opConfig and Compliance Management

Overview

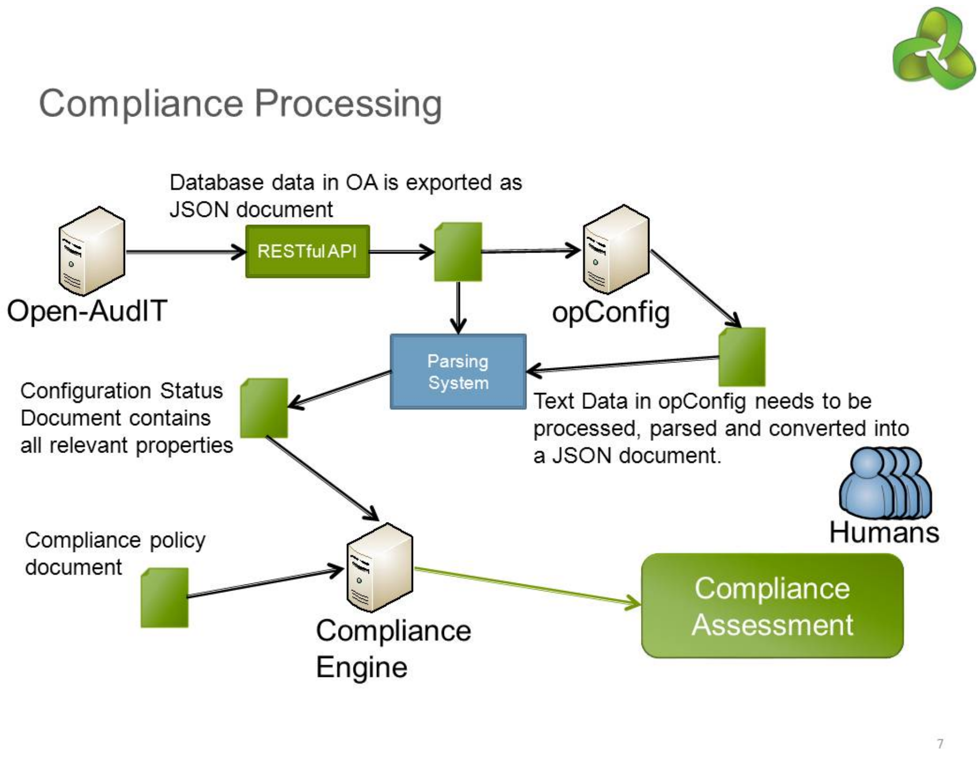
Version 2.0 adds a major new feature to opConfig: Compliance Management. The compliance engine in opConfig is very flexible and capable and this document describes how to configure and use it to assess your infrastructure.

Conceptual Components

Compliance management with opConfig has the following three aspects:

- **Defining Sources:**
    - Where do we get the status and configuration information from?
- **Declaring Policy Rules:**
    - What are the rules to be used to assess your systems' compliance?
- **Evaluation and Presentation:**
    - Which systems' are or were compliant with which sets of rules? What rules are violated or conformed to? How does the assessment status change over time?

Here is a diagram of the component interactions:



Compliance Processing

Database data in OA is exported as JSON document

Text Data in opConfig needs to be processed, parsed and converted into a JSON document.

Configuration Status Document contains all relevant properties

Compliance policy document

Internal Sources of Information

opConfig generally deals with (sets of) commands that are run on a node and collect configuration and status information; the compliance engine fits into that environment and makes use of any command outputs that opConfig has collected.

However, as command outputs are generally unstructured plain text, it is necessary to preprocess these outputs in order to provide both efficient mechanisms for checking compliance with an organization's rules. The section on Configuration Parses describes how this works.

Open-AudIT Enterprise as Source of Information

opConfig 2.0 also integrates with Open-AudIT Enterprise (versions 1.4.1 and newer), and can import the full set of audit information that Open-AudIT has collected. This is especially valuable in situations where one-off snapshots of large numbers of systems are involved, as Open-AudIT is especially capable for this purpose.

Configuration Parsers

Non-structured command outputs need to be condensed and transformed before opConfig can make compliance assessments in an efficient manner. This operation is performed by any number of 'configuration parsers', small components (written in Perl) that act as Subject-matter Experts and digest the textual input into one precise, unambiguous and structured document that's minimal in the sense of only containing relevant facts and properties.

