Hardening with Scapolite: A DevOps-based Approach for Improved Authoring and Testing of Security-Configuration Guides in Large-Scale Organizations

Patrick Stöckle
patrick.stoeckle@tum.de
Technical University of Munich (TUM)
Munich, Germany

Ionuț Pruteanu
ionut.pruteanu@siemens.com
Siemens AG
Bucharest, Romania

Bernd Grobauer
bernd.grobauer@siemens.com
Siemens AG
Munich, Germany

Alexander Pretschner
alexander.pretschner@tum.de
Technical University of Munich (TUM)
Munich, Germany

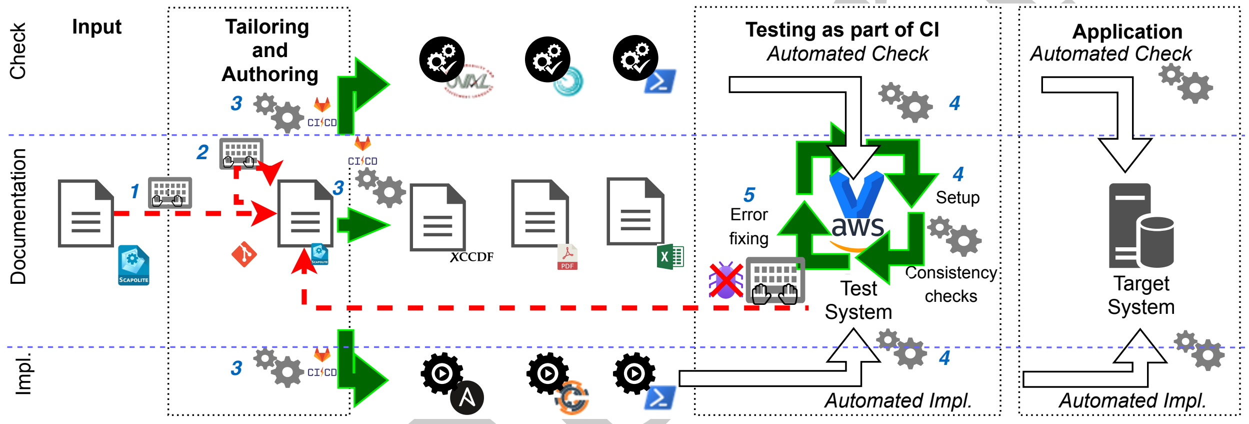

Figure 1: Improved process of security hardening. The green arrows represent activities that have been automated.

ABSTRACT

TOOL PAPER[1] Security Hardening is the process of configuring IT systems to ensure the security of the systems' components and data they process or store. In many cases, so-called security-configuration guides are used as a basis for security hardening. These guides describe secure configuration settings for components such as operating systems and standard applications. Rigorous testing of security-configuration guides and automated mechanisms for their implementation and validation are necessary since erroneous implementations or checks of hardening guides may severely impact systems' security and functionality. At Siemens, centrally maintained security-configuration guides carry machine-readable information specifying both the implementation and validation of each required configuration step. The guides are maintained within *git* repositories; automated pipelines generate the artifacts for implementation and checking, e.g., PowerShell scripts for Windows, and carry out testing of these artifacts on AWS images. This paper describes our experiences with our DevOps-inspired approach for authoring, maintaining, and testing security-configuration guides. We want to share these experiences to help other organizations with their security hardening and increase their systems' security.

[1]We submitted this article as a full-length paper. One can find the full-length version here [6]. We collected all code example we cut out here [4].

CODASPY '22, April 24–27, 2022, Baltimore, MD, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9220-4/22/04…$15.00
https://doi.org/10.1145/3508398.3511525

CCS CONCEPTS

• **Security and privacy** → *Usability in security and privacy*; *Software security engineering*.

