Leveraging Practitioners' Feedback to Improve a Security Linter

Sofia Reis
IST, University of Lisbon & INESC-ID
Lisbon, Portugal
sofia.o.reis@tecnico.ulisboa.pt

Rui Abreu
FEUP & INESC-ID
Porto, Portugal
rui@computer.org

Marcelo d'Amorim
Federal University of Pernambuco
Pernambuco, Brazil
damorim@cin.ufpe.br

Daniel Fortunato
FEUP
Porto, Portugal
daniel.b.fortunato@tecnico.ulisboa.pt

ABSTRACT

Infrastructure-as-Code (IaC) is a technology that enables the management and distribution of infrastructure through code instead of manual processes. In 2020, Palo Alto Network's Unit 42 announced the discovery of over 199K vulnerable IaC templates through their "Cloud Threat" Report. This report highlights the importance of tools to prevent vulnerabilities from reaching production. Unfortunately, we observed through a comprehensive study that a security linter for IaC scripts is not reliable yet—high false positive rates. Our approach to tackling this problem was to leverage community expertise to improve the precision of this tool. More precisely, we interviewed professional developers to collect their feedback on the root causes of imprecision of the state-of-the-art security linter for Puppet. From that feedback, we developed a linter adjusting 7 rules of an existing linter ruleset and adding 3 new rules. We conducted a new study with 131 practitioners, which helped us improve the tool's precision significantly and achieve a final precision of 83%. An important takeaway from this paper is that obtaining professional feedback is fundamental to improving the rules' precision and extending the rulesets, which is critical for the usefulness and adoption of lightweight tools, such as IaC security linters.

CCS CONCEPTS

• **Security and privacy → Vulnerability scanners**.

KEYWORDS

Security, Infrastructure, Linter

ACM Reference Format:
Sofia Reis, Rui Abreu, Marcelo d'Amorim, and Daniel Fortunato. 2022. Leveraging Practitioners' Feedback to Improve a Security Linter. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3560419

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3560419

1 INTRODUCTION

Software configuration management and deployment tools like Puppet[1], Ansible[2], and Chef[3] became popular amongst software development warehouses [18, 29]. The adoption of these tools has increased with the growing movement to run software in cloud servers. These tools help infrastructure teams increase productivity by automating various configuration tasks (e.g., server setup). In short, these tools describe the environment configuration in a set of provisioning scripts that can be versioned and reused. They enable configuration consistency between different environments and can reduce the time required to provision and scale the infrastructure. The process of managing and provisioning infrastructure through configuration scripts is called Infrastructure-as-Code (IaC). IaC tools take a script as input and create an infrastructure that typically runs in a virtual environment as output.

As with any piece of code, IaC scripts are also prone to defects such as security vulnerabilities [28]. For example, in 2020, Palo Alto Network researchers reported the discovery of over 199K vulnerable IaC templates [1]. Specifically, 42% of AWS CloudFormation templates, 22% of Terraform templates, and 9% of Google Kubernetes YAML files were vulnerable. In addition, researchers found more than 67k *potential* security smells in IaC scripts implemented in Ansible, Chef, and Puppet [32] through an ad-hoc tool created to show the presence of a new set of anti-patterns for security in the IaC domain. These reports highlight the importance of tools to prevent vulnerabilities from reaching production and shift security left in the development pipeline.

Figure 1 shows an example of an *Admin by default* weakness (CWE-250[4]), a potential vulnerability in a Puppet manifest in a module of the `PuppetLabs`.[5] The vulnerability manifests when the developer configures a user as "admin" or "root" for an infrastructure component. In this example, the `$grafana_user` is set as "admin" for the different services (Puppetserver, Puppetdb, Postgresql, Filesync) used by Grafana[6]. Therefore, any service can be prone to a privilege escalation attack. In IaC, all the infrastructure