For example, only small parts of the output of a Cisco's "`show interfaces`" command would be relevant for detecting configuration mistakes like directed broadcasts being allowed. opConfig itself cannot contain the domain-specific expert knowledge for all kinds of devices, therefore we decided to make these config parsers extensible components: anybody can provide their own parsers for a particular type of command output.

Naturally opConfig ships with a number of parsers, which can be used as examples for how to build your own. The section "Parser Interface" provides a detailed description of how to do that.

Audit data that is imported from Open-AudIT Enterprise is already in structured form and doesn't necessarily have to be transformed, but can be; for example, if your policies only check installed software then you might want to reduce the amount of material by excluding network port information (which Open-AudIT Enterprise does collect).

Configuration Status Document

The results of post-processing all of a node's command outputs with those parsers are merged into one document, the node's configuration status. This is the basis that the compliance policy rules act on.

Node configuration status documents are time-stamped, regenerated when necessary and older statuses remain in the database. It is thus possible to perform compliance checks against the historic status of a node when necessary.

The configuration status document has no prescribed structure but must be representable as a JSON document - this basically means that you can use nested lists (or arrays), hashes and the usual types of singleton scalars to express and arrange the characteristics that you consider relevant for your node. Naturally the structure that you choose for your parsers to produce must match up with the properties that your policy rules examine.

Compliance Policies

opConfig supports an arbitrary number of compliance policies in parallel. A compliance policy is a named sets of rules, which examine a node configuration status document and trigger exceptions or compliance assertions if a prescribed set of circumstances is encountered.

For example, your high-level policy restriction might be "MySQL servers may not have a Web server active"; this would be translated into a set of policy statements that find systems which are active MySQL servers and tests whether there is a web server software package installed and/or the ports 80 or 443 are active.

A policy rule can assert (negative or positive) compliance for a particular named compliance criterion; all such assertions (exceptions and confirmations) are saved in the database, together with a configurable amount of assertion context. For example, an assertion about a system's misconfigured DNS setup would normally identify the system but could also include the relevant DNS configuration components for further scrutiny or other system aspects.

The policy rules are structurally similar to the "Event Actions and Escalation" rules in opEvents, and the section "Policy Rule Language" below provides the necessary details.

Policy rules are named, time-stamped and versioned: whenever you instruct opConfig to import a policy document it will compare it against the most recent version of this policy and store it with a new revision if there are differences. This is to ensure that you can compare policy behaviours over time.

Currently compliance policies are performed on a per-node basis, but future releases will extend this capability to test coporation-wide policies like "there must be at least two active NTP servers".

Compliance Status Presentation

All compliance policy management and assessment activities in opConfig version 2.0 are performed using the command-line tool `opconfig-cli`. Improved GUI-based policy editing and management features are planned for the next releases.

Compliance status reporting, on the other hand, is already purely GUI-based and integrated within the normal opConfig web interface.

Technical Details

The following sections provide the technical details necessary for configuring and using opConfig's compliance features effectively.

Parser Interface

opConfig's configuration parsers are more than just parsers and we might have called them "*knowledge extractors*" instead. The purpose of such a config parser is to consume a particular type of command output and transform (relevant parts of) it into a tree structure.

Parsers are installed in the directory `/conf/config_parsers` under your opConfig installation dir.

All config parsers must be valid Perl scripts, and we made this choice for efficiency reasons: a language flexible enough to parse and extract information from arbitrary configuration or status command outputs would have been almost as complicated as perl but much less robust. In addition to that you will likely change your policy rules much more often than any of the parsers.

There is no need to worry about these being Perl, however - simple Perl is very simple indeed, and we've made sure that your parsers don't have to follow too many rules for successful integration.