KEYWORDS

Hardening, Security Configuration

ACM Reference Format:
Patrick Stöckle, Ionuț Pruteanu, Bernd Grobauer, and Alexander Pretschner. 2022. Hardening with Scapolite: A DevOps-based Approach for Improved Authoring and Testing of Security-Configuration Guides in Large-Scale Organizations. In *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy (CODASPY '22), April 24–27, 2022, Baltimore, MD, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3508398.3511525

1 INTRODUCTION

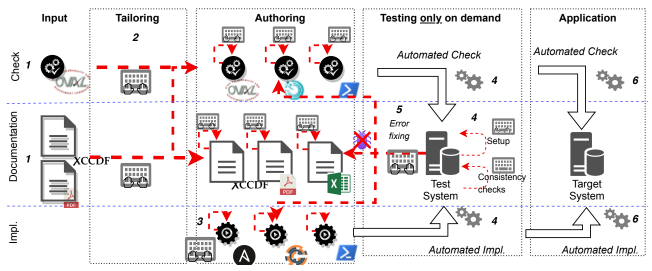

Figure 2: Typical process of security hardening. Dotted arrows represent manual tasks. Every arrow within the box is a task the administrators execute to harden the system.

Insecure configurations of operating systems and applications are known to be both common and detrimental to cybersecurity [1–3, 7, 9]. Organizations, therefore, need to identify the security-relevant configuration settings of the used software, determine the secure value or a set of secure values for each setting, and ensure that they configure each instance of the software used within their organization accordingly. This process is called security-configuration hardening and is part of the general security hardening of an organization's infrastructure. Security hardening is a continuous process rather than an one-time-only task since the IT infrastructure, the threat environment, insights about (in)secure configurations, et cetera are constantly in flux.

Organizations such as the Center for Internet Security (CIS) or the Defense Information Systems Agency (DISA) provide publicly available security-configuration guides (also called benchmarks, guidelines, or baselines) for various software components, e.g., operating systems like Windows 10, web servers like NGINX, or email clients like Outlook. These guides consist of rules, and each rule states which values should be used for a configuration setting relevant for security; some of these guides consist of more than 350 rules. Benchmarks written in the SCAP [8] standard often contain machine-readable definitions of *checks*, whereas mechanisms for *implementing* the required settings are usually either provided separately or not at all. The usual security-configuration hardening process, which is based on such public guides, contains many manual steps that are both inefficient and error-prone. Most of the time, one need to adapt the external guides for their target infrastructure by modifying specific settings, removing some rules, and adding others. This problem is intensified by the fact that these adaptions have to be replicated and kept consistent for each implementation, such as scripts (e.g., Bash), Infrastructure as Code (IaC) approaches (e.g., Ansible), et cetera, and for each check mechanism.

1.1 Problems of the Current Security Hardening Process

Figure 2 illustrates the usual security hardening process; the numbers in the figure refer to the following steps:

(1) Input is an external guide in the SCAP standard: The human-readable parts are defined in the *eXtensible Configuration Checklist Format* (XCCDF) with machine-readable checks in the *Open Vulnerability and Assessment Language* (OVAL).
(2) XCCDF offers a mechanism for tailoring the guide, e.g., configure changes via so-called profiles. The profiles are also reflected in the OVAL-based checks.
(3) Because machine-readable implementation mechanisms are not part of these guides (exception: ComplianceAsCode, discussed below), one must either manually develop implementation mechanisms or adjust them if one can re-use existing mechanisms. Since larger organizations may use several different implementation mechanisms, one may need to re-apply the same changes numerous times.
(4) Before applying the implementation mechanisms to and using the check mechanisms for production systems, one must test both of them: Erroneous implementation/checking of security configurations may severely impact the security and functionality of systems. Because the guides are used for many target systems (different operating systems and applications, et cetera), one must manage a corresponding multitude of test systems.
(5) Feedback about problems, e.g., faulty implementations or checks, might introduce changes for one or several implementation/check mechanisms.
(6) Finally, the tailored and tested security guides can be applied to production systems. If problems are detected in productive use or a new version of a guide is published, the whole process restarts.

Repeating these **manual** steps increases the risk of introducing **errors** and, thus, of **insecure systems**. Therefore, we identified the following *challenges* for improving the security hardening:

- Remove superfluous complexity in the security hardening process resulting from unnecessary manual steps and scattered information.
- Establish automatic quality assurance for the guides to find errors earlier and easier.

