PRIVACY IMPLEMENTATION RECOMMENDATIONS

The following recommendations are intended as a way to begin a conversation about implementation. Please see http://demo-us.ado.ai.accenture.com/murrayw-testbuilds/w3c/webapp/samplePage.html as a (quickly devised) set of sample code that demonstrates some of these concepts.

REQUIREMENTS

The privacy implementation should be as simple and clean as possible. The balance between simplicity and comprehensiveness is a delicate and important matter. Something that is too weak to enable enforcement may cause the entire specification to be ignored, as will something that is too complex to follow.

Privacy information is *metadata* and should be handled accordingly. In other words, information about the level of privacy or security or sensitivity of portions of the data layer is *data that describes data*. It should not be mixed-in with the data layer itself.

Implementation and enforcement should be *modular* and vendor-independent. If good design practices are followed, a privacy management (definition and enforcement) system should allow systems from different vendors to work together; it should be possible to replace vendors without completely disrupting the client's installation. Systems should be loosely coupled whenever possible.

The privacy implementation should be *complete* and *extendable*. This may be our only opportunity to create an interoperable system. There's no reason we can't design a set of functional recommendations that vendors can use to create a healthy ecosystem of tools.

DESIGN RECOMMENDATIONS

Break privacy information into four categories:

1. Type of Privacy (PII, Sensitivity, etc.)
2. Type of Data Usage (Advertising, Analytics, etc.)
3. Mapping of Data Usage to Privacy Type (who should be allowed to access what)
4. Mapping of Privacy to nodes and branches of the Data Layer

The specification should define primary categories for #1 and #2 above, as well as standards for enhancing these categories (adding new categories or creating sub-categories). The specification

should also create a clear and succinct way to define #3 (probably a one-to-many mapping table) and #4.

The last item, mapping privacy to data layer nodes, must be done carefully, and this data should not be intermingled with the data layer itself. I recommend a system akin to CSS rules whereby more specific rules (deep nodes and branches) override more general ones. The privacy standard could include a default set of privacy rules for the most common portions of the data layer. For example, nobody is going to dispute that the top level page identifier has the lowest privacy consideration, whereas payment information is absolutely secure.

Use Inversion of Control (IoC) and Delegation for implementation. By allowing the privacy and security enforcement to be performed by a modularly defined system (the delegate) and be instead defining an interface or "contract", vendors may have opportunities to innovate and offer specialized privacy-enforcement software.

The data layer should not be accessed (read or written) directly. A basic getter/setter mechanism should be defined for both reading from and writing to the data layer. A getter should return a "copy" of either the entire tree or (optionally) a portion of it, thus offering a layer of protection. Getters and setters should be passed an identifier that describes the system that wishes to read or write data. The privacy delegate should use this information, along with the defined privacy metadata (#1-4 in the list above) to allow, deny, or "filter" the results as appropriate.

By forbidding direct access, an implementation may be "enhanced" over time in a way that completely secures/hides the actual data without breaking things.

Note #1: some people may not realize this, but JavaScript ***can*** in fact hide data in a way similar to "private" members in languages like Java or C# or C++. See my sample code for an example.

Note #2: this approach supports some of the work done by the dynamic data subcommittee's work. (Their documentation recommends access via an API, although for different reasons.) In other words, using getters and setters is crucial for both purposes.

If the implementer doesn't care about privacy, a simple get() method should return the entire data layer tree. If implementers and vendors see that a simple digitalData.get() will return the full data tree (or a copy of it), then hopefully they will accept the standard and they can easily opt-in or opt-out from enforcement. That way a "reference implementation" can be as simple as one with a basic "get = function() { return this; }" kind of solution.