- All parser modules must consist of valid perl code, but don't need to be 'complete' perl Packages or Modules, or even use function definitions: the code is executed as given, sequentially, in a sandbox with a few variables and functions predefined.
- Parser modules may not contain "`use Some::Other::Module;`" constructs at this point.
- The variable "`$input`" always corresponds to the textual input of the command to analyze. The parser may consume and destroy the contents.
- The variable "`$input_structure`" contains a hash reference to the structural representation of the command output, IF and Only IF that command output was detected as tree-structured.
  Otherwise `$input_structure` is undef(ined). At this time only Open-AudIT Enterprise imports provide structured command outputs.
- The variable "`$node`" contains the node name in question, and "`$command`" is the name of the command that produced this output. This can be handy if you want to use one parser for two or more command types with similar outputs.
- The function '`&logger("severity","some message")`' can be used to create diagnostic messages for the opConfig log files. Severity must be one of "debug", "info", "warn", "error" or "fatal".
- If the parser detects an non-recoverable problem with the input then it must return a single string, containing an error message.
- If the parser has successfully analyzed an input text, it must build a tree representation of the collected properties (a deep hash in Perl terms) and return a reference to that tree structure.

Here is an excerpt from one of the parsers that ship with opConfig, called `cisco-interface.pm`:

```
my (%iface,$ifctx,$linectx);
for my $line (split(/\r?\n/,$input))
{
        if ($line =~ /^(\S+) is (.+), line protocol is (.+)$/)
        {
                $ifctx=$1;
                my ($admstatus,$linestatus) = ($2,$3);

                $iface{$ifctx}->{shutdown} = $admstatus eq "administratively down" || 0;
                $iface{$ifctx}->{up} = $admstatus eq "up" && $linestatus eq "up" || 0;
        }
# .... some more if/thens removed ....
}

return { config_features => { interface => \%iface } };
```

This parser creates a tree structure with a top-level key `config_features`, which contains a sub-object `interface` which in turn holds one record per detected interface. All parsers that are available for a node's commands will be run and their results will be merged into one structure.

Parser Caveats and Limitations

- Parsers currently cannot receive structured document representations for XML or HTML command outputs, only for JSON-formatted material. As a parser isn't allowed to make use of external Perl modules (e.g. an XML parser), this limits acceptable inputs to plain text or JSON. This limitation will likely be removed in a future version of opConfig.
- The output structure of a parser MAY overlap with other parsers' output hashes wrt. to the hash keys; thus a parser MAY amend another's config structure by filling in new material - but you need to note the following restrictions:
    - Overwriting of hash structure with a scalar is not supported - the scalar value is ignored.
    - Overwriting of a scalar is not supported - the earlier value remains set and the new value is discarded. This affects only entries that have been set explicitly.
    - all other cases are merged sensibly (ie. scalars are added to arrays, arrays and hashes are merged, etc.)
- Because of this order dependency, parsers must be configured in order of priority.
  If you do use overlapping structures, the most specific parsers need to run first, and establish correct values, while other parsers can later only fill in the blanks. If all your parsers are written so as to create separate (sub)structures this is not an issue.
- Parsers must not produce structure keys that contain "." (more specifically: that contain any characters outside of a-z, A-Z, 0-9, "-" and "_") or keys that are fully numeric.
  These restrictions are necessary to make dot-style structure selector expressions possible and unambiguous.

Parser and Import Configuration

To activate a particular config parser, you have to add an entry for it to `conf/opCommon.nmis` in the `opconfig_parsers` section. The entry must consist of a regular expression (which identifies which commands this parser handles), and of a path pointing to the parser file. Here is an example excerpt:

```
'opconfig_parsers' => [
        # regexp (for command) => parser file name (relative to <omk_conf>)
        [ '^show version$' => 'config_parsers/cisco-version.pm' ],
        [ '^show ip interface$' => 'config_parsers/cisco-interface.pm' ],
...
```

Note that *all* matching parsers will be applied for a particular command, in the order they are given in the configuration.

It's very much recommended that you test your parser for syntax errors with perl before letting opConfig use it: "`perl -cw ./conf/config_parsers/my_new_parser.pm`" will report gross syntactical problems. opConfig's log files will include reports for any problems with a parser that occur when it's actually used.

The result of the parsing/extraction of all of opConfig's input material is what we call the "Node Configuration Status Document": a tree-structured concise expression of properties.

To verify that your parsers have correctly extracted the expected properties, you can export the newest version of the config status document:

```
# if you don't want to see the output on screen, add file=/tmp/MYNEWDUMP.json
/usr/local/omk/bin/opconfig-cli.pl act=export_config_status node=SOMENODENAME
```

Normally opConfig updates the config status document only if are new command outputs available for a particular node; parser modifications are *not* taken into account!