1.2 Our Approach: Improved Authoring, Artifact Generation, and Automated Testing

Our solution to these challenges is twofold. We present our improved configuration hardening approach that focuses on automation to remove error-prone manual steps. Next, we illustrate our approach to automatic testing guides to detect errors as soon as possible. Figure 1 shows our improved security hardening process; again, the numbers refer to the steps below:

(1) We manage guides in a dedicated YAML-based format called *Scapolite*, which we keep under version control. Further, we enrich the format with machine-readable information about configuration requirements. Ideally, both implementation and check

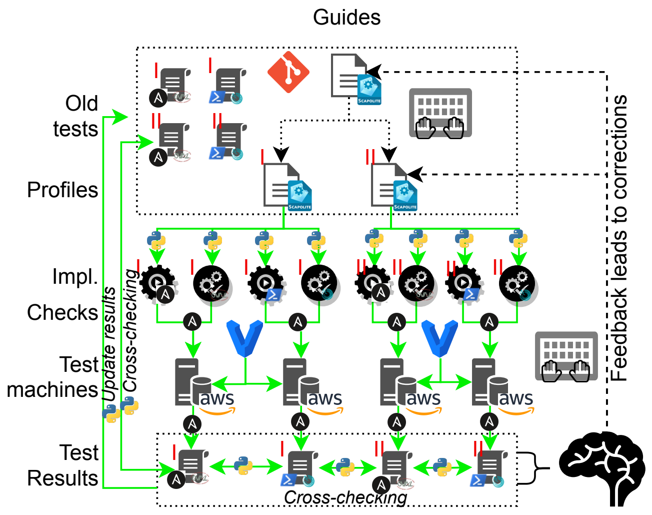

Figure 3: State-of-the-art execution of tests in a security hardening process. The green arrows denote steps that are now automated.

mechanisms can be automatically derived. Thus, we keep information about the check, implementation, metadata, and documentation, e.g., human-readable descriptions about the requirements, the rationale, et cetera, at a single location.[2]

(2) Tailoring to different use-cases in Scapolite works similarly as in SCAP: We can define profiles for the individual use cases and create per-use-case modifications.
(3) From this single source, i.e., the machine-readable information from 1), we automatically generate the required artifacts for implementing/checking the guides.
(4) Creation of the required test systems as VMs, applying the implementations/checks to these systems, and collecting the test results is carried out automatically as a part of a DevOps pipeline.
(5) By automatically generating the implementations/checks, we can fix detected problems with a single change in the Scapolite document defining the guide or a bug-fix in the transformation system, rather than changing in several different artifacts.

1.3 Contributions

Our contributions to the field of security hardening are:

- By pulling information required for generating both implementation and check mechanisms as machine-readable information into our security-configuration guides, we manage to restrict manual changes/corrections to a single location, thus reducing errors and increasing efficiency.
- We show how to operate a DevOps/CI-inspired approach of authoring and maintaining security-configuration guides. In our approach, changes in the guides trigger automated tests without human involvement in the execution of the tests, collection of test results, and correlation of test data with expected results.

The latter point deserves a closer examination: As explained above, security-configuration mechanisms are affected by the combinatorial explosion of test cases, requiring many test systems and test runs. Figure 2 illustrates the approach without the DevOps: a single test already requires a substantial manual effort that must be multiplied by the number of test systems/test cases; when we detect problems, we have to fix them at several locations. In contrast, Figure 3 illustrates the level of automation of our approach.

Our experiences of handling multiple guides with multiple profiles authored/maintained using version control and DevOps pipelines within an industrial context show that an approach that combines machine-readable information required for implementing and checking security-configuration requirements is not only feasible but provides enormous benefits. Errors are reduced, and the efficiency and the effectiveness of an organization's security-configuration hardening program are raised. Thus, we tackle two of the major causes for insecure configurations: erroneous application and ineffective or incomplete application of secure configurations.

