Usable Declarative Configuration Specification and Validation for Applications, Systems, and Cloud

Salman Baset, Sahil Suneja, Nilton Bila, Ozan Tuncer, Canturk Isci
IBM T.J. Watson Research Center

Abstract

Diagnosing misconfiguration across modern software stacks is increasingly difficult. These stacks comprise multiple microservices which are deployed across a combination of containers and hosts (VMs, physical machines) in a cloud or a data center. The existing approaches for detecting misconfiguration, whether rule-based or inference, are highly specialized (e.g., security only), cumbersome to write and maintain, geared towards a host (instead of container images), and can result into false-positives or false-negatives.

This paper introduces configuration validation language (CVL), a declarative language for writing rules to detect misconfigurations that can, for instance, impact security, performance, functionality. We have built a system, ConfigValidator, which applies the CVL rules across a multitude of environments such as Docker images, running containers, host, and cloud. The system is running in production and has scanned thousands of Docker images and running containers for identifying misconfigurations.

1 Introduction

Misconfiguration has been a major source of functional and security problems in software, cloud-based systems, and cloud providers [17, 30]. Examples of misconfiguration include an incorrect value for an item in a configuration file, open permissions for a file, incorrectly configured security groups in an infrastructure-as-a-service cloud, or expired TLS certificates. As software stacks become more complex with interdependent components, diagnosing misconfigurations that impact functionality and security is increasingly difficult.

Part of this problem is due to use of software components in the software stack which are third-party (open source or commercial). It is unreasonable to expect the teams (DevOps) deploying the software stack to master all the configurations of third party components. Yet another problem is the use of automation for deploying software stacks. DevOps personnel often use existing automation tools or scripts for deploying these third-party software (of which they have limited understanding) and integrate these automation scripts with their own automated deployment framework. It is unrealistic to assume that DevOps personnel can master configurations spread across all automation. The problem is further exacerbated by the fact that software components do not ship with a set of recommended configurations for production that can be programmatically evaluated to validate misconfigurations.

Middleware'17, Las Vegas, Nevada USA
2017. 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

Traditionally, the approaches for detecting misconfiguration can be categorized into two broad categories: rule-based [17, 19] and inference-based [31]. In rule-based approaches, rules are typically defined using scripts (typically, shell scripts) to validate a configuration. In a nut shell, these approaches search for a regular expression in a configuration file. A majority of security checklists such as Center for Internet Security (CIS) security benchmarks [5] XCCDF (The Extensible Configuration Checklist Description Format) [19] fall into this category. One of the major drawbacks of such approaches is that these checklists are often specific to a function (e.g., security), are cumbersome to use by average skilled developer for validation of other configuration validation tasks (e.g., configurations related to performance). Moreover, their efficacy is as good as the underlying script and are thus difficult to reason from a validation perspective. Consequently, recent approaches are taking a declarative approach for defining configuration rules [6, 17]. However, such approaches still require significant expertise from a DevOps person ([17]) or are specific to an application component ([6]) of the software stack. The inference-based approaches cite explicit rule specification as burdensome and thus rely on inference and machine learning techniques to validate configuration [31]. Thus, by design, they have some error deltas built into them. Since checking of security misconfigurations is not explicitly guaranteed, our experience in building production cloud systems informs us that inference-based approaches have rarely found use in production systems on a continuous basis.

In this paper, we present our system, ConfigValidator , for seamless configuration validation across heterogeneous entities - applications, systems and the cloud. We believe that rule specification should be declarative which makes validation checks easy to encode, comprehend, extend and maintain. To assist users – specialists and non-experts alike – in this task, we have designed a declarative Configuration Validation Language (CVL). We show ConfigValidator's coverage in terms of the entities and checklists currently supported, and efficiency both in execution time and rule specification in CVL. ConfigValidator has been operational in the IBM Cloud as part of IBM Vulnerability Advisor [12] for over an year. It has been validating of the order of tens of thousands of containers and images daily, capturing security misconfigurations. [1]

This paper has the following contributions:

- Introduces a Configuration Validation Language (CVL) with YAML-based syntax for writing and maintaining configuration rules for a single application component, or across multiple components of a software stack and cloud deployment.