While you are fine-tuning your parser behaviours, you will therefore likely want to force opConfig to re-parse and re-extract the command outputs, which is of course possible:

```
/usr/local/omk/bin/opconfig-cli.pl act=update_config_status node=SOMENODENAME force=1
```

Importing from Open-AudIT Enterprise

For importing audit information from Open-AudIT Enterprise, `conf/opCommon.nmis` requires a few configuration entries in the `opconfig` section:

```
'opconfig_audit_import' => 1, # to enable this functionality
'opconfig_audit_import_url_base' => "http://localhost:8042/omk/oae/", # the base url for YOUR open-audit
enterprise installation
'opconfig_audit_import_user' => "some_user_maybe_nmis_or_adm",                # user needs ro-access
'opconfig_audit_import_password' => "whatever_your_password_maybe",
```

Please note that the actual import does *not* happen automatically but must be triggered using the `opconfig-cli.pl` program (for example, from cron):

```
# if you run it from cron you may want to add quiet=1
/usr/local/omk/bin/opconfig-cli.pl act=import_audit
```

The imported audit information is treated as if a command named "`audit`" had been configured for and run on a particular node, and you can observe the resulting (long) JSON "command outputs" in the normal opConfig gui, perform version comparisons and so on.

The only difference between imported audit information and normal command outputs is that audit info is internally marked as "JSON-structured" and passed to a parser in structured form. To make audit information available in a node's configuration status document it still has to run through a parser, but that parser can do as little as select some parts of the tree structure to pass on. For example, the minimal parser "`jsonpassthrough.pm`" that ships with opConfig just passes on the *whole* audit document.

The Policy Language

As mentioned earlier the compliance policy language is very similar to opEvents' Event Actions and Escalation language.

Here is a quick overview of the structural rules:

- A policy consists of one hash (or "associative array"). All hash keys (=rule numbers) must be numeric, and the keys control the order of rule evaluation.
  Rule numbers do not have to be globally unique, just within the enclosing subpolicy.
- Each hash element must describe either one IF/THEN clause or one EACH/BLOCK iteration.
- THEN statements can be either a single string (describing the actions to take) or a nested sub-policy (in the form of a nested hash).
- EACH/BLOCK iterations always require a nested sub-policy.
- IF statements are single strings, made up from structure or variable selector expressions and Perl operators and expressions.
- The available actions for THEN statements are `ok()`, `exception()`, `CONTINUE()` and `LAST()`.
- EACH statements consist of a variable name (for the iterator variable to be) and a structure selector expression (for the objects to iterate over).
- The policy engine invokes policy rules with a number of pre-defined structure variables, to provide access to the configuration status document, the current node name and a few others.

Here is an example policy fragment:

```
20 => {
        IF => '$NODE.os_info.os eq "IOS"',
        THEN => {
                201 => {
                        IF => '$NODE.config_features.tcp-small-servers',
                        THEN => 'exception("TCP small servers should be disabled",3,node=$NODENAME,config=$NODE.
config_features)' },
                202 => {
                        IF => '$NODE.config_features.udp-small-servers',
                        THEN => 'exception("UDP small servers should be disabled",3,node=$NODENAME,config=$NODE.
config_features)' },
...
```

As you can see the IF statements contain `$VARNAME` expressions, which the policy engine expands and substitutes with actual values; the expanded IF statements are then handed to Perl for evaluation. An evaluation to anything but zero, an empty string or undef(inded) is considered a logical true value, and will cause the associated THEN statement to be executed. In the example the first IF selects IOS systems for the nested sub-policy, but the rules 201 and 202 have leaf THEN statements that contain the actions to take.

Structure Variables and Structure Selectors

At this time the policy engine provides the following pre-defined structure variables to every policy:

Variable Name	Description
$NODE	The complete node configuration status document, which generally will be a deeply structured tree.
$NODEINFO	A structure that contains the most essential node-related characteristics, i.e. node name, hostname, ip address(es), group information and so on.
$NODENAME	A convenience variable that contains only the node name, mainly for reporting and context capture.