2 OUR APPROACH TO SECURITY HARDENING

Everyone who published guides to their organization knows the need for automated implementation and validation of the settings. Especially in the case of operating systems, publishing a guide without providing automated mechanisms is inefficient and ineffective:

- multiple persons/groups in the constituency work in parallel on creating implementation/validation mechanisms;
- the manual transcription of required settings into an implementation mechanism will lead to errors and omissions;
- some constituency members will deem the task of implementation as too arduous or time-consuming and not bother with it at all.

The SCAP [8] format family defines the state of the art for providing automated mechanisms along with a guide. We can use the SCAP formats to augment human-readable information with machine-readable checks, usually specified in OVAL. In almost all cases, however, automated implementation mechanisms are maintained separately, except for the *ComplianceAsCode* [3]. At Siemens, we strive to operate at a higher level of abstraction – where possible – by specifying the desired configurations in a machine-readable form from which we can derive implementation and verification mechanisms. We combine this with a rigorous "DevOps"-approach for authoring and maintaining guides: we use pipelines for both automated derivation and test of implementation and validation mechanisms.

2.1 The Scapolite Format

We started by defining a format called "Scapolite," which encompasses SCAP's relevant features but additionally provides

(1) a form that can be created/maintained as text-files under version control (cf. above comment on changes in rules).
(2) generalizations and additional extension points to support a broader range of use cases.

[2]External guides in SCAP can be automatically converted into Scapolite. Adding machine-readable information from which implementations and checks can be derived requires, of course, manual effort, but such effort would be necessary for generating separate implementation mechanism, as well. Furthermore, we developed semi-automated mechanisms for deriving machine-readable information from human-readable text [5].

[3]https://github.com/ComplianceAsCode/content

```
---
scapolite:
    class: rule
    version: '0.51'
id: BL942-1101
id_namespace: org.scapolite.example
title: Configure use of passwords for removable data drives
rule: <see below>
implementations:
  - relative_id: '01'
    description: <see below>
history:
  - version: '1.0'
    action: created
    description: Added so as to mitigate risk SR-2018-0144.
---
## /rule
Enable the setting 'Configure use of passwords for removable
data drives' and set the options as follows:
   *  Select `Require password complexity`
   *  Set the option 'Minimum password length for removable data drive` to `15`.
## /implementations/0/description
To set the protection level to the desired state, enable the policy
`Computer Configuration\...\Configure use of passwords for removable data drives`
and set the options as specified above in the rule.
```

Listing 1: A very basic example of a rule in Scapolite. Lines referenced in the text are marked in blue.

(3) fields for tracking of document maintenance data such as change history information per configuration requirement.

Listing 1 shows a minimal example of Scapolite; highlighted lines contain the description of how to implement the required setting.

2.2 Adding Machine-Readable Automations

The setting prescribed by the example rule in Listing 1 concerns a Windows policy setting, specified via a path and the required *policy value*. We, therefore, augment the Scapolite rule object shown in listing 1 with an *automation* structure. For a programmatic implementation, however, an intermediate step is necessary. We have, therefore, implemented an automated transformation of the policy-based specification to a registry-based automation; similar transformations exist for other "low-level" mechanisms required.

2.3 Transforming Automations

The result of carrying out this transformation is that we must set three registry keys; the first key signifies that the setting is enabled; the second specifies that the requirements on password complexity are active; the third contains the minimum password length. Ideally, we specify all security requirements as abstractly as possible and transform them automatically into mechanisms for implementation and checking. However, if we cannot find a suitable abstraction level, we must include code in a scripting language.

2.4 Producing Code and Other Artifacts

With the machine-readable specifications of what needs to be implemented/checked and the associated transformation mechanisms, we can generate artifacts that the system administrators can use to carry out the rule's implementation and check. For guides targeting Windows, we generate a set of PowerShell commandlets together with a JSON file containing the necessary data used to implement or check the corresponding rule. We can also transform our structure in OVAL. Before the scripts implement a rule, they store each setting's current value to enable rollbacks.