[1] The rule set is constantly being expanded very soon the users will have the ability to write and edit their own rules, once ConfigValidator is open sourced.

- Describes a system, currently running in production, that validates the configuration declarations from CVL against real configurations that are converted into a 'tree' or a 'schema' like structure.
- Evaluates the rules against a multitude of environments, including the host systems, Docker images, running containers, as well as configuration residing inside the applications or the cloud.

2 Background and Related Work

One of the main goals for ConfigValidator is to validate configuration from an applications, hosts, or cloud environment. For ease of exposition, we use the word *entity* when referring to an application, host, or a cloud. In this section, we first describe the different styles in which configurations can exist across entities. Then we differentiate ConfigValidator against existing approaches for configuration validation.

2.1 Configurations

In this paper, we define configuration to be the following:

- **Configuration file(s)** containing modifiable parameters for software execution or system function. For example, nginx.conf or /etc/fstab stores configuration parameters for nginx web server [9] and device file systems, respectively.
- **System state** such as file and directory permissions, ownership, software packages and their versions.
- **Custom Configuration** stored within an entity's runtime state (e.g., cloud security groups) or in custom formats.

2.1.1 Configuration Files

This configuration type forms the bulk of configurations across all entities. The format of configuration files typically follows two types of patterns:

Key-value tree pattern Consider the configuration file (`foo-kv1.conf`) shown in the figure below. It has three sections, namely, A, B, and C. Section A has one key, namely A. Sections B and C have two one keys defined, respectively, namely, `subB1` and `subB2`, and `subC1`.

```
# foo-kv1.conf
A = valA
B: {
    subB1 : valsubB1
    subB2 : valsubB2
}
C: {
    subC1 : valsubC1
}
```

This pattern occurs in many configuration files such as Nginx [9], MySQL [8], Apache [3]. However, the section separators used may be different.

Schema pattern Consider the configuration file (`foo-schema1.conf`) shown in the figure below. It does not have a key / value or section structure. Rather, it has some values defined which are separated by a separator. The program that reads this configuration file is able to interpret the meaning of each component on each line depending on its position. Thus, the 'key' in this case is implicit.

```
# foo-schema1.conf
A1 B1:C1 D1=B1 E1
A2 B2:C2 D2=B2 E2
```

This pattern occurs in many configuration files such as `/etc/passwd`, `/etc/group`, `/etc/audit/audit.rules`. However, the separators used may be different.

Note that in some configuration files, the *schema* pattern can occur inside a *tree* pattern and vice versa. However, this mixing typically increases the configuration management complexity.

2.1.2 System State

System state comprises file and directory permissions, ownership, software packages and their versions. Depending on the configuration validation rules, certain files or directories may be checked for their presence or absence, having the right ownership and permissions, and belonging to packages with the right versions.The example below shows the permissions for `/etc/sysctl.conf` file.

```
-rw-r--r-- 1 root root 2210 Dec 21 18:09 /etc/sysctl.conf
```

2.1.3 Custom Configurations

Some applications (e.g., MySQL) either store configuration in their own formats or need certain commands to be executed for retrieving configuration for verification (e.g., SSL enabled). As such, these configurations are not stored in a text file and must first be retrieved by executing application-specific commands. In other cases, configurations might exist within an entity's runtime state (e.g. in-memory data structures). The OS also does not always explicitly expose all of its configuration. For example, `sysctl.conf` typically contains only a subset of the kernel parameters available via `sysctl -a`. Similarly, cloud platforms typically store state about cloud resources in a central/master management node, typically accessible over APIs or HTTP(S) endpoints. Obtaining these types of configurations requires querying the entity-specific interfaces, commands or APIs. Another alternative is to extract such configurations from the source code [21].

2.2 Configuration Validation Techniques

Enterprise policies dictate compliance with security checklists and configuration guidelines issued by specialized or authorized organizations such as CIS, DISA STIG, FEDRAMP, FISMA, PCI DSS, USGCB, NIAP, OSSG, and GovReady amongst others [15, 16, 20]. An example rule could be to 'disable root login over ssh' [13]. A typical configuration validation process would include comparing the values of the target configuration parameters against the ones specified in such checklists. For this particular example, this would translate to checking the value of `PermitRootLogin` inside `sshd_config` configuration file.