[1] https://puppet.com/
[2] https://www.ansible.com/
[3] https://www.chef.io/
[4] CWE-250 details available at https://cwe.mitre.org/data/definitions/250.html
[5] Admin by default example available at https://github.com/puppetlabs/puppet_operational_dashboards/blob/9eb67a407aa44c2f924f67f207edc7032f81f86a/manifests/profile/dashboards.pp#L137 (Accessed October 13, 2022)
[6] Grafana is an open-source software application for data exploration and visualization. More information available at https://grafana.com/

Figure 1: Simplified example of an *Admin by default*.

components are configured through scripts, including the access credentials. Specifying default users as administrative gives privileges to users that only an administrator should have. Admin accounts can be exploited to access sensitive data and execute unauthorized code/commands. Infrastructure engineers should avoid setting admin passwords and usernames to user accounts that do not need the privileges. Detecting these issues automatically essential to make infrastructure engineers aware of possible security problems and to help companies build more robust infrastructures.

Problem: Puppet IaC Security Linters are not reliable yet! In 2019, Rahman et al. showed that IaC scripts—just like any piece of code—are not immune to security vulnerabilities [28, 30]. They focused on Puppet configuration files and listed seven anti-patterns that could lead to security vulnerabilities. The work led to the development of SLIC, a linter to detect those defects in Puppet scripts. Linters are often imprecise tools [4, 8, 12, 19, 22, 34]. Therefore, motivated by the report of very high accuracy (i.e., precision and recall) from their paper (Table IV [28]), we decided to conduct a reproduction study of SLIC based on a different and larger set of projects. We asked students (co-authors of this paper) and developers to analyze the warnings that the tool reports. To validate the students observations of low precision, we reported a sample of the warnings of the tool to maintainers of 86 open-source projects. From the 228 issues created, we obtained responses to 51 issues where only 33 issues were discussed or clear. Results showed that the tool performs differently in a new set of projects, particularly when validated by the software owners. The precision observed was smaller than the one reported in SLIC's original paper (28% instead of 99%) which indicates that security IaC linters for Puppet are not reliable yet due to the high false positive rates.

Like many linters, SLIC uses simple rules to detect issues. Essentially, it searches for string patterns in the values of tokens (many times) regardless of their type (e.g., variable, string, etc.) and the relationship between them. For instance, the "Usage of Crypto. Algorithms" checker (CWE-326[7]) searches for any token whose value includes `sha1` or `md5`. Both are built-in Puppet functions and SLIC fails to consider the context of usage of these algorithms, i.e., these functions are called in Puppet manifests to encrypt data (e.g., $encrypt\_key = md5(\texttt{key})$). Therefore, SLIC incorrectly detects `md5checksum = '07bd73571b7028b73fc8ed19bc85226d'` as a CWE-326. This simplicity creates much noise for developers. In this preliminary study, we observed that the rules for the current IaC security anti-patterns must be better designed to be safely adopted by the industry and avoid productivity disruption.

Solution: Our preliminary study revealed that (1) there is a need to improve the precision of IaC security linters for Puppet, and that (2) security tools can be iteratively improved and extended by incorporating feedback from the developer community as suggested in previous work [34]. This paper reports on the process we followed to iteratively and incrementally improve the precision of an IaC linter according to user feedback. For example, the experiments

[7]CWE-326 details are available at https://cwe.mitre.org/data/definitions/326.html

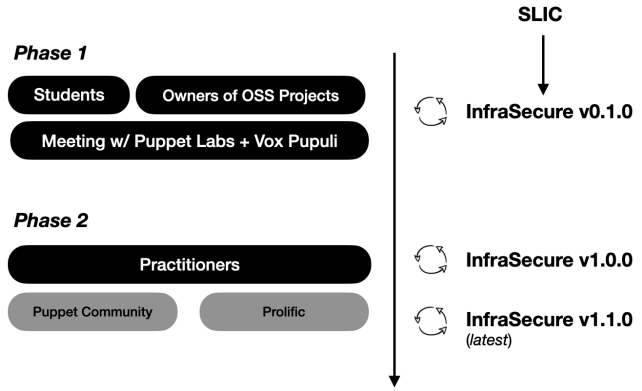

Figure 2: Timeline of feedback collection.

described above ignited discussions with members of the development and security teams of Puppetlabs[8], as well as one project manager from Vox Pupuli[9]. The feedback collected from the team, OSS maintainers and the Puppet community led to the creation of a new tool, which we dubbed as InfraSecure. Later, we leveraged the expertise of practitioners experienced in IaC tools or security to iteratively and incrementally improve the new tool.