Our approach to security hardening has two main advantages: It concentrates all information of a rule in one place and reduces the risk of inconsistencies, and the transformations replace many manual steps and significantly reduce the risk of errors.

3 OUR APPROACH TO TESTING

3.1 Maintenance and Release Process

Our workflow in authoring, maintaining, and releasing security-configuration baselines is as follows:

(1) Authors write guides using Scapolite. The Scapolite files are kept under version control at an internal *GitLab* instance.
(2) We use GitLab pipelines to automatically transform the machine-readable automations into artifacts for implementation and check.
(3) Once we release a guide, Siemens's security-regulation portal *SFeRA* generates human-readable versions (web view, PDF, etc.) directly from the Scapolite sources.
(4) The pipeline-based transformation mechanism is triggered for the released version of the Scapolite sources, and we provide the artifacts to users via dedicated repositories.

In a parallel process, we maintain the technological basis of this process and develop it further, namely:

(1) libraries for creating and manipulating Scapolite content, e.g., imports from SCAP, methods for enriching existing Scapolite content with additional information, et cetera;
(2) libraries for transforming abstract machine-readable automations into more concrete automations (cf. Section 2.3);
(3) libraries for further transformation into code or other artifacts (cf. Section 2.4)

3.2 Test Requirements during Guide Creation

Creating a guide is an iterative process between writing and testing its implementation. The author, therefore, requires a test environment in the form of a virtual image.

Manual creation/maintenance of such a test environment, as well as the manual execution of the tests, is a tremendous overhead: we must start/reset the virtual image, generate the artifacts, transfer them to the image, and execute the artifacts; usually, we execute this process several times for implementing and checking rules for different use-cases. In the end, we must collect the test results and prepare them for the manual analysis. Efficient guide creation, therefore, is impossible without automated testing.

3.3 Test Requirements During Guide Maintenance

Automated testing also is essential during maintenance. Every change either in the Scapolite source or the underlying infrastructure required for generating the artifacts for implementation and checking may lead to errors. For example:

(1) Errors in the metadata introduced during maintenance may lead to rule omissions in the generated artifacts.
(2) Errors in the transformation from abstract to concrete information, e.g., through bugs in the library, may lead to faulty specifications respectively implementations/checks.
(3) Similarly, errors in the transformation to program code or other artifacts may lead to faulty implementations/checks.

Further, we need to detect errors in a timely manner that are introduced by changes that have nothing to do with our process:

1. Maintainers may misspecify the machine-readable information when making changes during maintenance.
2. Changes in the target, e.g., upgrades of the OS, may invalidate or break a particular way of implementing or checking.
3. Changes in execution environments for a created artifact, e.g., changes in a vulnerability scanner we generate a specification for, may invalidate the created artifact.

Only a high automation degree allows us to run the required regression tests whenever a change occurs.

3.4 The Testing process

Testing a guide's implementation and checking on a target is likely to require several test runs: one for each combination of use-case, target-system revision, and implementation or check runtime environments. We use the CIS-CAT scanner to verify implementations/checks generated for CIS baselines. Nevertheless, we can also have different results for the same tools, e.g., because of different versions. A test run typically has the following shape:

Run initial checks Run checks on the unchanged system to establish the status quo *before* the implementation.
Apply security settings Execute the generated mechanism for implementing the desired security settings.
Carry out checks for compliance Re-run checks.
Revert settings Reset the revertable settings to their initial values.
Check reverted settings Check the status after we restored the settings' old state.

3.4.1 Analysis of a test run. Relevant data that can be collected from such test runs are:

Quantitative data How many rules were successfully applied? For how many rules did the check return a success, a failure, or a runtime problem?
Detailed information Which rules were successfully applied? For which rules was the check successful, a failure, ran into a problem, et cetera?

Analysis of the complete set of test runs for a specific setting, i.e., a combination of use-case and target system, usually entails two types of comparison:

Comparisons within a test run to find discrepancies, e.g.:

- A rule is reported as applied, but the check mechanism reports the rule as non-compliant.
- Two check mechanisms report different results for a rule.
- The check mechanism marked a rule as non-compliant before the implementation, compliant after the implementation, but still as compliant after the reverting.