A popular configuration validation approach to enforce these rules and guidelines is to encode them in the form of scripts to be executed on the target machines [14]. However, such ad-hoc scripts become hard to maintain as they are extended. An alternative is to encode compliance rules and guidelines in more structured specification formats. Examples include the SCAP standard's XML-based configuration description and vulnerability specification languages- XCCDF, OVAL, and OCIL [19]. But, even standardized specification formats like XCCDF can be hard and cumbersome to comprehend and encode in, as we highlight in Section 4.2.

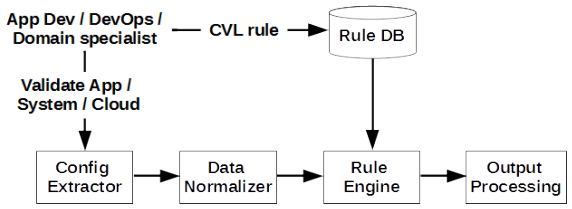

Figure 1. Conceptual Architecture of ConfigValidator .

We believe rule specification should be more declarative that makes validation checks easy to encode, comprehend, extend and maintain. There has been recent support of this philosophy in ConfValley [17], which although still requires significant DevOps expertise, and Chef Inspec [6], which seems to have been developed along the same time as ConfigValidator .

Although Inspec and ConfigValidator share the same belief of clear, declarative, and easy-to-comprehend specification of compliance, security and policy requirements, but there are a few subtle differences. While Inspec requires writing application-specific custom parsers from scratch, leveraging opensource Augeas parser makes ConfigValidator easier to extend to more targets. Although Chef Inspec supports declarative constructs (via app-specific libraries), Chef Compliance's CIS-rules encoding for Inspec-based validation just uses plain bash scripts instead, as discussed in Section 4.2. Another differentiation is ConfigValidator's ability to work against system configuration frames [24], which allows it to validate systems without requiring any local installation or remote access.

An alternative configuration validation approach is to use deployment automation tools, such as Chef [7], Puppet [10], Ansible [2], to validate the deployed configurations encoded as part of the automation process. However, such approaches suffer from two problems. First, assertions about correct configurations are not cleanly specified and are mixed in the code. Second, lack of clean specification makes it difficult to validate configurations. Deployment and validation are two distinct operations, and mixing them (even in the deployment code) makes detecting misconfiguration harder and maintenance of this code difficult. Facebook [25] also follows the configuration-as-code approach, and consequently has to employ an entire ecosystem to keep misconfigurations in check, including code review, compiler-based invariant verification, continuous integration testing, incremental rollout and rollback mechanisms.

In addition to these explicit rule-based validation approaches, there exist complementary code-level and system-level approaches to prevent or troubleshoot misconfigurations. Examples of the former include inferring configuration constraints from the source code [18, 29], dynamic flow analysis [11], and instrumentation [26, 32]. Examples of system-level misconfiguration detection approaches include snapshots diff-ing [28], peer-based comparison [27], and kernel level dependency tracking and speculative execution [23]. Out of scope are configuration storage and distribution systems like Akamai's ACMS [22].

3 Design and Implementation

In this section, we describe how ConfigValidator (i) extracts a given entity's (application / system / cloud) configuration parameters, (ii) normalizes them, and (iii) validates them against specific rules/checks. We also present the key attributes of our Configuration Validation Language (CVL), which is used to encode these rules.

3.1 Architecture

ConfigValidator's architecture in Figure 1 is conceptually similar to other rule-based systems. However, the novelty lies in (i) how rules are defined and interpreted by the system, and (ii) how the same rules can be seamlessly run against the configurations belonging to a multitude of environments such as the host, guest VMs, containers, Docker images, as well as the ones residing inside an application or the cloud runtime. The roles of the different ConfigValidator components are described as follows.

Config Extractor We use Crawler open source tool [1] developed by our team to extract an entity's configuration files, and, optionally, metadata such as file/directory permissions and ownership, software package versions, etc, when required by specific validation rules. For configurations that don't exist in the form of text files, and instead reside inside the entity's runtime (e.g., in-memory data structures) or in custom formats, these would have to be extracted by querying the entity-specific commands, interfaces or API calls. The Crawler tool contains several application-specific plugins to extract such runtime state, which ConfigValidator can then consume. In other cases, a script (or a crawler plugin) would still need to be written to perform this state extraction.