Figure 2 shows the timeline of feedback collection followed to design and improve InfraSecure. To sum up, we bootstrapped the design of InfraSecure with rules obtained from the revision of SLIC's ruleset, according to the feedback of the research team and owners of OSS projects (*phase 1* in Figure 2); and, incrementally evolved the linter according to the recommendations of practitioners (*phase 2* in Figure 2). We improved 7 rules of the SLIC ruleset and added 3 new rules that were either recommended by practitioners (e.g. weak password); or relevant for the infrastructure domain (e.g. homograph attacks[10] and malicious dependencies).

Main Results: This paper performs the following contributions:

- ★ **Study.** A replication study of `SLIC`'s precision, including a preliminary study conducted with two researchers (co-authors of this paper) and a study with several GitHub scripts validated by project maintainers;
- ★ **InfraSecure.** A new linter adjusting 7 rules of the original SLIC ruleset and adding 3 new rules with a final precision of 83%.
- ★ **Dataset.** A dataset of IaC scripts with more than 1000 warnings classified as false positives and true positives that researchers can use to evaluate other security linters;

Take-away message: The takeaway message of this paper is that it is feasible to tune security linters to produce acceptable precision for important classes of warnings (confirming the findings reported in a study at Google [34]) and that involving practitioners in discussions is an effective way to guide the improvement of those linters.

Replication Package: All the scripts and data used in this study (including feedback obtained from the maintainers and practitioners) are available at: https://figshare.com/s/6b6a769b1393eae0774c.

[8]GitHub PuppetLabs organization website: https://github.com/puppetlabs

[9]Vox Pupuli is the organization responsible for maintaining modules and tools for the Puppet community: https://voxpupuli.org/

[10]Apple Domain Attack (2017): https://www.xudongz.com/blog/2017/idn-phishing/

Table 1: Examples of security smells per weakness.

CWE	Example
CWE-798	$username = "mariadb"
CWE-259	$password = "!TQ23Rg"
CWE-321	$key = "A67ANBD7"
CWE-319	$req = "http://www.domain.org/secret"
CWE-546	#https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=538392
CWE-326	password => md5($debian_password)
CWE-284	$bind_host = "0.0.0.0"
CWE-258	$rabbitmq_pwd = ""
CWE-250	$user = "admin"
CWE-521	pwd => "12345"
CWE-1007	$source = "http://deb.debi*a*n.org/debi*a*n"
CWE-829	$postgresql_version = 8.4

2 BACKGROUND

This section provides background information on Puppet and discusses security weaknesses in IaC scripts.

2.1 Puppet

Developers no longer deal with systems composed of a single machine and database. Instead, system administrators must manage multiple diverse operating systems, databases, and virtual machines. Perhaps most importantly, they must ensure their configurations are consistent at any given time. Configuration management technologies have been around for over a decade—Puppet was founded in 2005. Puppet is a solution that helps IT infrastructure management through code by supporting software deployment, packages, and configuration. In Puppet, programs are written using Puppet's declarative language. They are called manifests. More information about the language can be found elsewhere[11].

2.2 Security Weaknesses

This section describes in more detail the potential weaknesses in Puppet scripts. Table 1 illustrates examples of each weakness.

Hard-coded secrets (CWE-798): This warning refers to the practice of including sensitive information such as passwords or cryptographic keys in code files. Table 1 shows a hard-coded password example. Rahman et al. considered 3 kinds of data as sensitive: usernames, passwords, and cryptographic keys. However, the Common Weakness Enumeration[12] list does not consider solo hard-coded usernames as a threat. Practitioners involved in our validation studies shared the same opinion. Therefore, we argue that hard-coded usernames should be only reported when there is a paired password (CWE-259) or cryptographic key (CWE-321). More discussion on this is provided later in Section 4.2.