Comparison with previous test runs : as regression tests, the newly collected data are compared with data from previous test executions. Were there changes? If so, are these *desirable* changes, e.g., we improved an implementation or check that did not work before, or *undesireable* changes, e.g., unsuccessful check.

3.5 Our Approach to Test Automation

In order to automate testing as much as possible, we implemented the following approach: Our tooling executes a machine-readable test specification on VMs created on-demand in AWS; the tooling carries out the specified test activities, collects the raw data generated from implementation/check mechanisms, and automatically prepares summary data and comparisons for the test analysis.

This complete automation of test activities allows an author or maintainer to carry out tests with no effort; the extensive preprocessing of the test data enables them to see directly whether there are deviations from the expected results and enables them to focus on analyzing the cause of these deviations.

3.5.1 Test Specification. We specified a YAML-based test file format to define one or more test runs; they are executed on different instances in parallel. We specify:

- for each test run, a sequence of activities such as implementing, checking, or reverting rules.
- for each activity, a list of so-called validations; each validation compiles data from the result or log files created by an activity (for example, validations can count successfully checked rules, collect these rules' identifiers, compare the current results to results of previous activities, et cetera);
- for each validation, the expected results (as basis for regression tests along with each validation)

The test specification file is kept under version control with the Scapolite sources for each guide.

3.5.2 Test Execution. We have implemented a test runner that is part of the DevOps pipeline that generates the artifacts for implementation and checks. The test runner accesses the test specification file in the repository and executes the tests:

- For each test run, the runner starts the required AWS image.
- The runner transfers the created artifacts and additional resources required for implementation/checking to the image.
- The runner uses Ansible to carry out the specified activities.
- In the end, the runner retrieves the created result/log files from each activity from the image, stops and destroys it.

3.5.3 Preprocessing of test results. As described in Section 3.5.1, we can specify validation tasks for each action carried out in the test run. Hence, after the runner collected all raw data, the tooling carries out the validation tasks: it compiles the required data and compares them to the expected results from the specification file.

As a final step, our tooling commits (1) a detailed log, (2) a report of found deviations, (3) an updated test specification file with the current validation results, and (4) all raw data retrieved from the image to a staging repository.

3.6 Execution of Tests

3.6.1 Testing in the cloud. We test our guides using AWS EC2[4] and integrated an S3 bucket into our architecture to transfer the data within the AWS data center rather than via the internet.

3.6.2 Integration into DevOps pipeline. We generate the artifacts for implementing and checking from the Scapolite sources with a DevOps pipeline maintained as a GitLab-CI *include file* within a dedicated repository. By factoring out the actual code for the pipeline,

[4]https://aws.amazon.com/ec2/

we (1) keep the project's CI file concise, and (2) can carry out the maintenance of the pipeline via the single pipeline repository.

Each test entails the creation of several virtual machines, and the execution of a test run may take up to an hour. We, therefore, chose to carry out only static tests for each push but require an active request by the author/maintainer for dynamic tests.

3.6.3 Dealing with negative effects of secure configurations on test execution. The infrastructure relies on specific mechanisms for accessing and manipulating the test VM. Usually, the guides' rules disable some of these mechanisms. If we implemented one of these rules, the following test activities would fail, with little or no information about why the activity failed. To facilitate finding those breaking rules, we implemented the following features:

- Tests can implement the rules in an *apply* activity one by one rather than in bulk; they can thus attribute a failure to the last successfully applied rule.
- Tests can configure the rule implementation to start at a specific rule or the last rule contained in the blacklist. Unless a combination causes an execution failure, this suffices to find all breaking rules.

3.7 User Feedback

We have highly automated the testing process, but the test analysis still requires human interaction. The analysts need to know whether something went wrong and they need easy access to the raw data for an in-depth analysis of problems uncovered by the test.