Structure variables can be accessed in IF, THEN and EACH statements. To do so, the structure variable name is given (including the leading "$" sign), optionally followed by further substructure accessor directives (which are deliberately similar to MongoDB's Dot Notation):

- `$VARNAME.something` is translated into accessing key `something` in the hash structure `$VARNAME`. If `$VARNAME` is not a hash, the expansion fails and a "Rule Error" exception is generated.
  Hash accessors do nest: `$VARNAME.outerkey.innerkey` is perfectly valid and performs two hash key accesses.
  If key `something` does not exist or is undef(ined), Perl's `undef` is inserted.
- $VARNAME.123 is translated into accessing the 123rd element of the list structure `$VARNAME`. Like before, should `$VARNAME` not be an array you will see a "Rule Error" exception. Array accesses nest, too: `$VARNAME.4.7` works, and you can even use negative indices to access the end of a list like this: `$VARNAME.-1`. Array indices start at *zero* (=the first element), not one.
- Hash key and array indexing accesses can be mixed: `$NODE.outerstructure.2.innerstructure` works fine. If an indirection fails because the structure type doesn't match your access expression a "Rule Error" exception is generated.
- If you use `$VARNAME` without substructure accessors, then the variable will be expanded as follows:

  a. If it's a string or number, that will be inserted (strings will be quoted with double quotes).
  b. If `$VARNAME` does not exist or is undefined, Perl's `undef` is inserted.
  c. If `$VARNAME` is a hash or an array, the number of elements that it contains will be substituted.

Note that a structure variable name is only allowed at the beginning of an access expression (for now), so you cannot dynamically create a hash key from another access expression.

If you happen to need `'$NOTAVAR`' to be kept untranslated, simply prefix it with a backslash, e.g. `\$NOTAVAR`.

Besides the pre-defined structure variables, iterations with EACH also create variables.

IF statements

An IF statement can contain any number of structure access expressions, operators and Perl constructs. All structure accesses ($THISNTHAT.substruct...) are replaced as described above, then Perl is used to evaluate the expression. Evaluation errors (ie. syntactical mistakes) result in "Rule Error" exceptions.

Iteration with EACH/BLOCK

To iterate over structures or loop over arrays you can use nested EACH/BLOCK constructs, as in the following example:

```
220 => {
        EACH => 'INTF=$NODE.config_features.interface',
        BLOCK => {
                1 => {
                        IF => '$INTF.shutdown',
                        THEN => "CONTINUE()" },
                2 => {
                        IF => '$INTF.smurfamplifier',
                        THEN => 'exception("Interface should have directed broadcasts disabled",3,
node=$NODENAME,interface=$INTF_INDEX,interface_details=$INTF)' },
...
```

The EACH statement must contain exactly one `"NEWVARNAME=struct.accessor.sub"` expression.

`NEWVARNAME` must not be all numeric and may contain only characters from the set A-Z, 0-9 and the underscore "_". `NEWVARNAME` must also not clash with an already defined variable of the same name, nor can it be *anything*`_INDEX`.

`NEWVARNAME` is set to each of the elements in the hash or array structure identified by the accessor in turn, and becomes available as `$NEWVARNAME` for the rules in the subpolicy in BLOCK. In addition to that, `$NEWVARNAME_INDEX` will be set to the index or hash key in question.The rules in BLOCK are run for each of the strutures we're iterating through.

If the structure access fails because there is no object identified by the structure selector, or because it is not a hash or array, or empty, a log message is created but ***no*** rule error exception is generated; this is because it's fairly common to come across an empty substructure when iterating.

In the example above, rule 220 establishes the iteration over all substructures in `$NODE.config_features.interface` (which holds Cisco interface properties, if you use the default `cisco-interface.pm` parser). The BLOCK of rules can access individual interfaces' info blocks via `$INTF`, and the interface name was used as hash key and thus is available as `$INFT_INDEX`.

EACH/BLOCK statements nest, and mix perfectly with IF/THEN statements.

Policy Actions

Policy actions in THEN statements contain an arbitrary number of AND-separated `ok()`, `exception()`, `CONTINUE()` or `LAST()` expressions.