Use of HTTP without TLS (CWE-319): This warning refers to the practice of using HTTP without the Transport Layer Security (TLS) to transmit sensitive data. This means that attackers can more efficiently exploit the communication channel as the data is transmitted unencrypted, as cleartext[13].

Suspicious comment (CWE-546): This warning refers to comments that suggest the presence of bugs, missing security functionalities, or weaknesses of a system. Details provided in comments about bugs, security functionalities or weaknesses can be crucial for hackers to exploit the infrastructure.

Use of weak cryptography algorithms (CWE-326): This warning refers to using weak crypto algorithms. Attackers can easily crack the encryption schemes and have access to the data [37].

Invalid IP address binding (CWE-284): This warning refers to assigning the address 0.0.0.0 to a server. This practice allows connections from any IP address to access that server [26]. While mail servers have to listen on 0.0.0.0 to receive mail, database servers should not since it can lead to critical data breaches.

Empty password (CWE-258): This warning refers to using empty strings as passwords, which are easily guessable.

Admin by default (CWE-250): As detailed in the Introduction, this warning refers to defining users with administrative privileges. This can result in security weaknesses since it can disable or bypass security checks performed by the system.[14]

Weak Password (CWE-521): This warning refers to the usage of weak passwords. Weak passwords are easily guessable and can be bypassed to gain access to systems.

Insufficient Visual Distinction of Homoglyphs Presented to User (CWE-1007): This warning refers to malicious actors using homoglyphs that may cause the user to misinterpret a glyph and perform an unintended, insecure action. The homograph attack performed against the apple website[15] is a well-known example of this type of weakness. Table 1 shows an example of a domain where the character "*a*" could be replaced by the respective homoglyph, as in the apple attack. This warning might be essential to uncover malicious domains implanted by malicious open-source contributors—typosquatting attacks.[16]

Malicious Dependencies (CWE-829): This warning refers to malicious software by nature, i.e., dependencies that integrate known vulnerabilities (CVEs). These are the leading cause of supply chain attacks, and one of the main challenges the security community faces nowadays [15].

3 PRELIMINARY STUDY

This section reports on the findings of two studies—involving different sets of participants—to assess the performance of SLIC, a recently-developed linter for Puppet.

3.1 Validation with Students

This section reports on a study involving two of this paper's authors to assess the precision of SLIC on an independent benchmark. The study consisted in inspecting 502 warnings reported by the tool. The warnings were validated by one senior PhD student whose research focuses on security and static analysis and one junior PhD student in software engineering with basic security skills.

[11]More about Puppet: https://puppet.com/docs/puppet/7/puppet_overview.html
[12]CWE-798 description is available at https://cwe.mitre.org/data/definitions/798.html
[13]This type of issues can lead to man in the middle (MITM) attacks: https://owasp.org/www-community/attacks/Man-in-the-middle_attack

[14]Why you should not use an admin account: https://www.lbmc.com/blog/why-you-should-not-use-an-admin-account/
[15]Phishing with Unicode domains: https://www.xudongz.com/blog/2017/idn-phishing/
[16]OpenSSF post on scanning OSS software for malicious behavior: https://openssf.org/blog/2022/04/28/introducing-package-analysis-scanning-open-source-packages-for-malicious-behavior/

Table 2: Breakdown of warnings reported by SLIC.

Rule	#	%
Hard-coded secrets	22365	69.9
Use of HTTP without TLS	3757	11.7
Suspicious comments	2780	8.7
Use of Weak Crypto. Algos.	1489	4.7
Invalid IP Address Binding	769	2.4
Empty Password	684	2.1
Admin by default	146	0.5
Total	31990	100

3.1.1 Dataset. To build our dataset, we mined GitHub projects containing `Puppet` scripts. We used three different queries to search for repositories: 1) `language:puppet is:public`; 2) `puppet in:readme is:public`; and, 3) `devops is:public`. We discarded results pointing to forked repositories (to avoid duplicates) and discarded results pointing to repositories without any code in `Puppet` scripts. Our crawler found a total of 1419 GitHub repositories and 34574 associated `Puppet` scripts. SLIC scanned these scripts for the seven sins and reported a total of 31990 security warnings involving 9144 `Puppet` scripts (=26.5% of the total) from 1093 repositories (=73.5% of the total). Table 2 shows the breakdown of warnings reported by SLIC. Column "Rule" shows the name (kind) of the warning, column "#" shows the number of warning reports of that kind, and column "%" shows the percentage of the total associated with that number. This table lists the warnings in order of their prevalence.

