MTR 070224

MITRE TECHNICAL REPORT

Open Checklist Reporting Language (OCRL)

A Language for Automated Generation of Reports for Security Guidance Evaluation

September 2007

Charles M. Schmidt
Lisa A. Nordman

Sponsor:	NSA	**Contract:**	W15P7T-07-C-F600
Department:	G026	**Project:**	0707N60C

Approved for Public Release; Distribution Unlimited.
Case Number 09-0182

"©2009 The MITRE Corporation. All rights reserved.

MITRE

Center for Integrated Intelligence Systems
Bedford, Massachusetts

MITRE Department Approval: ______________________________
Jeffrey Picciotto

MITRE Project Approval: ______________________________
Leonard J. LaPadula

Acknowledgments

This work was produced by the System Security Analysis Project, an NSA-sponsored project of The MITRE Corporation. Charles Schmidt was the task lead and Leonard LaPadula was the lead technical developer. The team would also like to recognize the valuable contributions of Shaan Foltz, Lisa Nordman, and Matthew Wojcik in this development effort.

Intentionally Blank

Table of Contents

Section **Page**

1 Introduction and Background 1

2 Using the Open Checklist Reporting Language (OCRL) 3

3 High-level Report Structure 7

4 DataSources 11

4.1 The Open Checklist Reporting Language Data Model 11

4.2 The <WmiDataSource> 12

4.3 The <FileDataSource> 14

4.3.1 The <Instance> Element 14

4.3.1.1 The Structure of <Instance> Elements 16

4.3.1.2 <Property> Elements 19

4.3.2 The <InstanceReference> Element 20

4.3.3 Visualizing Parsing Structures 20

4.4 The <ExecutableDataSource> 23

4.5 The <OvalDataSource> 23

5 Reporting 27

5.1 <Text> Elements 27

5.2 <Select> Elements 27

5.2.1 <Qualifier> Elements 28

5.2.1.1 Complex Qualifiers 30

5.3 <Item> Elements 30

5.4 Putting It All Together 31

5.4.1 Iterating Over Sub-Instances 32

5.4.2 Using Variables 33

6 Conclusions 35

Section 1

Introduction and Background

Machine-readable security benchmarks have a great deal of potential to assist users in determining a system's compliance with a set of security recommendations. Benchmark languages, such as the eXtensible Configuration Checklist Description Format (XCCDF) and the Open Vulnerability and Assessment Language (OVAL), already provide capabilities for structuring guidance in a machine-understandable way and describing how to gather and evaluate system information to determine compliance. However, our experience has been that the expressiveness of these two languages does not yet match the expressiveness of much well-written security guidance, especially when this guidance covers large, complicated applications. Sometimes relevant information cannot be discovered programmatically, sometimes it can be discovered and gathered but a human is still necessary to determine policy compliance, and sometimes, while there may be some way to perform a compliance check programmatically, the existing languages simply do not have the necessary descriptive capability to encapsulate this check in a machine-readable way. As a result, security guidance can often not be completely wrapped using XCCDF and OVAL.

The Open Checklist Reporting Language (hereafter referred to as OCRL) addresses part of the gap between existing benchmark languages and the security guidance these languages attempt to encapsulate. In particular, it addresses cases where there is a known way to gather relevant information from a system programmatically, but the evaluation of this information to determine policy compliance either requires human involvement or is simply beyond the evaluation capabilities of existing benchmark languages. For example, many guides include a policy recommendation that states "The user should disable unnecessary services on the computer." While it may be possible for a software tool to discover what services are running on a given machine, the term "unnecessary" in the recommendation is one that has different interpretations depending on the needs of a given enterprise and the role of a particular computer within that enterprise. Determining whether a particular computer complies with this guidance requires someone to determine what services are necessary–a task that requires human judgment. On the other hand, a given user may not know how to determine what services a particular computer is offering or the procedure to do so may be time consuming. The user would be greatly aided if the information necessary to evaluate the recommendation, namely, a list of all the services running on a given computer, could be generated for them. The user could then simply look at this list and determine whether any unwanted services are present. OCRL provides this benefit to users.

OCRL, like XCCDF and OVAL, is an XML-based language defined by an XML schema file. An interpreter can read files written in this language and generate reports based on their contents. Authors can write files according to the syntax defined in this schema and expect

an interpreter will be able to read them. In this regard, OCRL can be viewed as an interpreted programming language in which authors describe procedures for generating reports and an interpreter uses the procedures to query a given computer for the specified information and then organize it into a report.

This document serves as a technical guide to OCRL. It covers all aspects of the language as well as how this language is used by an interpreter tool to construct reports. It is best used in conjunction with "Open Checklist Reporting Language (OCRL) by Example", which provides a number of examples of the use of OCRL.

Section 2

Using the Open Checklist Reporting Language

OCRL defines a language for creating reports for individual security recommendations. Documents created using this language can be used alone by an interpreter to create one or more reports; however, OCRL was also designed to fit within the larger structure created by existing benchmark languages, specifically XCCDF and OVAL.

XCCDF is an XML language designed to encapsulate the structure and organization of guidance documents. An XCCDF document consists of Rules, each of which corresponds to a recommendation in a piece of guidance. The other structures in XCCDF (Groups, Profiles, and Values) organize and refine Rules. In addition to supporting the structuring of guidance, XCCDF Rules also contain a "check-structure." The check-structure supports some form of automated processing of that Rule by referencing or encapsulating some set of processing procedures. An XCCDF interpreter is expected to read a check-structure and then call out to some external tool that is given the referenced procedures, performs the automation, and returns some value to the XCCDF interpreter corresponding to the result of the automation. XCCDF assumes that these automation procedures are automated checks that return Pass if the recommendation has been followed and Fail if it has not, although other return values are possible as well. In this way, an XCCDF document serves not only as a source for document generation (using XML style-sheets or other tools) but can, in theory, control a checking tool in evaluating a computer against some piece of security guidance. For the latter case, however, it is necessary to have some other language that XCCDF check-structures can reference that can describe how to evaluate compliance with recommendations in an automated way.

The OVAL language is an XML language that encapsulates procedures to gather and evaluate system information. It serves as an excellent complement to XCCDF in that OVAL can describe the checking procedures in an XCCDF Rule's check-structure. OVAL is structured in multiple hierarchical parts with the two most basic parts being OVAL Objects and OVAL States. An OVAL Object describes the location of some piece of information. For example, an OVAL Object might identify a Windows registry key, the metadata of a specific file, a particular line of a text file, or a query to an SQL server. An OVAL State describes how to evaluate the information gathered from an OVAL Object. The other structures in the OVAL language (Definitions, Tests, and Variables) are used to combine and tune Objects and States to represent sophisticated queries regarding system state. An OVAL Object and an OVAL State are used together by an interpreter to gather information (guided by the OVAL Object) and evaluate it against certain criteria. From the perspective of writing automatable security guidance, OVAL is used to describe automated compliance checks that are referenced by an XCCDF Rule's check-structure. Once the OVAL has been evaluated, a

return value is passed back to the XCCDF interpreter to indicate whether a Rule's recommendation has been followed.

OCRL fills a role similar to that of OVAL within the context of benchmark automation. As noted earlier, there are many cases of guidance that cannot adequately be encapsulated by OVAL. This was recognized early on by the developers of XCCDF, so they ensured that the check-structures in XCCDF Rules were not specifically tied to OVAL but could reference other mechanisms as well. To understand how this is accomplished, consider the following example of a check-structure from an XCCDF Rule:

```
<check system="http://oval.mitre.org/XMLSchema/oval-definitions-5">
   <check-content-ref href="SCAP-WinXPPro-OVAL.xml"
   name="oval:gov.nist.1:def:29" />
</check>
```

In this example from NIST Special Publication 800-68, a check-structure in an XCCDF Rule refers to an OVAL definition. The check-structure includes the "system" attribute, the "href" attribute, and the "name" attribute. The "system" attribute identifies the language in which the check is written, in this case OVAL version 5. The "href" attribute identifies the file in which the OVAL definition of interest is located as SCAP-WinXPPro-OVAL.xml. The "name" attribute identifies the oval:gov.nist.1:def:29 OVAL definition whose evaluation determines compliance with the associated recommendation. Note that this is not the only way in which an XCCDF check-structure can be written, but it is a good example for our purposes.

