ESB Message-audit Standards

1. Logging vs. Auditing

Logging and auditing are not equivalent concepts or activities.

1.1. Logging

A log is informational in nature - tracing execution, transformations, errors, notifications etc.; as useful and important as general logging may be it is typically not adequate to meet regulatory and security standards unless specific information is logged, and certain part of information are obscured or encrypted.

1.2. Auditing

Audit-records are intended to be preserved for a defined period at minimum, and may be retained longer. There are three key aspects that differentiate auditing from logging:

1. specific information is required to be present in audit-logs
2. certain fields, if they appear in audit logs at all, are required to be masked or encrypted.
3. The length of retention of audit-records is often regulated [1: see for example https://www.law.cornell.edu/cfr/text/17/210.2-06]

Audit-logs usually require specific fields to be present; the required fields vary by industry, regulation and also by organization:

Audit-fields fall into the following 3 main categories:

- the actor (who did it)
- action (what they did)
- origin (where they came from, referring system/service)
- date and time-stamp
- contextual information such as account-number, amounts drawn/transferred/deposited etc.

The specific application will determine what additional data must be included, and often these items are mandated by regulations. An example of the meticulousness of audit-field specification can be seen here [2: see *536.B. Globex Order Entry* para. 2 on pg 2]

2. Unified auditing implementation

Due to the wide variation of audit requirements across industries and organizations, a single design or static implementation cannot meet all needs, so we provide patterns and guidance which should help to reduce reinvention and time wastage.

The following are main components of implementation, along with brief motivations:

- an audit-message object-model is needed so that message-assembly can be automated and standards can be enforced.
  - the object model must provide features that allow for selective encryption of audit-message parts (eg. credit-card details, account-number, SSN)
- the framework must support built-in capabilities for automation of message-assembly, encryption, injection of a correlation-ID, etc.
- in its base form, the framework must function seamlessly in API Gateway and MuleESB runtimes.

2.1. data model

An extensible data-model is needed to accommodate variability in the audited information.

See "Reference Implementation for Mule Runtime" for details.

2.2. design and component standards

A consistent naming convention will assist

- designers in specifying requirements, standards etc.
- developers in their self-review of code for compliance
- creation of automated code-quality checks at build-time

2.3. Audit-message disposition

The framework must allow simple configuration to point to an non-proprietary protocol endpoint for reliable, transaction-supporting ,persistent storage of audit-message.

The framework is not expected to support the actual storage of the audit-log messages - delivering these to the endpoint is a complete solution to the problem in hand. For example, a VM-queue that receives a message without error is considered to have successfully audited the message.

NOTE	don’t expect this to be a closed-ended recipe-book. In time we can get to majority coverage of possibilities but this will always need extend and improvement.

2.3.1. Practical considerations for reliability and performance

An auditing component or framework cannot guarantee retention or security, or persistence. Hardware, software and networking failures are inevitable - auditing practices and components can’t eliminate failure in other parts of an organization. General architecture considerations are handled elsewhere - this document focuses on the task of efficiently creating secure, compliant audit-messages.

2.4. Synchronous or Asynchronous Auditing?

Definitions:

"synchronous" : Message construction and commit to transactional-sink is complete before main workflow processing continues

"asynchronous": Raw audit-field data is submitted asynchronously. Pre-processing to generated the audit-message itself executes on a separate thread. Even if raw data is submitted to a transactional endpoint, audit-message loss or delay is possible if raw audit-data is malformed and causes error in the assembly of the message.

The tradeoffs between synchronous and asynchronous are summarized below.

2.4.1. synchronous auditing

- requires audit-activity to completion before the main work can continue
- reduces performance due to blocking while sufficient audit-completion occurs.
- simplifies the detection and rectification (if possible) of any errors that occur during auditing
- if regulations or standards require guaranteed-completion of audit at certain points in the workflow, synchronous auditing is the only option.

2.4.2. asynchronous auditing

(i.e. asynchronously dispatch raw auditing data, not waiting for complete message-construction and disposition)

- risks losing audit information if errors prevent assembly and transactional-disposition of the audit-message
- enhances compute-resource efficiency due to elimination of wait-state
- is not acceptable in cases where transactional-commit of a complete, well-formed audit-message is required.

3. Reference Implementation Mule-runtime

This section describes a pattern for repeatable audit-message creation and formatting. It does not deal with the disposition mechanism itself, downstream reliability, message-delivery or ordering. These are general architecture concerns and are handled elsewhere in M^2.

A reference implementation can be found at M^2MVP GitHub repository

3.1. Conceptual object-model

Below is a proposed canonical JSON object-model that would serve the purpose in numerous verticals (credit-transactions, healthcare etc.) [3: keep in mind this is not generalized secure object-storage - this is just audit-logging]. The idea is that essential elements are standard:

Table 1. Canonical object model

information-item	JSON field-name	comments
unique TX identifier	correlationID + *	
action information	actionData + *	* extensible * arbitrary field-names!
person or agent taking action	agentID + *	* extensible * arbitrary field-names!
time-stamp	timeStamp + *	any format - see tooling
secure fields	secure *	* ***optional*** * extensible * arbitrary field-names! * The field-values are encrypted automatically by the framework before dispatch of message

- items marked * must named as shown - not customizable
- items marked + are mandatory
- ! - these field-names will be replicated as-is in the dynamically generated audit-message string

```
{
   "corrID":"#[message.id]",
   "actionData":{
      "action":"#[flowVars['action']]",
      "source":"#[flowVars['source']]"
   },
   "agentID":{
      "someFieldName":"#[flowVars['someFieldName']]",
      "otherFieldName":"#[flowVars['otherFieldName']]"
   },
   "timeStamp":"#[java.time.ZonedDateTime.now().toString()]",
   "secure":{
      "ssn":"#[flowVars['ssn']]",
      "accountNumber":"#[flowVars['accountNumber']]",
      "thing":"special#[message.id]"
   }
}
```

The person configuring the auditor would defines a template with the necessary extensible fields, according to the industry and the specific application. As long as the JSON is valid and contains the mandatory canonical fields, the object-model will be able to convey the needed information to the message-assembly component.

3.2. Message Assembly Component

The main features of this component are:

- it reads and caches a message-template string at startup
- it extracts the needed fields from the MuleMessage (from pre-defined invocation-properties) and
  - selectively encrypts the some field-values in the JSON-object using a DataWeave script
  - build the overall string according to the configuration pattern

4. Disposition

The wide variety of possible disposition destinations makes bundling such endpoints in the audit-component futile; we recommend that disposition destinations be coded in the integration-flow, or packaged for reuse on a case-by-case basis