3.1.2 Methodology. **Samples.** Given the high number of warnings reported by the tool (31990) and the need for humans to analyze each warning, we sampled a set of reported warnings. We leveraged two popular complementary sampling strategies to that end [5]. *Stratified sampling* is a method to draw samples from a set by taking into account the distribution of kinds—in our case, the distribution of kinds of warnings. A *proportional* (stratified) sampler draws samples in a number proportional to the size of the sets associated with each kind whereas an *uniform* (stratified) sampler draws the same number of samples for each kind. Intuitively, a proportional sampler values more the most prevalent kinds of warnings (as to make more accurate measurements on those kinds) whereas an uniform sampler treats every kind equally (as to avoid inaccurate measurements on uncommon kinds). We sampled 250 warnings *proportionally* and 252 (=36*7) warnings *uniformly*. In total, we analyzed 502 warnings, which is a substantial increase when compared to the 58 warnings analyzed in the SLIC's paper [30]. **Metric.** We focused on precision to measure the reliability of the reports of the tool. Precision is especially important for security linters. Reporting scores of false warnings can be highly disruptive for a team's productivity, as team members tend to interrupt work to address high-priority tasks [13]. Developers are less willing to use linters with low rates of precision because they find them not trustworthy and unreliable [34]. **Statistical Tests.** Each one of the 502 warnings was manually inspected by two co-authors. Cases where the authors found disagreement were discussed to reach a consensus. We report on the results of a Cohen's Kappa analysis [10] to measure the inter-rater reliability of human decisions.

Table 3: Performance of SLIC. (Validation with Students)

SLIC	*proportional*			*uniform*		
Rule	#TP	#FP	Pr.	#TP	#FP	Pr.
Hard-coded secrets	122	52	0.70	26	10	0.72
Use of HTTP without TLS	9	20	0.31	10	26	0.28
Suspicious comments	10	12	0.45	8	28	0.22
Use of Weak Crypto. Algorithms	7	4	0.64	25	11	0.69
Invalid IP Address Binding	6	0	1.00	28	8	0.78
Empty Password	4	2	0.67	21	15	0.58
Admin by default	1	1	0.50	21	15	0.58
Total	159	91	0.64	139	113	0.55

3.1.3 Results. Table 3 shows SLIC's results for both sampling strategies: *proportional* and *uniform*. For each sampling strategy, we present the number of true positives (#TP), the number of false positives (#FP), and the Precision. Considering the results for *proportional* sampling, the authors found a total of 159 true positives and 91 false positives. The average precision of SLIC was 0.64 for the proportional set. Considering the results for *uniform* sampling, the authors found a total of 139 true positives and 113 false positives. On average, SLIC's precision was 0.55 for the uniform set.

We ran a Cohen's Kappa analysis to measure the inter-rater reliability of human decisions in classifying the warnings. The kappa coefficient (k) shows the level of agreement between the two co-authors. The analyses yielded k=0.89 and k=0.94 for the *proportional* and the *uniform* sampling sets, respectively. According to McHugh's interpretation of k [24], the reported levels of agreement are strong and almost perfect for the proportional and uniform sampling sets, respectively. For illustration, agreement was reached in 482 out of 502 warnings. The two co-authors discussed the warnings that raised disagreement. Cases where consensus was not reached were replaced by a new one and re-evaluated. Cases where agreement was reached were updated with the final conclusion—inferred from the discussion between both co-authors.

Summary: Results show that the original precision of SLIC drops considerably from 99% (reported in the original work [28]) to below 65% when tested in a new set of puppet scripts—which might indicate that SLIC needs to be improved. However, the lack of context on the software under analysis by the co-authors may also be the reason for a lower precision. Therefore, we conducted a new experiment with the owners and maintainers of open-source software, i.e., people with more knowledge and context of the applications.