ConfigValidator's seamless multi-platform configuration validation capability is derived in part from the crawler's ability to run directly against hosts, VMs, Docker images and running containers.

Data Normalizer This module normalizes the extracted configuration data based on its type. It converts the raw data inside configuration files into a tree or schema structure for further processing. We use the open source Augeas tool [4] tool to perform this conversion. In Section 3.3, we discuss the challenges related to Augeas parsing.

Rule Engine The rule engine is the brain of ConfigValidator . It applies the validation checks (e.g. a config parameter's value correctness) on the normalized configuration data and produces a validation output. It takes as input entity-specific rule definitions written in the Configuration Validation Language (CVL) (Section 3.2). The engine itself is written in Python. For single entity validation checking, the rule engine simply applies the corresponding rules against the entity's configuration parameters. For cross-entity validation (see 'Composite rules' in Section 3.2), the rule engine performs a logical conjunction/disjunction over the per-entity rule evaluations for each component entity.

Output Processing The output processing module converts the rule validation output from the rule-engine into a human readable format. It combines the rule engine's validation result with a rule description, validation output description and a possible suggestive action, obtained from the rule specification file.

3.2 Configuration Validation Language

Configuration Validation Language (CVL) defines the configuration validation rules which are fed as an input to ConfigValidator . CVL is designed so that potentially any user– be it a developer, an IT admin, a DevOps personnel, or a domain specialist – can easily specify configuration assertions to be checked against configuration files in a simple declarative syntax. Below, we list its key properties.

Syntax CVL uses the YAML syntax for encoding rules. The choice was motivated by its simplicity, brevity, declarative nature, and ease of understanding, which aids ConfigValidator in its goal for comprehensible and maintainable rules. YAML is also popular with many modern configuration and deployment tools such as Docker Compose, Ansible, and Kubernetes.

CVL Keywords A CVL rule comprises of a handful of key-value pairs. The keys in the key-value pairs are "keywords" in CVL. They are interpreted by ConfigValidator during rule execution. CVL has a total of 46 keywords across all rule types and entity description. A configuration rule typically has no more than ten keywords.

Keywords in CVL are grouped as follows.

Keywords Common Across Rules These keywords are used to define an entity and rules for that entity. These keywords include entity name, entity rule file, the path for a file containing CVL rules for an entity, a parent rule file if any, rule type, rule description, the value to match (or not match), an output string to generate in case of a success (or failure), and various tags specified by the rule writer. The tags can be used for filtering based on compliance standard such as #HIPAA, or ruleID from an official checklist such as #cisubuntu14.04_2.1). A total if 19 such keywords are defined. These keywords are shown in bold in Listings 1-5.

Listing 2. Config tree rule example

```
config_name: ssl_protocols
config_path: ["server", "http/server"]
config_description: "Enables the specified SSL protocols."
preferred_value: [ "TLSv1.2", "TLSv1.3" ]
non_preferred_value: [ "SSLv2", "SSLv3", "TLSv1", "TLSv1.1" ]
non_preferred_value_match: substr,any
preferred_value_match: substr,all
not_present_description: "ssl_protocols is not present."
not_matched_preferred_value_description: "Non-recommended TLS ver."
matched_description: "ssl_protocols key is set to TLS v1.2/1.3"
tags: ["#security", "#ssl", "#owasp"]
require_other_configs: [ listen, ssl_certificate, ssl_certificate_key ]
file_context: ["nginx.conf", "sites-enabled"]
```

Keywords Specific to Rule-Types CVL allows a rule writer to specify five types of rules based on the configuration type (see Section 2.1. The keywords specific to each rule type are shown in parenthesis.

- config tree - the configuration has a hierarchical tree structure (9 keywords).
- schema - the configuration has a SQL-table like structure. (6 keywords).
- path - the configuration is path or directory with associated metadata (that may or may not exist). (6 keywords).
- script - the configuration needs to be extracted from within the target entity's runtime state using a script. (3 keywords).
- composite - the rule is an aggregation across multiple entities. (3 keywords).