To have an XCCDF Rule utilize OCRL, we simply need to construct a check-structure with the appropriate values. The "href" attribute names a file in which a report definition appears. The "name" attribute holds the identifier of a specific report definition within that file. Finally, the "system" attribute is http://mitre.org/ReportSchema/v1 to indicate that the "check" is written using version 1 of OCRL. A check-structure that references a report might appear as follows:

```
<check system="http://mitre.org/ReportSchema/v1">
   <check-content-ref href="ReportDefinitionFile.xml" name="reportId1" />
</check>
```

This check-structure tells the XCCDF interpreter to invoke the OCRL interpreter to read the file ReportDefinitionFile.xml and process the report definition whose "id" attribute (in the report definition file) has a value of "reportId1".

It was noted earlier that XCCDF expects a check-structure to call out to some other language's interpreter and that this interpreter then returns some result. With OVAL, this is usually an indication of whether the recommendation associated with this check-structure's Rule was successfully followed. Likewise, an interpreter for OCRL needs to return some

XCCDF result value for each of the check-structures it is asked to process. In most cases, the XCCDF result is "informational" indicating that a report was generated but that the user has yet to decide as to compliance.

In this way, OCRL can be used alongside the OVAL language in XCCDF documents to provide added capabilities in benchmarks. The use of both languages allows more automation to be called out from an XCCDF document than could be done using OVAL alone. The result is a significantly enhanced capability for benchmark automation.

Intentionally Blank

Section 3

High-level Report Structure

This section covers the overall structure of a document written using OCRL. Subsequent sections describe some of the more significant components of this structure in more detail.

The root element of an OCRL document is <ReportDefinitions>. This element simply serves as a wrapper for the other content in an OCRL document. It has no attributes and may contain only one or more <ReportDefinition> elements.

The <ReportDefinition> element corresponds to a single report definition and results in one generated report. An OCRL document may define any number of report definitions each corresponding to one <ReportDefinition> element, all located within the main <ReportDefinitions> tag.

Each report definition contains the information necessary to generate a complete report. The <ReportDefinition> tag itself contains a number of attributes necessary for report generation. These are:

- *id* – The mandatory "id" attribute gives a unique name to a particular report definition. No two <ReportDefinition> elements within a file may have the same value for their "id" attribute (although it is not necessary for the "id" attribute to be unique across all documents written using OCRL). The "id" attribute labels a report definition to enable unambiguous reference to that report definition by an external agent such as an XCCDF interpreter. The name attribute of a check-structure in an XCCDF Rule uses the value of the report definition's "id" attribute to specify to the OCRL interpreter which report definition to process.

- *filename* – The optional "filename" attribute identifies the base name of the file to which an off-line report should be written. The full name of the report file consists of a date and timestamp prepended to the file name given in this attribute. If this attribute is absent, the base is computed as a combination of the name of the file where the report is defined and the report's unique "id" attribute. For example, a report generated from a report definition file named XP_reports.xml, with "id" attribute of 9-8 and no "filename" attribute, would be written to a file named *date_and_timestamp*_XP_reports9-8.txt. (Note that the format of an output report, and therefore its file extension, may vary among interpreters. Other OCRL interpreters might produce HTML or PDF files.)

- *recommendation* – The mandatory "recommendation" attribute contains the recommendation for which a particular report is being generated. For example, a recommendation might read "Remove all unnecessary services running on this

computer". The contents of the "recommendation" attribute appear in the output report for this report definition.

- *instruction* – The mandatory "instruction" attribute contains instructions to the reader of a report as to how to use the contents of the report to determine whether or not a particular recommendation has been followed. For example, for the above recommendation, a report's "instruction" attribute might read, "Inspect the services listed in this report and make sure that all are necessary for this computer to meet mission needs". The contents of the "instruction" attribute appear in the output report for this report definition.
- *title* – The mandatory "title" attribute contains a title for this report. The title appears in the output report for this report definition.
- *display* – The optional "display" attribute suggests to an interpreter whether the report should be presented directly to the user upon interpretation (value of "online") or should be written directly to a file for later inspection (value of "offline"). The "display" attribute only accepts values of "online" or "offline", the latter being the default value. Report definition authors can use this to suggest a way to display the report, usually based on the expected size of the generated report. If a report is likely to be extremely large, such as a list of all users in an enterprise, then an offline report is more likely to be useful to a user as the user probably does not wish to run through such a large list immediately. Note that the display method in this attribute is only a suggestion and an interpreter tool may choose to ignore it in favor of some other display method.

In addition to the above attributes, the body of a <ReportDefinition> element must contain either one or two child elements: a <DataSources> element and a <Reporting> element.

The first child of the <ReportDefinition> element must be one <DataSources> element. The child elements of the <DataSources> element contain instructions that tell the interpreter tool what system information to gather for a report. The <DataSources> block may contain instructions to gather information from multiple distinct sources, allowing the data from these different locations to be merged in the final report. The <DataSources> element and its contents are described in detail in Section 4 of this document.

The <DataSources> element must be followed by one <Reporting> element. The <Reporting> element contains instructions regarding how the data gathered in the <DataSources> segment of the report should be organized for presentation. In particular, the <Reporting> section enables the report definition writer to select specific items from specific DataSources allowing for hierarchical presentation of data and cross-correlation of otherwise separate data sets. The <Reporting> element and its contents are described in detail in Section 5 of this document.

In summary, a document written using OCRL consists of a single <ReportDefinitions> block containing one or more <ReportDefinition> elements. Each <ReportDefinition> element describes one report, including high-level descriptive information (title, instructions, recommendation, display) as well as how to gather the report's data, how to structure this data in the final report. Each report definition in a file has its own unique identifier that allows a specific report to be selected by some external entity, such as an XCCDF document. The following sections go into more detail about how a report definition identifies data sources and formats reports.

Intentionally Blank

Section 4

DataSources

This section describes in detail how OCRL structures its <DataSources> block and how the contents of this block are used by an interpreter to gather information for report generation. The <DataSources> element holds one or more data source definitions. Data sources are defined using extensions of the abstract <DataSource> element. Each type of data repository that is supported as a data source in OCRL has its own extension of the <DataSource> element. For example, the <WmiDataSource> element defines a data source that extracts information from a WMI query, while the <FileDataSource> element defines a data source that extracts information from a file. The only feature common to every extension of the abstract <DataSource> element is the presence of the mandatory "id" attribute. This attribute's value provides a name that can be referenced within the <Reporting> section. The ids of all <DataSource> extensions within a <DataSources> block must be unique to avoid ambiguity in references.

The information identified in the <DataSources> block can come from a wide range of potential sources including text files, WMI, OVAL queries, and the execution of command-line applications. Because each of these sources is structured differently, each source has its own extension of the abstract <DataSource> element and each extension has its own specifications for gathering the relevant information. Despite differences among the various <DataSource> extensions, the information extracted from all types of sources is structured by an interpreter tool using the same data model. This allows the <Reporting> section to use the information from all types of sources by referring to the same data model structure regardless of the type of repository from which the data was gathered.

4.1 The Open Checklist Reporting Language Data Model

The information collected in the <DataSources> block is organized using the same data model regardless of the type of repository from which it is extracted. The <Reporting> section relies on this common data model when constructing a report. This model can be viewed as having three levels of abstraction.

The simplest structure in the OCRL data model is called a "Property". A Property represents a name-value pair. The value of a Property is extracted by the OCRL interpreter when it reads data from a source identified in a data source's definition. The name of a Property provides a handle for this value and is used within the <Reporting> section to identify which Property's value should be processed at a given point.