Expression	Description	Example
ok("rule name", contextitems...)	Affirms that a node complies with "rule name"	`ok("proxy arp is disabled", node=$NODENAME, interface=$INTF_INDEX)`
exception("rule name", priority,contextitems...)	Asserts that a node does NOT comply with "rule name"	`exception("IIS must not be active on DB servers",9,node=$NODENAME)`
CONTINUE() CONTINUE (LOOPVARNAME)	Skips the remaining rules in this iteration block (and possibly outer loops as well), and continues with the next iteration of `LOOPVARNAME`.	`CONTINUE(INTF)`
LAST() LAST (LOOPVARNAME)	Terminates this iteration block, possibly outer loops as well - up to and including iteration with `LOOPVARNAME`	LAST()

Exceptions and compliance affirmations can use arbitrary "rule name" strings. Exceptions must be given a numeric priority (0 lowest, 10 highest).

Context items are for capturing and saving contextual information with the exception event. Expressions are of the form `"contextname=$STRUCTURE.subaccessor"`, which will be expanded and attached to the exception/affirmation event with the name "contextname". In the examples above the context item "node" will be given the value of the node name, and in many others deeper structures (e.g. all settings for an interface) are captured as well. opConfig does not enforce any limits to the number or structure of context items to capture.

You can have multiple ok() or exception() expressions in a single THEN (unlikely as that may be), but at most one LAST or CONTINUE.

Policy Caveats and Limitations

- The parsing system currently used for the `THEN` statements uses the string `<space>AND<space>` as separator, therefore the exception or ok rule name cannot contain that text. Lowercase <space>`all`<space> is fine, however.
- At this time, all compliance policy rules are evaluated on a per-node basis. Extending this system to support policies that can span the whole organization are already planned for future releases.

Policy Management

All policies are named and versioned, and managed through `opconfig-cli.pl` (but advanced GUI-based management is already planned for the next releases).

To make a policy available to opConfig, it must be imported like this:

```
/usr/local/omk/bin/opconfig-cli.pl act=import_policy name="my new policy name" file=/some/path/pointing/to
/my_policy.nmis
```

If the policy file `my_policy.nmis` is different from any earlier version of "my new policy name" then opConfig will store the new version with an automatically updated revision and timestamp.

You can list the existing policies, versions and timestamps, as well as export particular ones:

```
/usr/local/omk/bin/opconfig-cli.pl act=list_policies
...
Policy                          Version Date
cisco-testpolicy    2       2014-09-03T16:46:26
...

# add file=somefilename.nmis if you don't want to see the export on screen
/usr/local/omk/bin/opconfig-cli.pl act=export_policy name=testone
```

Policy Evaluation

At this time compliance policy assessments are *not* performed automatically but have to be triggered with `opconfig-cli.pl`:

```
# to check only a particular node add node=the_node_name
/usr/local/omk/bin/opconfig-cli.pl act=check_compliance name='policy name'
```

This command will not produce output unless there are fatal problems during the policy evaluation. All compliance assessments (and any "Rule Error" exceptions caused by benign rule problems) are stored in the database and are managed using the opConfig gui (Menu Views, Entry "Compliance Status").

Setup Sample Compliance for Cisco Devices

opConfig comes with a sample compliance policy for Cisco devices based on the NSA Cisco Best Practices document

Import the Compliance Template

```
/usr/local/omk/bin/opconfig-cli.pl act=import_policy name="cisco-nsa" file=/usr/local/omk/conf
/compliance_policies/cisco-nsa.nmis
```

View the Available Compliance Templates

```
/usr/local/omk/bin/opconfig-cli.pl act=list_policies
```

The result will look like this

```
Copyright (C) 2012 Opmantek Limited (www.opmantek.com)
This program comes with ABSOLUTELY NO WARRANTY;
See www.opmantek.com or email contact@opmantek.com


opConfig 1.0 is licensed to Opmantek for 50 Nodes


Policy      Version   Date
cisco-nsa   1         2014-10-27T11:21:10
```

Run the Cisco NSA Compliance Template

```
/usr/local/omk/bin/opconfig-cli.pl act=check_compliance name='cisco-nsa'
```

View the Compliance Status

You can now check the Compliance Status in the opConfig GUI. Access the opConfig GUI at http://YOUR_SERVERNAME/omk/opConfig, login and then from the Menu Bar "Views -> Compliance Status".

Create Compliance Report

You can now check the Compliance Status by generating a compliance report, in csv format, which shows compliance to Compliance Policies.

```
/usr/local/omk/bin/opconfig-cli.pl act=create_compliance_report file=/tmp/cisco-nsa_compliance.csv node='the-
node-name'
```