Depending upon the rule type, a CVL rule definition would include certain type-specific keys such as `config_path` for tree-type as in Listing 2 and `query_constraints` for schema-type as in Listing 3. This provides the configuration parameter context for the rule engine to apply a particular rule against. Listings 1-4 show a rule definition in CVL for composite, config tree, schema, and path rule types for various applications and host configurations.

Listing 3. Schema rule example

```
config_schema_name: check_tmp_separate_partition
config_schema_description: "Check if /tmp is on a separate partition"
query_constraints: "dir = ?"
query_constraints_value: ["/tmp"]
query_columns: "*"
non_preferred_value: [""]
non_preferred_value_match: exact,all
not_matched_preferred_value_description: "/tmp not on sep. partition"
matched_description: "/tmp is on a separate partition"
tags: ["#cis", "#cisubuntu14.04_2.1"]
```

Listing 4. Path rule example

```
path_name: /etc/mysql/my.cnf
path_description: "Permissions and ownership for mysql config file"
ownership: "0:0"
permission: 644
tags: [ "#owasp" ]
```

Manifest To provide the rule engine the complete context to validate configurations, a per-entity manifest specifies (i) the location(s) within the target environment to search for the configuration file(s) in, and (ii) the rule specification file that groups together all CVL rule definitions for the entity. The manifest also contains a few other keys to aid ConfigValidator in its rule processing, such as the rule-type (as discussed above), an enabled/disabled boolean, etc. Listing 5 shows an example manifest for nginx application.

Listing 5. Manifest for nginx configuration validation

```
nginx:
  enabled: True
  config_search_paths:
    - /etc/nginx
  cvl_file:
    "component_configs/nginx.yaml"
```

Inheritance CVL enables easy extensibility atop the baseline configuration checks created by application developers or community users. When these rules are run against a target environment, some of them may need to be updated to match deployment-specific peculiarities. To this end, a rule writer may inherit rules from a parent CVL file, override the rule checks (e.g., by updating the acceptable configuration parameter value), or disable the rules.

Composite Rules An application such as nginx may be dependent on system configuration such as sysctl. For complex configuration checks that span multiple entities, CVL allows specifying 'composite' rules. Listing 1 highlights this feature across three entities, by defining a rule to be true only when (i) MySQL is running with authorized SSL certificates, (ii) IP forwarding is disabled, and (iii) nginx has SSL enabled on listening sockets.

Listing 1. Composite rule example

```
composite_rule_name: "mysql ssl-ca path and sysctl and nginx SSL"
composite_rule_description: "Check if nginx is running with SSL, ip_forward is disabled, and mysql server ssl-ca has a cert"
composite_rule: mysql.ssl-ca.CONFIGPATH=[mysqld].VALUE == "/etc/mysql/cacert.pem" && sysctl.net.ipv4.ip_forward && nginx.listen
tags: ["docker", "nginx", "sysctl"]
matched_description: "mysql server ssl-ca has a cert, ip_forward is disabled, and nginx has SSL enabled."
not_matched_preferred_value_description: "Either mysql server ssl-ca does not have a cert, or ip_forward is enabled, or nginx has SSL disabled."
```

Applications	apache, nginx, hadoop, mysql
System services	audit, fstab, sshd, sysctl, modprobe
Cloud services	openstack, docker

Table 1. Targets supported by ConfigValidator

Tool	Specification Language	Implementation Language	Time to run 40 rules
ConfigValidator	YAML	Python	1.92 s
Chef Inspec	Ruby	Ruby	1.25 s
CIS-CAT	XCCDF/ OVAL	Java	14.5 s
OpenSCAP*	XCCDF/ OVAL	C	0.4 s

Table 2. Comparison across validation tools. *: OpenSCAP was run against different rules than the others

3.3 Challenges in Configuration Parsing

As mentioned in Section 2.1, configuration files typically follow a key-value tree pattern or a schema pattern or a mix of the two. While it is possible to convert a schema pattern into a key-value tree pattern and a vice versa, we determined after some experimentation that it would actually increase the amount of work needed to parse the configurations and digress us from our core work of configuration validation. Thus, we maintain the 'natural' format of configuration files while parsing them using Augeas [4].