The next structure in the OCRL data model is called an "Instance". An Instance is a collection of Properties. Properties that share an Instance are connected in some way. This

connection may be in scope, in subject, in time, or some other aspect. Thus, Instances preserve the context of Properties within a data source. For example, consider a medical device monitoring a patient. Every minute, the machine records the time and the patient's temperature. In the OCRL data model, these would be represented by two Properties, possibly with the names of "time" and "temp". Assume we have an hour's worth of material from this device. If the information we extract is going to be useful to us, each "temp" Property must be associated with a "time" Property since, otherwise, we end up with sixty "temp" Properties without any context telling us their order. OCRL uses the Instance structure to record this correlation between Properties. Specifically, the "time" and "temp" Properties at each monitoring interval are stored into one Instance, giving us sixty Instances over an hour's worth of monitoring. When we examine the data we recorded, we know the "time" Property corresponding to each "temp" Property because they share an Instance.

Finally, the highest level structure in the OCRL data model is called a DataSource. Each DataSource corresponds directly to one <DataSource> extension within the <DataSources> block. A DataSource holds one or more Instances. In the above example, the DataSource contains sixty Instances, corresponding to the entire hour of collected data. Since the <DataSources> block in a report definition may hold multiple <DataSource> extensions, a single report definition may utilize many DataSource structures.

It is important to note that the OCRL data model does not necessarily correspond to any physical representation. That is to say, the OCRL interpreter does not necessarily create any file or database with actual components corresponding to DataSources, Instances, and Properties. Instead, the data model is simply a way to visualize the data collected in the <DataSources> block. The subsequent <Reporting> block uses this organization of the data when describing how to construct a report. The data model we have just described is used throughout our description of OCRL. Throughout this document, capitalized "DataSource", "Instance", and "Property" refer to structures within the OCRL data model.

4.2 The <WmiDataSource>

The first of the defined extensions of the abstract <DataSource> element is the <WmiDataSource> element. It is used to build a DataSource from a single WMI query. WMI (Windows Management Instrumentation) is very similar to SQL in both syntax and function, but instead of a query being sent to an SQL server the query is sent to a WMI provider which processes the query and returns a result consisting of sets of named properties.

In order to explain how we construct a WMI query, it is necessary to understand something about WMI itself. WMI can be thought of as being divided up into a set of "namespaces". Within a namespace is a set of "classes". Each class defines a template for some set of information. The template specifies a set of WMI properties for containing information. The information returned by a WMI query is in the form of WMI instances. WMI instances are structured according to the template defined by a given class. Each WMI

instance contains named properties and their values. (Please do not be confused by the terminology here. Capitalized Instances and Properties refer to structures in the OCRL data model. Lower case instances and properties refer to WMI components. There is a good reason that similar terms are used here, but it is important that these elements not be confused.)

The first child element of a <WmiDataSource> element must be the <Namespace> element. The body of this element is the WMI namespace to which the query should be directed.

The query itself may be specified in one of two ways. The simplest way is through the <Query> child element. The body of this element is a WMI query exactly as it should be passed to the WMI provider for evaluation. This method is simplest if a user is familiar with the WMI query language. In this method the <Query> element must be the only child element of <WmiDataSource> after the <Namespace> element.

Alternatively, the document author may specify the query information piecemeal through the <Class>, <Property>, and <Where> elements. Each of these elements represents some piece of a full WMI query. This method may prove easier for those not familiar with the syntax of WMI queries. In this method the second child of <WmiDataSource> (after <Namespace>) must be one <Class> element. The body of the <Class> element must contain the name of some class within the specified WMI namespace. It is instances of this WMI class that are to be returned by the query.

The <Class> element may be followed by zero or more instances of the <Property> element. The body of each instance of the <Property> element holds the name of one WMI property in the specified WMI class. If this element is present, only the properties named in <Property> elements are included in the returned WMI instances. Since any number of <Property> elements may be present, any number of properties may be requested. If no <Property> elements are present, the query asks for all properties of the named class.

Finally, the user may include one <Where> element. Like SQL, a WMI query may also include a WHERE clause that filters out potential return instances based on a set of criteria. The body of the <Where> element should be the body of a WMI WHERE clause and is appended to the query sent to the WMI provider. The returned WMI instances are filtered based on the WHERE clause.

The results of a WMI query fall quite directly into the OCRL's data model. In particular, WMI instances map directly to the Instance structure in OCRL while WMI properties map directly to Properties. This means every WMI instance returned by the WMI query becomes an Instance in the DataSource created for this <WmiDataSource>. Likewise, every WMI property in each of the returned WMI instances becomes a Property in the corresponding Instance. The name of the WMI property becomes the Property name while the value of the WMI property becomes the Property value.

4.3 The <FileDataSource>

The <FileDataSource> element is used to create a data source from a text-based file, such as a text or XML file. Because different files may be structured differently, this <DataSource> extension not only needs to identify the data source itself (i.e. locate the file) but also explain how to read this file and map its data into Instances and Properties.

The <FileDataSource> begins with one <Filename> element. The body of this element contains the name of the file that contains the source data. The <Filename> element has an optional "directory" attribute. If the "directory" attribute is present and is an absolute path, then the OCRL interpreter attempts to locate the file in the named directory. If the "directory" attribute is a relative path or if it is absent, the OCRL interpreter attempts to locate the file relative to the interpreter's current working directory.

An <InstanceStructures> element must follow the <Filename> element. The <InstanceStructures> element contains the instructions to the OCRL interpreter of how to read the contents of the named source file and map the relevant information in this file into the OCRL data model.

There can be two parts to the information contained with an <InstanceStructures> element: an optional “definitions” part and a required “declarations” part. The “definitions” contains instance definitions that can be referenced within the “declarations” part. It is the “declarations” part that specifies what will actually be read from the named source file and mapped into the OCRL data model.

The first child of the <InstanceStructures> element can be the optional <InstanceDefinitions> element. This <InstanceDefinitions> element, if present, must contain one or more <Instance> elements. The <InstanceStructures> element must contain an <InstanceDeclarations> element, following the <InstanceDefinitions> element if it is present. The <InstanceDeclarations> element must contain one or more <Instance> or <InstanceReference> elements. The <Instance> and <InstanceReference> elements describe how to identify relevant data within the given file and how to map this data into the OCRL's data model. These elements are described in detail below.

4.3.1 The <Instance> Element

As noted earlier, different files structure their content differently. Because of this, the <FileDataSource> must describe how to identify the relevant structures of a file so that Instances and Properties can be correctly extracted. The <Instance> element and its child elements accomplish this.

As the name suggests, the <Instance> element is used to identify the structures within the target file that map to Instances within the OCRL's data model. This document calls these structures in the target file "Instance-structures". An <Instance> element, along with its

children, identifies these by defining a template, called an "Instance-class", which describes how to recognize Instance-structures within a file and how to build an Instance and its Properties within the OCRL data model given these Instance-structures. Please note the terminology: an Instance represents a structure within the OCRL data model, an <Instance> element is an XML structure in OCRL, an Instance-class is the template that an <Instance> element defines that identifies how to find pieces of a source file that map to Instances and Properties, and Instance-structures are the actual pieces of a data file that match the patterns in an Instance-class and so can be mapped to an Instance in the data model. In a typical case, a Report Interpreter reads the <Instance> element in a report definition and from this it builds an understanding of the Instance-class that this element defines. Each <Instance> element defines one Instance-class. When the interpreter reads through a data source file, it uses this Instance-class to try to locate Instance-structures within the file. A single data source file may contain multiple Instance-structures that match an Instance-class. That is, there may be multiple places in the source file where the patterns defined in the Instance-class are satisfied. When it finds an Instance-structure within a data source file, the interpreter uses the instructions in the Instance-class to map the information contained within the Instance-structure into an Instance in the OCRL data model. Every Instance-structure located within a file becomes one Instance in our data model.