3.7.1 Summary Report. Our tooling generates a summary report providing concise information for each activity:

(1) Did failures occur during an activity, e.g., because a setting interrupted the connection to the VM?
(2) If no failure occurred, did the test yield the expected results as documented in the test specification file?
(3) Where possible: if the test yielded different results, did the test show an *improvement*? Were more rules implemented successfully than during the previous test run?

With item 3) we intend to provide the user with an initial assessment of the test results based on a formal definition of what constitutes an improvement/degradation the users can specify in the test specification file.

3.7.2 Documentation of full results. The users can access detailed information about found deviations for each validation step and see the raw data for each activity within a staging repository containing the generated artifacts. Also, they can use different mechanisms provided by *git* and *GitLab* such as viewing differences between test executions, e.g., within the generated artifacts.

3.7.3 Further automation. We provide further support to the users if they need to re-test several guides, e.g., when the transformation mechanism was updated. These command-line scripts that use the *GitLab* API include tasks like:

- starting pipelines in parallel for several guides;
- informing about the pipelines' status;
- compiling an overview with the results of all test pipelines;
- showing differences between the newly-generated artifacts and the latest published version for each guide;

By automating repetitive manual tasks carried out for each guide, we achieve that tests are executed frequently. Especially small or seemingly *harmless* changes are now more often tested because we lowered the effort for starting the tests and analyzing the test results for more than one guide significantly.

4 CONCLUSION

We developed an approach towards authoring and maintaining machine-readable security-configuration guides that extends the DevOps principle of Continuous Integration to this domain. We achieved this by creating the Scapolite format that enables authors to combine human-readable information with machine-readable information on security-configuration requirements. The latter serve as input for a process that automatically generates artifacts for implementation and checking, and tests the created artifacts. Because the authors specify the rules on an abstract level, we could significantly reduce the risk of errors because of manual errors.

Due to the high degree of automation in our process, we test the guides and their generated artifacts much more frequently during authoring and maintenance than in the usual case. As a result, we detect the majority of problems before the release of a guide.

In summary, our approach to security hardening via machine-readable security-configuration guides combined with automated testing allows us to publish automated, well-tested mechanisms for implementing and checking along with the guide. Consequently, we can comply with these configurations more quickly and less error-prone, leading to better-secured systems.

REFERENCES

[1] Andrea Continella, Mario Polino, Marcello Pogliani, and Stefano Zanero. 2018. There's a Hole in That Bucket!: A Large-scale Analysis of Misconfigured S3 Buckets. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. ACM, New York, NY, USA, 702–711. https://doi.org/10.1145/3274694.3274736
[2] Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. 2018. Investigating System Operators' Perspective on Security Misconfigurations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 1272–1289. https://doi.org/10.1145/3243734.3243794
[3] A. K. Jha, S. Lee, and W. J. Lee. 2017. Developer Mistakes in Writing Android Manifests: An Empirical Study of Configuration Errors. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 25–36. https://doi.org/10.1109/MSR.2017.41
[4] Patrick Stöckle. 2022. *Hardening with Scapolite: Code Examples.* github.com/tum-i4/CODASPY2022.
[5] Patrick Stöckle, Bernd Grobauer, and Alexander Pretschner. 2020. Automated Implementation of Windows-Related Security-Configuration Guides. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 598–610. https://doi.org/10.1145/3324884.3416540
[6] Patrick Stöckle, Ionuț Pruteanu, Bernd Grobauer, and Alexander Pretschner. 2022. Hardening with Scapolite: Original Version. https://i4.pages.gitlab.lrz.de/conferences-public/preprints/2022/CODASPY/hardening-with-scapolite.pdf
[7] 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* (Monterey, California) *(SOSP '15)*. ACM, New York, NY, USA, 328–343. https://doi.org/10.1145/2815400.2815401
[8] David Waltermire, Stephen Quinn, Harold Booth, Karen Scarfone, and Dragos Prisaca. 2018. The Technical Specification for the Security Content Automation Protocol (SCAP) Version 1.3. https://doi.org/10.6028/NIST.SP.800-126r3
[9] 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. https://doi.org/10.1145/2791577