4 Evaluation

In this Section, we first highlight ConfigValidator's coverage in terms of the entities and checklists supported today. Next, we compare it qualitatively and quantitatively against existing configuration validation tools.

4.1 Coverage

As shown in Table 1, currently ConfigValidator supports 11 different target types, spanning 135 rules in total. These checks are in adherence with the CIS benchmarks, except for Apache, Nginx and Hadoop applications, which conform to OWASP, HIPAA, and PCI standards, and Openstack which conforms to OSSG guidelines. ConfigValidator presently covers 41% of the CIS Docker checklist, and all of the audit rules of the Ubuntu checklist. Work is under progress to increase ConfigValidator's rule coverage, and we hope to achieve a community boost when our ongoing internal-clearance process for opensourcing completes.

4.2 Comparison

Here we compare ConfigValidator with other compliance/-validation engines – OpenSCAP, CIS-CAT and Chef Inspec – quantitatively in terms execution times, and qualitatively in terms of effort required to encode a rule. Both OpenSCAP and CIS-CAT follow the XCCDF/OVAL specification formats, while ConfigValidator and Chef Inspec employ declarative rule specifications.

We selected 40 CIS rules common to ConfigValidator , Chef Inspec and CIS-CAT. These rules target validation of system services in Ubuntu Linux, as shown in Table 1. We targeted maximum common rule coverage across the different engines. However, since OpenSCAP does not implement CIS rules yet, we ran it against random 40 rules from its Ubuntu security guide (XCCDF) to get it's expected validation time. Table 2 shows the average time taken to run the same rules under the 4 engines, as well as the corresponding rule specification and engine implementation languages.

The unusually high time for CIS-CAT should not be related to XCCDF/OVAL since openSCAP also uses it as its rule specification language. It might be due to JVM overhead, or related to some license checking during initialization, since CIS-CAT is a commercial software and not opensourced unlike the other tools. Other than that, the engines take the expected amount of time to run, given their underlying implementation language. A catch may be that although Inspec supports several declarative constructs to encode rules (via app-specific `Inspec::Resources` libraries), Chef Compliance's CIS-rules encoding for Inspec-based validation boils down to just bash scripts as shown at the bottom of Listing 6.

Listing 6 also highlights how a rule is encoded under the different tools. For brevity, we trimmed down the contents of several XCCDF XML flags in the listing. In the particular example shown, it takes 45 and 10 lines respectively to specify the rule with XCCDF/OVAL and ConfigValidator . These emphasize our belief that ConfigValidator's declarative specification is easier to encode, reason, and maintain, than bash scripts and XML-based encodings.

5 Contribution

ConfigValidator has been deployed in production as part of IBM Container Service's Vulnerability Advisor [12]. It has been operational for over an year, and has been validating on the order of tens of thousands of containers and images daily, and identifying security misconfigurations. Its ability to work against system configuration frames [24] allows it to validate systems without requiring any local installation or remote access, although it is equally applicable and functional in these settings as well. Process is under way to clear ConfigValidator for opensourcing, which shall enable leveraging community support to increase ConfigValidator's coverage further.

6 Observations and Limitations

While CVL eases rule encoding and checklist maintenance, it might require a one-time parsing effort ('normalization', Section 3.1) for an entity's configuration parameters, in cases where the entity is not already supported in ConfigValidator (Table 1). This parsing effort is greatly simplified by leveraging the Augeas configuration editing tool, but for a new rule writer