OCRL assumes that the boundaries of Instance-structures can be identified using regular expressions. For example, in an Apache configuration file, the configuration directives for an individual virtual host are bracketed by <VirtualHost> and </VirtualHost> lines. By defining an Instance-class that identifies these patterns, an interpreter can record that any Properties extracted from data between these two lines belong in the same Instance and are thus linked to each other in some way. (In this case, they are linked because they share the same scope.) <Instance> elements can be nested, allowing for a hierarchy of Instances within a DataSource. For example, in an Apache configuration file, <File> and </File> lines bound directives that should be applied to a single file. The <File> block can appear between <Directory> and </Directory> lines, which bound directives that apply to all files in a particular directory, and both <File> and <Directory> blocks can appear within the bounds of a <VirtualHost> block. All three of these group configuration directives by scope and therefore are all appropriate for mapping to Instances. However, like Properties, the contexts in which these sub-structures appear are also significant. (For example, it is important to note that certain directives apply to a given file only when accessed through a given virtual server. Thus it is as important to note that the <File> block is within a given <VirtualHost> block as it is to note that a given directive is within the <File> block itself.) By nesting <Instance> elements, we can capture these relationships and record that certain sub-structures exist within the context of larger structures.

<Instance> elements that are children of the <InstanceDeclarations> element as well as <Instance> elements that appear within other <Instance> elements serve two purposes. First, they define an Instance-class. Specifically, they define how to recognize the bounds of an

Instance-structure within a file and they describe the Properties that may be found within that Instance-structure. Secondly, <Instance> elements can also declare the portions of a source file that may contain the Instance-structures corresponding to the given Instance-class. Specifically, the placement of the <Instance> element within the hierarchy of other <Instance> elements in the <FileDataSource> tells the OCRL interpreter where it should be looking for a specific Instance-structure. For example, if an <Instance> element is defined within a parent <Instance> element, then the interpreter only searches for the patterns defined in the inner Instance-class when it is parsing within Instance-structures that correspond to the parent Instance-class. This means that if the interpreter finds a pattern that marks the beginning of the inner Instance-class but this pattern is located outside the body of a parent Instance-structure, then the child Instance-class's context is not satisfied.

<Instance> elements that are children of the <InstanceDefinitions> element serve only the first of these two purposes, defining an Instance-class but not specifying the context in which the associated Instance-structures might be found within a document. By placing an <Instance> element in the <InstanceDefinitions> element, the associated Instance-class may be referenced in an <InstanceReference> element.

<InstanceReference> elements only serve the second of the above purposes; namely, it specifies a context for the referenced Instance-class. Because multiple <InstanceReference> elements may reference the same Instance-class, this allows the author to indicate that a single Instance-class may appear in multiple contexts. For example, the Apache <File> block can appear at the top-level of a file, within a <VirtualHost> block, or within a <Directory> block. By placing the <Instance> element defining a <File> block's Instance-class in the <InstanceDefinitions> element, the author can then use <InstanceReference> elements to indicate the three contexts (top-level, within a <VitualHost>'s Instance-class, and within a <Directory>'s Instance-class) in which a <File>'s Instance-class might be found.

4.3.1.1 The Structure of <Instance> Elements

All <Instance> elements must contain the "classid" attribute. This attribute provides a unique name, for each Instance-class. (Recall that each <Instance> element defines an Instance-class.) The “classid” attributes of all <Instance> elements within a <FileDataSource> must have different values. The "classid" attribute serves two purposes. First, it supports the use of the <InstanceReference> element as it is the classid that the <InstanceReference> uses to identify the Instance-class to which it refers. Secondly, every Instance created using the named Instance-class contains a Property whose name is ClassId and whose value is the value of the <Instance> element's "classid" attribute. This Property is useful within the <Reporting> block to identify the Instances in a DataSource that were defined using a given Instance-class.

The first child element of an <Instance> element is the optional <Begin> element. The body of this element is a regular expression. This expression describes the pattern that marks

the beginning of Instance-structures for a given Instance-class. If the <Begin> element is absent and the Instance-class declaration is not within any other <Instance> elements, then the OCRL interpreter assumes the start of the file is the starting boundary of the associated Instance-structure and begins parsing for contents immediately. (As a side effect, if the <Begin> element is absent, then the file can only produce a single Instance for the DataSource since the start of the file only occurs once.) If an Instance-class has no <Begin> pattern and it is declared within another <Instance> element, then the child Instance-structures starts at the beginning of the parent's Instance-structures. The <Begin> element has an optional attribute named "caseInsensitive" that, if given a value of "true", instructs the regular expression matcher to do a case-insensitive match. Otherwise, the regular expression pattern is case-sensitive.

Next, the <Instance> element may contain the optional <End> element. Like the <Begin> element, the body of the <End> element is a regular expression and it has an optional "caseInsensitive" attribute. The regular expression in the <End> element denotes a pattern that marks the end of an Instance-structure. When a line that matches the pattern is encountered within a file, the OCRL interpreter closes out the created Instance and does not add any more content to it. It then continues parsing the file. (If the Instance that was closed is a child Instance-class of a parent Instance-class, then the interpreter continues to parse according to the parent's Instance-class until the parent's <End> pattern is satisfied.) If the <End> element is absent, the Instance automatically closes at the earliest of the following circumstances (in the following order if there is a tie):

1) The end of the file is reached.
2) A pattern is discovered that matches the <Begin> pattern of this <Instance>. In this case, the current Instance is closed and a new Instance of this Instance-class class is started.
3) A pattern is discovered that matches the <Begin> pattern or <End> pattern of this <Instance> element's parent element. In this case both the current Instance and the parent Instance are closed. If it was a <Begin> pattern that was located, a new Instance of the parent's Instance-class is started.
4) A pattern is discovered that matches the <Begin> pattern or <End> pattern of this <Instance> element's grandparent element. In this case the current Instance, the parent Instance, and the grandparent Instance are all closed. If it was a <Begin> pattern that was located, a new Instance of the grandparent's Instance-class is started.
5) Etc. through successive ancestors.

Note that if an Instance-class contains child Instance-classes, finding a line that matches the <Begin> pattern of the child Instance-class does not close the processing of the Instance associated with the parent Instance-class. Instead, it simply defers adding more information

to the parent Instance until the child Instance has closed. Once the child Instance-class's closing pattern is found, the interpreter continues populating the parent Instance as normal. Note also that if a parent and child share a closing pattern and this pattern is encountered while processing the child, only the child Instance closes. The reasoning for this is that the closing pattern was found in the context of the child Instance and therefore only applies there.

Note also that Instance-structures that are at the same level (that is, they are children of the same parent Instance-class or have no parent Instance-class) may overlap. For example, consider the following configuration file:

```
Users=4
### A ###
Language=English
Option=Complete
### B ###
File=Config.xml
Nested=True
```

Suppose we define two Instance-classes: The first Instance-class, called class-A, has a <Begin> pattern of "### A ###". The second Instance-class, class-B, has a <Begin> pattern of "### B ###". Neither Instance-class has an <End> pattern. This means that the Instance-structure corresponding to class-A starts with the second line of the file and goes to the end, while the Instance-structure corresponding to class-B starts with the fifth line of the file and goes to the end. This means that lines 5-7 are part of the Instance-structures of both Instance-classes.

Following the <Begin> and <End> elements, the body of an <Instance> element may contain any number of <Instance>, <InstanceReference>, and <Property> elements in any order. There is no significance to the order in which these child elements appear. Both child <Instance> and <InstanceReference> elements denote the possible inclusion of some child Instance within Instances defined by the parent Instance-class. The <Property> element describes how to locate data that maps to a Property within an Instance defined by the parent Instance-class. Note that just because a particular set of <Instance> and <Property> elements appear within the body of a given Instance-class, this doesn't mean that every Instance created using the parent Instance-class must contain child Instances and Properties for all these elements. It is, in fact, perfectly acceptable for no child Properties or Instances to be discovered before the parent's <End> pattern is reached, in which case the given parent Instance has only the ClassId Property and no other Properties or child Instances. An <Instance> element's child <Property> and <Instance> elements merely indicate what might appear within an Instance corresponding to that Instance-class rather than what must appear.