3.2 Validation with Owners of OSS projects

Complementing the preliminary study reported in the previous section, this section reports on the findings of a validation study of SLIC conducted with the maintainers of open source projects. The motivation of this study was to confirm the observations of the previous experiment but now with open source developers, i.e., developers with more context of the software under analysis. GitHub issues were designed to illustrate the security smells (including references to the corresponding CWEs[17]) and to guide the developer towards patching the issue. We followed guidelines for issue reporting from the literature. Issues include code samples, links to more information and we strive to make the report message

[17] Common Weakness Enumeration (CWE) taxonomy available at http://cwe.mitre.org.

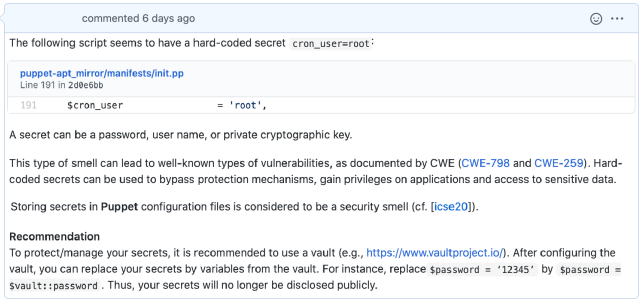

Figure 3: Example of issue opened (based on SLIC report).

as brief and objective [6]. Figure 3 shows an example of an issue created for a "Hard-coded Secret". The title is "Potential vulnerability in Puppet file: Hard-coded Secret". The issue (1) shows where the potential vulnerability is (code: `cron_user = 'root'`, script: `puppet-apt_mirror/manifests/init.pp` and line: 191); (2) explains the vulnerability and its possible implications (bypass protection mechanism, gain privileges on applications, and access to sensitive data); and (3) makes a recommendation to the developer on how she can fix the vulnerability (by using a vault, in this case). We created different templates of this message for the different warning types.

3.2.1 Dataset. The dataset used in this study was different from the one from Section 3.1.2. We mined GitHub projects with activity in 2020 (at least one commit) and containing Puppet scripts. We conjectured that focusing on projects with recent activity would increase our chances of obtaining responses. We used two different queries to search for repositories: 1) `language:puppet is:public`; and, 2) `puppet in:readme is:public`. We discarded results pointing to forked repositories (to avoid duplicates), repositories without any activity in 2020 (no commits), and repositories without any code in Puppet per project. Our tool scanned 3740 Puppet scripts from 287 GitHub repositories. In total, 1975 security warnings were detected in 1147 Puppet scripts (=30.7% of the total). We issued alerts to projects with maintainers involved in the slack of the Puppet community. We received 51 answers to the 228 issues we submitted, but only 33 issues were clearly validated by practitioners—which were the ones we considered for our conclusions.

3.2.2 Methodology. This section presents the methodology followed to submit the issues. **Sample.** A total of 287 GitHub repositories were scanned to this study. We ensured that the repositories had recent activity (at least one commit in 2020) to improve the probability of obtaining responses. The total amount of scripts scanned was 3740 —7 times the sample used in our preliminary study (Section 3.1.1) and 28 times the sample used in Rahman et. al [28] to evaluate the SLIC's precision and recall. **Issues.** We reached out to the software owners through the Puppet community slack and submitted issues to projects with maintainers involved in the slack community. All the issues followed a specific template depending on the type of warning (cf. Figure 3). The issued message not only located and explained the vulnerability but also recommended an example of a patch (i.e., actionable messages to save maintainers time). **Reply Evaluation.** We monitored the discussion threads associated with the issues. For each response obtained, we classified the warnings reported as true positives (TP) or false positives (FP) according to the opinion of the maintainers in their responses. Issues closed by the maintainers without any reply or activity were discarded. Issues closed with unclear responses (e.g., "N/A", ":thumbs_down") were also discarded since they did not provide clarity on the validation of the issue. We only considered the issues where there was some sort of discussion of the issue (e.g., "These todo's shouldn't be there, I agree ... but it's not about defects/weaknesses here. It's just a marker to include more operating systems in the future.") or a clear validation of the issue (e.g., "All false positives", "This is not a secret."). From the answers obtained, two of the co-authors manually inferred the classification of each warning. We use Cohen's Kappa [10] to measure the inter-rater reliability of human decisions. **Metrics.** We measured Precision as described in Section 3.1.2.