Listing 6. Comparing rule encoding for "Disable SSH Root Login" across different formats

```
                                ******* OpenSCAP: XCCDF/OVAL [45 Lines] *******

<select idref="xccdf_org.ssgproject.content_rule_sshd_disable_root_login" selected="true"/>
:
<Rule id="xccdf_org.ssgproject.content_rule_sshd_disable_root_login" selected="false" severity="medium">
  <title xml:lang="en-US">Disable SSH Root Login</title>
  <description xml:lang="en-US"> The root user should never be allowed to login to a system ... </description>
  <reference href="http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r4.pdf">AC-3</reference>
  :
  <rationale xml:lang="en-US"> Permitting direct root login reduces auditable information ... </rationale>
  <ident system="https://nvd.nist.gov/cce/index.cfm">CCE-</ident>
  <check system="http://oval.mitre.org/XMLSchema/oval-definitions-5">
    <check-content-ref name="oval:ssg-sshd_disable_root_login:def:1" href="ssg-ubuntu1604-oval.xml"/>
  </check>
  :
</Rule>
:
<definition class="compliance" id="oval:ssg-sshd_disable_root_login:def:1" version="1">
  <metadata>
        <title ... /> <affected ... /> <description ... /> <reference ... />
  </metadata>
  <criteria comment="Check...sshd_config" negate="true" test_ref="oval:ssg-test_sshd_permitrootlogin_no:tst:1"/>
</definition>
:
<ind:textfilecontent54_test check="all" check_existence="none_exist" comment="Tests the value of the
PermitRootLogin[\s]*(&lt;:nocomment:&gt;*) ... " id="oval:ssg-test_sshd_permitrootlogin_no:tst:1" version="1">
  <ind:object object_ref="oval:ssg-obj_sshd_permitrootlogin_no:obj:1"/>
</ind:textfilecontent54_test>
:
<ind:textfilecontent54_object id="oval:ssg-obj_sshd_permitrootlogin_no:obj:1" version="2">
  <ind:filepath>/etc/ssh/sshd_config</ind:filepath>
  <ind:pattern operation="pattern match">^[\s]*(?i)PermitRootLogin(?-i)[\s]+no[\s]*(?:|(?:#.*))?$</ind:pattern>
  <ind:instance datatype="int">1</ind:instance>
</ind:textfilecontent54_object>


                                ******* ConfigValidator: YAML [10 Lines] *******
config_name: PermitRootLogin
tags: ["#security","#cis", "#cisubuntu14.04_5.2.8"]
config_path: [""]
config_description: "Enable root login."
file_context: ["sshd_config"]
preferred_value: [ "no" ]
preferred_value_match: substr,all
not_present_description: "PermitRootLogin is not present. It is enabled by default."
not_matched_preferred_value_description: "PermitRootLogin is present but it is enabled."
matched_description: "Root login is disabled."


                                ******* Chef Inspec: Ruby (Expected) [6 Lines] *******
control 'sshd-06' do
  impact 1.0
  title 'Server: Do not permit root-based login or do not allow password and keyboard-interactive authentication'
  desc 'Reduce the potential risk to gain full privileges access of the system .... '
  describe sshd_config do
    its('PermitRootLogin') { should match(/no|without-password/) }
  end
end

                                ******* Chef Inspec: Ruby (Observed) [ 7 Lines] *******

control "xccdf_org.cisecurity.benchmarks_rule_9.3.8_Disable_SSH_Root_Login" do
  title "Disable SSH Root Login"
  desc  "The PermitRootLogin parameter specifies if the root user can log in using ssh(1). The default is no."
  impact 1.0
  describe bash("grep '^\\s*PermitRootLogin\\s' /etc/ssh/sshd_config
                | head -1").stdout.to_s.[](/\s*\S+\s+(.+?)\s*(#.*)?$/, 1) do
    it { should eq "no" }
  end
end
```

it might involve a learning curve for writing (or extending) an entity-specific Augeas parser 'lens'.

In our experience, the configuration definition style for different entities introduces a tradeoff with respect to parsing ease. It might be trivial to parse a more descriptive but semingly tedious configuration style, as in `sysctl.conf`, as compared to a more modular style as in `apache2.conf` but one that makes it non-trivial to programmatically infer relationships between configuration parameters and sections.

Developers of an application are best suited to define what constitutes a 'correct' or 'secure' configuration of an application. Our hope is that one day, all applications will ship with their configuration profiles possibly defined in CVL or equivalent, which will it significantly ease the effort in validating their configurations.