4.3.1.2 <Property> Elements

Just as <Instance> elements tell the OCRL interpreter how to identify file structures that represent Instances within a DataSource, the <Property> element tells the interpreter how to identify Properties within an Instance. Because all Properties must be stored within an Instance, <Property> elements must always appear as children of <Instance> elements.

The <Property> element has a mandatory "name" attribute. This specifies the name part of the name-value pair that represents a Property within the OCRL data model. All <Property> elements within a given <Instance> definition must have a different value for their "name" attribute. It is not necessary, however, for the "name" attributes of <Property> elements to be unique across different <Instance> definitions.

There are two possible child elements within a <Property> element, both of which tell the OCRL interpreter how to locate the data in the source file that maps to a Property. The first of these is the <Line> element. The body of the <Line> element is an integer. The value of this Property is set to the text of the line in the source file that is this many lines after the start of an Instance-structure matching the parent Instance class. For example, if a given <Property> contained <Line>3</Line>, then the entire contents of the third line after the line that starts the parent Instance-structure becomes the value of this Property. A line-number of 0 indicates the line that starts the parent Instance-structure. (That is, the same line that matches the <Begin> pattern of the parent Instance-class.) It is no error for the parent Instance to close before reaching the line-number specified in one of the child <Property> elements – it simply means that this particular Property is not present in the parent Instance. The <Line> element is useful for extracting information items from structured text that always have the same ordering within a file.

The second way to identify a Property within an Instance-structure is to use the <PatternMatch> element. The body of the <PatternMatch> element is a regular expression. The <PatternMatch> element has an optional "caseInsensitive" attribute that, if set to "true" tells the pattern matcher in the interpreter to perform a case-insensitive match. If the regular expression matches a given line within an Instance-structure, then that line is used to provide the value to this Property. If the regular expression contains a group (parentheses within the regular expression) then the value captured in the first group becomes the value of the Property. Otherwise, the value of the entire matching line becomes the value of the Property.

It is possible to use both <Line> and <PatternMatch> elements within a single <Property> element. In this case, both the line-number and the regular expression must match for some line of the source file in order for the Property to be created. This sort of structure is especially useful when the value that one wishes to associate with a Property is a substring of a given line (and therefore we need to use a regular expression with a group to extract the substring), but the line itself does not have unique components that can readily

distinguish it from other lines within an Instance-structure (in which case the line-number is the only way to identify the desired line).

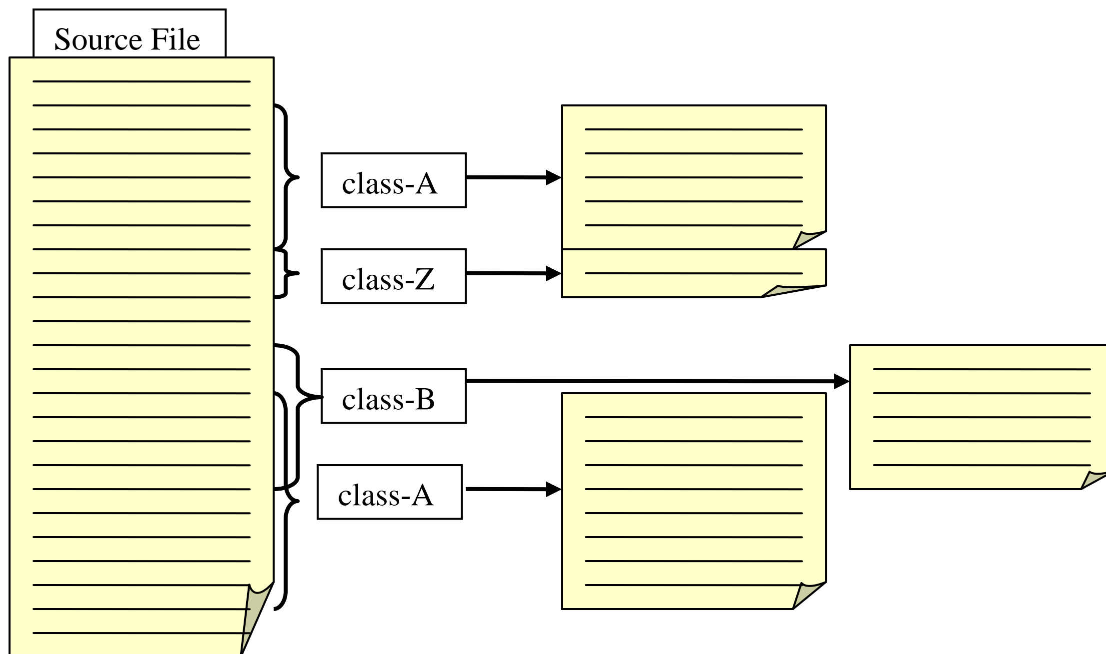
4.3.2 The <InstanceReference> Element

The <InstanceReference> element has been discussed before on a number of occasions. It is used to declare that an Instance-class can be extracted within the context in which the <InstanceReference> resides. For example, if the <InstanceReference> appears within a parent <Instance> element, it indicates that the referenced Instance-class can be found within the Instance-structure of that Instance-class. It is especially useful when the same Instance-class can be extracted from multiple contexts.

An <InstanceReference> element has one mandatory attribute named "targetClassId". The value of this attribute is the value of the referenced <Instance> element's "classid" attribute. The named <Instance> element must be present within the <InstanceDefinitions> element and it must be a child of <InstanceDefinitions>. (That is, it cannot exist as a child of another <Instance> element.) An <InstanceReference> element can appear within any <Instance> element (including <Instance> elements that appear in the <InstanceDefinitions> block and anywhere within the <InstanceDeclarations> block.

4.3.3 Visualizing Parsing Structures

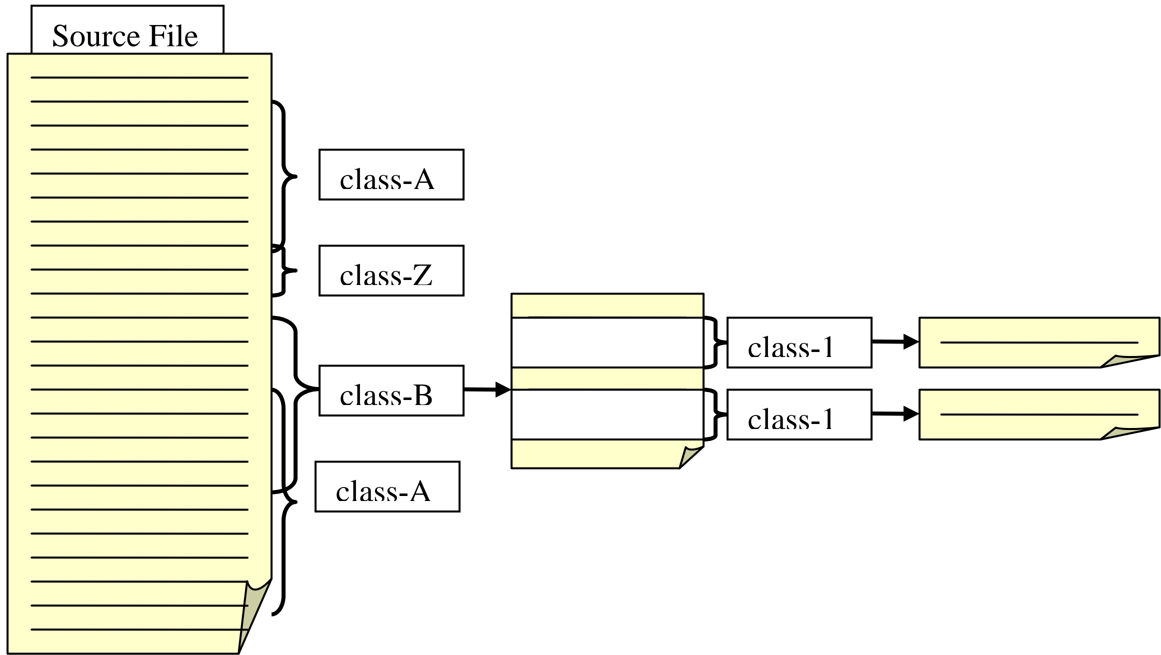
The power and flexibility of the parsing structures outlined above can also make them confusing to work with for novice authors. One way to visualize these structures that might prove helpful is to view Instance-classes as operators that break a file up into blocks. Consider the case where the <InstanceDeclarations> element contains two <Instance> elements and one <InstanceReference> element. The two <Instance> elements define and declare two Instance-classes named class-A and class-B. The <InstanceReference> element references a class named class-Z, whose <Instance> element appears in the <InstanceDefinitions> element. Given a file, we might have the following Instance-structures:



