant some capabilities based on user prompting. Prompting is not implemented by the security engine since prompting styles may vary across implementations and instantiations of the WRT. Instead, prompting is delegated to the WRT via a user prompt handler interface. (See Section 3.1.2). An instance of the WRT can

implement its own user prompt handler or it can further delegate prompting to service implementations.

4.2.5 Storage

Policy files and persisted sessions require secure storage. Policies may be device-wide, in which case they must be readable but not modifiable by WRT instances. Or policies may be per WRT instance, in which case they must be stored in private storage for each instance.

Persisted sessions must be stored in private storage for each WRT instance and the WRT must guarantee that a session maintained on behalf of one content instance cannot be accessed by another content instance.

5. REFERENCES

1) *Recommended Security Policy for GSM/UMTS Compliant Devices,* http://jcp.org/aboutJava/communityprocess/maintenance/jsr118/**MIDP**_2.0.1_MR_addendum.pdf

2) *eXtensible Access Control Markup Language (XACML) Version 1.0*, http://www.oasis-open.org/committees/download.php/2406/oasis-xacml-1.0.pdf

3) *OSGi Service Platform, Core Specification Release 4*