7 Conclusion

We presented our ConfigValidator approach for seamless configuration validation across heterogeneous entities - applications, systems and the cloud. We highlighted the key attributes of our declarative Configuration Validation Language to assist users - specialists and non-experts alike - in easy encoding, reasoning, and maintenance of configuration checklists. We compared ConfigValidator with existing approaches and demonstrated its efficiency both in execution time and rule specification. While ConfigValidator has been assisting the users of our cloud service in security validation of their containers, we are also working towards opensourcing it to the broader community. Work is under progress to increase ConfigValidator's coverage in terms of supported entities, and we hope to leverage the community support once the opensourcing process completes.

References

[1] 2017. Agentless System Crawler. https://developer.ibm.com/open/openprojects/agentless-system-crawler/. (2017).
[2] 2017. Ansible. https://www.ansible.com/. (2017).
[3] 2017. Apache HTTP server. https://httpd.apache.org/. (2017).
[4] 2017. Augeas tool. http://augeas.net/. (2017).
[5] 2017. Center for Internet Security (CIS) Security Benchmarks. https://benchmarks.cisecurity.org/. (2017).
[6] 2017. Chef InSpec: Compliance as Code. https://www.chef.io/inspec/. (2017).
[7] 2017. Chef Tool. https://www.chef.io/. (2017).
[8] 2017. MySQL server. https://www.mysql.com/. (2017).
[9] 2017. NGINX web server. http://nginx.org/. (2017).
[10] 2017. Puppet. https://puppet.com/. (2017).
[11] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis.. In *OSDI*, Vol. 10. 1–14.
[12] Salman Baset. 2016. Identifying insecure configurations with IBM Vulnerability Advisor. https://ibm.co/2waEFMm. (2016).
[13] United Compliance. 2017. STIG Viewer: The system must not permit root logins using remote access programs such as ssh. https://www.stigviewer.com/stig/red_hat_enterprise_linux_6/2013-02-05/finding/RHEL-06-000237. (2017).
[14] Docker. 2017. Docker Bench for Security. https://github.com/docker/docker-bench-security. (2017).
[15] Center for Internet Security. 2017. CIS Benchmarks. https://www.cisecurity.org/cis-benchmarks/. (2017).
[16] GovReady. 2017. Toolkit for getting open source apps ready for secure, approved government use. https://github.com/GovReady/govready. (2017).
[17] Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. 2015. ConfValley: A Systematic Configuration Validation Framework for Cloud Services. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 19, 16 pages. DOI:https://doi.org/10.1145/2741948.2741963
[18] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 140–151.
[19] NIST. 2017. The Extensible Configuration Checklist Format (XCCDF). https://scap.nist.gov/specifications/xccdf/. (2017).
[20] OpenSCAP. 2017. Security Compliance. https://www.open-scap.org/features/security-compliance/. (2017).
[21] Ariel Rabkin and Randy Katz. 2011. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 131–140.
[22] Alex Sherman, Philip A Lisiecki, Andy Berkheimer, and Joel Wein. 2005. ACMS: The Akamai configuration management system. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 245–258.
[23] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. 2007. AutoBash: improving configuration management with operating system causality analysis. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 237–250.
[24] S Suneja, C Isci, R Koller, and E de Lara. 2016. Touchless and always-on cloud analytics as a service. *IBM Journal of Research and Development* 60, 2-3 (2016), 11–1.
[25] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic configuration management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 328–343.
[26] John Toman and Dan Grossman. 2016. Staccato: A Bug Finder for Dynamic Configuration Updates. In *30th European Conference on Object-Oriented Programming, ECOOP 2016*. 24:1–24:25.
[27] Helen J Wang, John C Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. 2004. Automatic Misconfiguration Troubleshooting with PeerPressure.. In *OSDI*, Vol. 4. 245–257.
[28] Andrew Whitaker, Richard S Cox, and Steven D Gribble. 2004. Configuration Debugging as Search: Finding the Needle in the Haystack.. In *OSDI*, Vol. 4. 6–6.
[29] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 244–259.
[30] Tianyin Xu and Yuanyuan Zhou. 2015. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Comput. Surv.* 47, 4, Article 70 (July 2015), 41 pages. DOI:https://doi.org/10.1145/2791577
[31] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 687–700. DOI:https://doi.org/10.1145/2541940.2541983
[32] Sai Zhang and Michael D Ernst. 2013. Automated diagnosis of software configuration errors. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 312–321.