The above source file contains four Instance-structures, which means the DataSource ends up with four Instances (one for each Instance-structure). There are a couple items to note: First, note that class-A has two Instance-structures in the file. This indicates that its <Begin> pattern was discovered twice. Secondly, notice that the Instance-structure for class-B overlaps the second Instance-structure for class-A. This simply means that the <Begin> pattern for class-A was discovered in the file between the <Begin> and <End> patterns of class-B.

In the above diagram we have physically broken out each of the Instance-structures into their own sub-files. While the OCRL interpreter does not necessarily do this, it is a useful way to visualize how an interpreter handles <Instance> elements. The interpreter only looks for the patterns of <Property> elements that appeared in class-A's <Instance> element when parsing the sub-files associated with class-A. In other words, the <Instance> element in which a <Property> element appears limits the context in which the interpreter looks for that <Property> to the Instance-structures (represented above by sub-files) associated with the parent <Instance> element's class. Note that if the <Instance> elements for class-A and class-B both had a <Property> element that looked for the same pattern, if a line of the source file that matches the pattern appears in the overlapping segment of class-B and class-A, then that one line could result in one Property being created in each of the associated Instances.

Continuing this example, assume that the <Instance> element associated with class-B also contains an <Instance> element defining a class called class-1. In our diagram above, this might be represented as follows:



As with the case of child <Property> elements, if an <Instance> appears within the body of another <Instance> element, the OCRL interpreter only searches for the associated Instance-structures in the Instance-structure associated with the parent. Put another way, the child Instance-structures are only extracted from within the sub-file associated with the parent Instance-class, even if the <Begin> pattern could be satisfied elsewhere. As before, any Properties extracted for the child Instance-classes must come from lines within the child's Instance-structure (sub-file in the above diagram).

Earlier we saw how sibling Instance-structures, that is, Instance-structures that share the same parent Instance-class or that have no parent Instance-class, can both add a Property from the same line of a source file. This is not true between parents and children. A line of the source file can only belong to either the parent or the child but not both. This is why, in the above diagram, there are gaps in the parent sub-file where the child Instance-structures appear. No <Property> patterns from the parent Instance-class can match any line within the Instance-structures of children. (The lines of the child Instance-structures are still counted for the purpose of computing the <Line> pattern of a <Property>; but if the specified line number appears within text that corresponds to a child Instance-structure, then this is treated as a failed pattern and the Property is not added to the parent Instance.)

One final complexity should be addressed here. Consider the case where class-B contains an <InstanceReference> that references class-Z. In this case, if the Instance-structure for class-Z existed within the Instance-structure (sub-file) of class-B, then this Instance-structure of class-Z results in the creation of two Instances, one at the very top level of the Instance tree and one as a child of an Instance of class-B. This is because class-Z can appear both at the top-level context (that is, it has no parent <Instance>) and within the context of class-B. As a result, the same block of text is matched within both contexts and the result is two Instances containing the same information at different locations of the Instance tree.

4.4 The <ExecutableDataSource>

The <ExecutableDataSource> element extracts information gathered by running a console application and parsing its output. It specifies which executable to run, the arguments that should be passed to this executable, and how to parse the resulting output into the Instance and Property structures of the OCRL data model.

The first child element of the <ExecutableDataSource> element is the <Executable> element. The body of this element is the name of the executable that the OCRL interpreter executes. The <Executable> element contains an optional "directory" attribute. If this attribute is present, the OCRL interpreter looks for the named executable in the given directory (either absolute or relative to the interpreter's current working directory). Otherwise, the interpreter looks for the executable in the user's regular executable path.

Following the <Executable> element, the <ExecutableDataSource> element may contain an <Arguments> element. The body of this element holds the arguments that are provided to the executable when it is run by the interpreter. Of special note, the interpreter expects that the executable's results to be sent to the standard output stream. If this is not the case, the <Argument> element must include command-line instructions to redirect the application's results to standard output.

Following these elements, the <ExecutableDataSource> element must contain an <InstanceStructures> element. This element has the same structure and function as the corresponding element described under the <FileDataSource>. To understand how this element and its children are used in this data source, treat the console output from the executable as the contents of a file - it is this "file" that the OCRL interpreter parses.

4.5 The <OvalDataSource>

The OVAL language describes how to locate information on a computer and how to evaluate this information against some set of criteria. OVAL has been extensively developed and now contains structures to locate information in dozens of potential repositories across many operating systems. OCRL can take advantage of this existing framework to identify

and extract DataSources using OVAL Objects, which are the component of the OVAL language responsible for locating information.

To understand how OCRL uses an OVAL Object to populate the OCRL data model, it is helpful to understand the OVAL System Characteristics (SC) schema. The OVAL SC schema is used to store the information that OVAL gathers during its evaluations. Files created using the SC schema hold all gathered information that might be used for the evaluations in the associated OVAL State. As such, they represent a snapshot of some segment of the data repository identified by the OVAL Object.

OVAL SC files contain "Item" elements in the <system_data> element of the root <oval_system_characteristics> element. These Item elements are all extensions of the abstract <item> element defined in the core OVAL SC schema. Just as each type of repository defines its own Object and State extensions, each type of repository supported by OVAL also defines its own extension to the <item> element. These extensions consist of one XML element for each element in the corresponding OVAL State extension, plus a "status" attribute that indicates how successful the OVAL interpreter was at gathering the specified information. Note that a single OVAL Object may lead to multiple Item elements in an OVAL SC file if the Object identifies multiple locations on a system.

The OCRL interpreter creates one Instance for each Item element that would be produced in an OVAL SC file by the OVAL Object identified in an <OvalDataSource>. The XML elements of each of these Items become the Properties of the associated Instance, where the name of the Property is the name of the element and the value of the Property is the body of this element. In addition, a Property with the name "Status" is created whose value was the value of the Item's "status" attribute. Note that despite the strong tie to the structure of the OVAL SC file, an <OvalDataSource> does not necessarily result in the creation of an SC file. It is up to the implementer of the OCRL interpreter as to whether the interpreter uses an SC file as an intermediate source or simply gathers the requisite information through direct interpretation of the OVAL Object.

The <OvalDataSource> element must contain the <OvalObjectId> element as its first child. The body of this element is the identifier of the OVAL Object as indicated by the Object's "id" attribute. The <OvalObjectId> element has one mandatory attribute named "filename" which must contain the name of the file in which this OVAL Object is to be found. The filename can include the path of the file if appropriate.

In addition, the <OvalDataSource> element may also contain an optional <SystemCharacteristicsFilename> element. The body of this element is the name of an existing system characteristics file. This named file must contain OVAL Items produced as a result of evaluating the OVAL Object named in the <OvalObjectId> element. This is a convenient shortcut if an OVAL interpreter has already evaluated that relevant Object. Note that even if the <SystemCharacteristicsFilename> element is present, the <OvalObjectId>

element must still be present so the interpreter knows which OVAL Object's Items to extract from the SC file. The <SystemCharacteristicsFilename> element may have an optional attribute named "directory" that specifies the path to the SC file if that file is not in the current working directory.

Intentionally Blank

Section 5

Reporting

This section describes the <Reporting> block of OCRL and how the contents of this block structure the information defined in the DataSources block into a report. As noted earlier, a Report Interpreter gathers information from data sources and maps this information into the OCRL data model, consisting of DataSources, Instances, and Properties. The <Reporting> block uses this data model structure for building a report.

The <Reporting> block is best understood by viewing it as an interpreted programming language with loops and declarative statements. Certain elements within a <Reporting> block can be understood as loops, iterating over a DataSource's Instances and invoking some set of actions in each iteration. We start this section by looking at the main functional pieces of a <Reporting> section, namely <Text>, <Select>, and <Item> elements, and describe how they should be interpreted before explaining how they are combined within a <Reporting> block.

5.1 <Text> Elements

The <Text> element is the simplest of the functional pieces of the <Reporting> block. It corresponds to a "print" statement in most programming languages, adding the text that appears in the body of the element to the final report every time it is encountered. The <Text> element is used by report authors to provide descriptive information as well as a means to separate sets of data. For example, a <Text> element might be placed first in the body of a loop so that it can clearly denote the start of each loop-iteration in the output report. The body of each <Text> element appears on its own line in a report file. It appears in the report verbatim with the exception of html escape sequences (<, &, ", etc.) which are converted to their corresponding character in the report file (<, &, ", respectively). The exact appearance of the text in the report (as well as the format of the report itself) varies with each interpreter so authors should ensure text does not require any special formatting to be understandable.

5.2 <Select> Elements

The <Select> element is the structure in the <Reporting> block that corresponds to a programming language loop. A <Select> element iterates over a set of Instances within a named DataSource. Recall from the data model that a DataSource consists of one or more Instances. The body of a <Select> element consists of one or more child elements (<Text>, <Item>, and <Select>) and each of these elements is "invoked" in the order in which it appears for each iteration of the <Select> loop. The <Select> element identifies a

DataSource[1] and then, for each Instance in this DataSource, the OCRL interpreter invokes each of the <Select> element's child elements.

The <Select> element itself has two attributes: "datasourceid" and "instances". The "datasourceid" attribute identifies which DataSource should be iterated over by specifying the value of some DataSource's "id" attribute. The OCRL interpreter gathers all of the top-level Instances of that DataSource in the order in which they were created and for each of these Instances, the interpreter processes each child element in the <Select> element's body in the order in which they appear. Note that a <Select> element only iterates over the top-level Instances of a DataSource. If those Instances contain child Instances, then the children are not included in the list over which the <Select> element iterates.

Of course, sometimes we do not want to process every Instance in a given DataSource. One way to limit the Instances over which the <Select> element iterates is through the optional "instances" attribute. The "instances" attribute can be "all", "first", or "last". If set to "all", the <Select> element iterates over all the top-level Instances in the given DataSource. If set to "first" or "last", the <Select> element only takes the first or last Instance, respectively, in a DataSource's Instance list. Note that in the latter two cases, the <Select> element is only iterating over a single Instance and therefore the body of the <Select> is only applied once. By default, the <Select> element iterates over all top-level Instances of the named DataSource.

5.2.1 <Qualifier> Elements

If the user wishes to further restrict which Instances are processed in a <Select> element, the <Select> element may have a <Qualifier> or <Where> element as its first child. Both these elements have identical functionality (and are, in fact, exactly the same XML type). Without loss of generality, this document describes the resulting functionality using the <Qualifier> element.

The best way to understand a <Qualifier> is to think of it as a test at the head of a <Select> loop. The test is applied to each Instance over which the <Select> element is iterating. If the test passes, the body of the <Select> element is processed as usual. However, if the test fails, the given Instance is skipped and the <Select> skips to the next iteration using the next Instance in its list. In this way, authors can place constraints that filter a DataSource's set of Instances such that only a subset of them appear in the report.

---

[1] Actually, <Select> elements can iterate over the child Instances of another Instance as well as all the top-level Instances of a DataSource. The details of this behavior are discussed later in this document. For the moment, however, the reader will not be led far astray by simply associating each <Select> element directly with a DataSource.

A <Qualifier> element itself makes a simple comparison between two values. If the comparison is true, then the condition is met and the Instance is processed. This comparison is defined by an operator, an operator type, and two operands.

The operator of a comparison is specified by the mandatory "operator" attribute in the <Qualifier> element. The value of this attribute may be one of the following: "equal", "not equal", "greater than", "greater than or equal", "less than", or "less than or equal". The operator type indicates whether the operands should be treated as numeric or string values. This can change how the operator compares two values. For example, the values "12" and "0012" are numerically equal but their strings are different. The operator type is specified by the mandatory "operatortype" attribute in the <Qualifier> element, and its values must be either "string" or "number".

The operands of a <Qualifier> are specified in child elements of the <Qualifier> element. All <Qualifier> elements must have exactly two child elements to describe a valid comparison. There are three types of elements that may appear as operands of a <Qualifier>, each type of element corresponding to a different way in which the operand's value is derived. A <FixedValue> element's body holds some static text that should be used directly in the comparison (either as a string or converted to a number). A <Property> element's "name" attribute contains the name of a Property within the given Instance (as described in the section on DataSources) and the value of this Property in the current Instance is used in the comparison. (Recall that the body of a <Select> is applied to each Instance of a DataSource in turn, so the <Property> element of a <Qualifier> gathers the value of the named Property from the Instance corresponding to the current iteration of the <Select>, possibly leading to different results for each iteration.) Finally, a <Variable> element's "name" attribute contains the name of some Variable. The value of this Variable is then used in the <Qualifier> element's comparison. Variables are described more when we discuss <Item> elements.

Both the <Variable> and <Property> elements can themselves have a child element named <Substring>. The <Substring> element indicates that only some part of the value of the given Variable or Property should be used in the comparison, rather than the entire value. The body of the <Substring> element contains a regular expression. This regular expression should contain at least one group marker, represented by parenthesis in the regular expression. If the regular expression matches, the value captured in the first group becomes the value used in the comparison, rather than the full value of the associated Variable or Property. This allows greater control over the types of tests performed in the language. The <Substring> element has an optional "caseInsensitive" attribute that, if present and set to "true", instructs the regular expression engine to perform a case insensitive match. Note that if the pattern in the <Substring> fails to match the value of the associated variable or Property, then the check described in the <Qualifier> automatically fails.

One special use of <Qualifier> elements within <Select> elements is for the ability to filter Instances by the Instance's class-id. Recall from the section on DataSources that some types of DataSources that explicitly define their Instances (in particular FileDataSources and ExecutableDataSources) include a ClassId Property in each Instance. Recall also that, by default, when a <Select> element identifies a DataSource it iterates over every top-level Instance in that DataSource regardless of that Instance's type. On some occasions, however, one might wish only to report Instances created using a specific class of <Instance> element from a DataSource. By creating a <Qualifier> element that compares an Instance's ClassId Property against a static string equal to the "classid" attribute of the <Instance> definition of interest, the <Select> loop only processes Instances that were created using the named <Instance> element.

Note that, if both a <Qualifier> element and the "instances" attribute are used, the "instances" attribute comes into effect first, filtering the Instance of the DataSource before the <Select> block began iterating. As such, the <Qualifier> is used to further filter Instances that had already been filtered by the "instances" attribute.

5.2.1.1 Complex Qualifiers

As described above, a <Qualifier> element can be used to skip a given Instance based on a single condition. However, sometimes the decision to include a particular piece of information may hinge on multiple conditions. OCRL supports this as well through the use of complex qualifier statements. Specifically, the language defines three logical elements that can combine the results of <Qualifier> elements: <AND>, <OR>, and <NOT>. Both the <AND> and <OR> elements must hold two or more logical elements (<AND>, <OR>, or <NOT>) or <Qualifier> elements, in any order and combination. An <AND> element evaluates to true if all its child elements evaluate to true while the <OR> element evaluates to true if any of its child elements evaluate to true. The <NOT> element may only contain one logical or <Qualifier> element as its child and evaluates to true only if that child evaluates to false. Since logical elements can be nested to an arbitrary depth, virtually any combination of conditions can be used to restrict the Instances evaluated by a <Select> loop. (Note that only the <Qualifier> element and not the <Where> element can exist within a complex qualifier. The <Where> element can only be used alone.)

5.3 <Item> Elements

The final major component of the <Reporting> block is the <Item> element. An <Item> element identifies a named Property within a given Instance in a DataSource. Because Properties can only exist within Instances, the <Item> element can only appear within a <Select> element. An <Item> element adds a line in the output report that includes both the name of the Property as well as the Property's value. The exact way in which this information is presented varies among interpreter implementations.

All <Item> elements must include the "name" attribute. This attribute identifies the name of the Property whose value is to be extracted for inclusion in the report. As noted earlier, the list of Properties within a set of Instances may vary, even if these Instances have the same class-id. How an interpreter handles the case where an <Item> element names a Property that doesn't exist in a given Instance is left up to individual tool authors, but such an occurrence should not result in an error.

In addition to the "name" attribute, <Item> elements may have two more optional attributes. The first is the "setvariable" attribute. The value of the "setvariable" attribute is the name of a Variable whose value should be set to the value of the Property identified by this Item. All "setvariable" attributes within a given <Reporting> block must have distinct values to prevent duplication of Variable names. Once created, a Variable can be utilized in <Qualifier> elements through the <Variable> operand element. The benefits of Variables are explained in more detail when discussing how to combine <Text>, <Select>, and <Item> elements within the <Reporting> block.

A second optional attribute of the <Item> element is the "hidden" attribute. The value of this attribute may be either "true" or "false". If it is present and set to "true", the given <Item> element does not result in any output to the report. This attribute is typically used in conjunction with the "setvariable" attribute, allowing a Variable to be created and initialized without producing a line in the report.

Finally, like the <Select> element, <Item> elements may contain a child <Qualifier> element or even a complex qualifier. Note that, unlike the <Select>, which may use the synonymous <Where> element, only the <Qualifier> element is valid within an <Item>. A <Qualifier> element within an <Item> element has exactly the same structure and function as in the <Select> element. The only difference is that, if the <Qualifier> comparison fails, it simply means that the given <Item> does not get displayed in the report and/or does not set a Variable (if this <Item> has a "setvariable" attribute). Other siblings within the body of the encapsulating <Select> block are not affected when an <Item> element's <Qualifier> fails and are processed as normal. This is to say, the <Select> element does not jump to the next Instance in its DataSource when an <Item> element's qualifier fails. Skipping Instances only happens if the <Qualifier> is the immediate child of the <Select> element itself. Using the <Qualifier> element with an <Item> allows the OCRL document author to exclude the display of specific Properties based on certain conditions.

5.4 Putting It All Together

This section describes how the aforementioned <Text>, <Select>, and <Item> elements may be combined to form the body of a <Reporting> block. The <Reporting> block itself may only contain <Text> and <Select> child elements, although it may contain any number of these in any order. The child elements of the <Reporting> block are processed one-at-a-

time in the order in which they appear, with <Text> elements writing directly to the report and <Select> elements looping through the Instances of their named DataSources.

The <Text> elements contain no children and <Item> elements can only contain <Qualifier> elements, but <Select> elements can contain <Text>, <Select>, and <Item> elements in addition to a <Qualifier>. These child elements can appear any number of times and in any order. As described earlier, the child elements of a <Select> element are processed, in order, for each Instance of the <Select> element's DataSource after filtering by the <Select> element's "instances" attribute and <Qualifier> child element. The behavior of <Text> and <Item> elements within a <Select> is relatively simple. A <Text> element adds its body to the produced report once for each iteration of the <Select> loop. Likewise, an <Item> element adds the named Property's name and value to the produced report and/or sets the value of a named Variable once for each Instance over which the <Select> iterates.

The behavior of a <Select> element nested within another <Select> element requires some additional explanation. The best way to view nested <Select> elements is to visualize them as nested loops, keeping with our programming language analogy. In this case, the inner loop iterates over its DataSource's Instances once for each iteration of the outer loop. This means that, for each Instance in the outer <Select> element's DataSource, the inner <Select> element iterates over all the Instances in its own DataSource. If both the parent and child <Select> elements reference the same DataSource, they loop independently of each other. This is to say, the counters of the two loops are independent even if they are iterating over the same DataSource. Since the inner <Select> element can also contain all of the <Text>, <Select>, and <Item> elements, <Select> loops can be nested to an arbitrary depth.

5.4.1 Iterating Over Sub-Instances

One special case of the nested <Select> loop concerns Instances that have sub-Instances. As described in the DataSources section, some types of DataSources may have (or define) hierarchical Instance structures with Instances appearing within other Instances. We noted earlier that, when an Instance names a DataSource (using its "datasourceid" attribute) that that Instance iterates over the top-level Instances contained in that DataSource. How, then, does one reach Instances that are children of other Instances?

To iterate over the child Instances of a parent Instance, one uses a child <Select> element (within a parent <Select>) that omits its "datasourceid" attribute. This tells the interpreter that, instead of iterating over the top-level Instances of some DataSource, the <Select> element should iterate over the immediate child Instances of the currently selected Instance. Because we must iterate over the child Instances of some other Instance, <Select> elements with no "datasourceid" attribute can only appear within the body of some other <Select> element. <Select> elements that appear as immediate children of a <Reporting> element must have a datasourceid. Because <Select> elements can be nested within each other to an

arbitrary depth, this allows us to reach child Instances of any depth within a DataSource's Instance hierarchy.

When iterating over the child Instances of a parent Instance, all of the immediate child Instances of that parent are considered. This means that if a parent Instance defines multiple types of child Instances (for example, through multiple uses of nested <Instance> elements in a <FileDataSource>), all of the child Instances loaded within that parent are considered regardless of which <Instance> definition created them. If the author wishes to only iterate over one particular type of child Instance, they need to utilize a <Qualifier> element, as described earlier. Note likewise that the child <Select> element only iterates over the immediate child Instances of the current parent Instance and not all the child-level instances of the parent's DataSource. If the parent Instance has no child Instances, the child <Select> is skipped.

5.4.2 Using Variables

It was noted earlier that using the "setvariable" attribute of an <Item> assigns a Property's value to a named Variable. Recall that <Property> operands in a <Qualifier> use the value of the named Property extracted from the current Instance. However, this only references the current Instance of the local <Select> and not the Instance of any parent <Select> loop. In other words, the Properties of Instances other than the Instance currently selected by the immediately encapsulating <Select> loop are out of scope. We still may wish to have conditions that take into account the values of Properties that are present within a parent Instance. Variables have no scope restrictions and are usable anywhere once they are initialized. If Properties are used to set a Variable within the parent <Select>, the Variable's value can be used within the child <Select> as an operand to the <Qualifier>. In this way, Variables allow authors to utilize the values of Properties that are otherwise out of scope.

Intentionally Blank

Section 6

Conclusions

In summary, OCRL adds a number of useful pieces of functionality to the existing security benchmark languages. It supports the collection of information from a wide range of sources. It can not only collect information from every source that can be used in the OVAL language (through the <OvalDataSource>) but can also gather from files, executables, and WMI in such a way as to preserve the context of the information retrieved. By retaining this context, authors are able to gather a more complete picture of a system's configuration than can be acquired through OVAL alone.

In addition, the language supports taking the structured information extracted in the <DataSources> segment of a report definition and arranging it in an informative report. This report can be as simple as a list of field values or as sophisticated as a nested cross correlation of information pulled from multiple data sources. As a result, the information a user needs in order to make an informed decision as to recommendation compliance is not simply present, but structured in a way that makes it easier for the user to make the compliance decision.

Please do not delete these paragraphs or the final end-of-section mark in your document.
They are important for correct functioning of the RoboTech technical document template.
RoboTech: Version 3.0