Table 4: Performance of SLIC. (Validation with Owners)

Rule	#TP	#FP	Precision
Hard-coded secrets	77	119	0.39
Use of HTTP without TLS	1	72	0.01
Suspicious comments	3	15	0.17
Use of Weak Crypto. Algos.	0	3	0.00
Invalid IP Address Binding	0	1	0.00
Empty Password	1	5	0.17
Admin by default	1	0	1.00
Total	83	215	0.28

3.2.3 Results. This section reports results. **Issues.** We reported a total of 1975 warnings in 228 issues (9 warnings per issue, on average). Project owners responded to 51 issues of the 228 issues we submitted, but only 33 issues were clearly discussed or validated—the equivalent to 298 warnings (Table 4). One issue for an "Empty Password" warning was fixed by one of the maintainers (82c3cb7[18]); one tagged the issue with "waiting for contribution"; another commented asking to perform a pull request. **Reply Evaluation.** We used the following method to determine the warning classification (i.e., false or true positive) from the answers of project owners. We discarded warnings related to issues closed without any response or activity and issues that remained open or without any response by the time of our analysis. After that stage, two of this paper's co-authors reviewed each of the answered issues. Each warning received two votes. Then, we ran a Cohen's Kappa analysis to measure the inter-rater reliability of our choices to assess the confidence in our classification method. The kappa coefficient (k) shows the level of agreement between the two co-authors. The analysis yielded $k = 1.0$, i.e., a total agreement between both co-authors. **Precision.** Table 4 shows the number of true positives (TP) and the number of false positives (FP) per type of warning. In total, SLIC reported 83 true positives and 215 false positives for the 33 issues considered, which resulted in an average precision of 0.28 for SLIC. Note that the samples used for "Use of Weak Crypto. Algorithms", "Invalid IP Address Binding", "Empty Password" and "Admin by default" are relatively small, so results might not reflect the entire reality.

Summary: Results indicate that the precision of SLIC is even lower when evaluated by maintainers—developers with more

[18] Fix for "Empty password" issue: https://github.com/jtopjian/puppet-sshkeys/commit/82c3cb7e78c16cf6517207779f79ab5b2a71b603 (Accessed October 13, 2022)

knowledge and context of the applications—of the software (drops to 28%). These results confirm our initial observations and indicate that better security IaC linters for Puppet are needed.

4 INFRASECURE: PUPPET SECURITY LINTER

The observations described in the previous section motivated the pursuit of a more precise security linter for Puppet scripts. The previous experiments ignited discussions with members of the development and security teams of PuppetLabs, as well as a project manager from Vox Pupuli. We leveraged the feedback obtained from the previous studies and the professional feedback to design a new security linter for the Puppet community, which we dubbed as InfraSecure. More precisely, we created a new linter as a result of *phase 1* (Figure 2) and incrementally evolved the linter according to the recommendations of professionals (Figure 2, *phase 2*), improving 7 rules of the SLIC ruleset and adding 3 new rules. The following sections report the new architecture (Section 4.1), design choices (Section 4.2) and security checkers (Section 4.3) that resulted from the feedback collection (Figure 2):

- *Phase 1*: feedback from the **owners of OSS projects**, **PuppetLabs** and **Voxpupuli Engineers** (as described in Section 3) that led to the creation of InfraSecure (v0.1.0);
- *Phase 2*: two cycles of feedback from the **Puppet** and **Prolific** communities (as described in Section 5) that led to two new releases of InfraSecure (v1.0.0 and v1.1.0);

4.1 Linter Architecture

One recommendation from the PuppetLabs team (*phase 1*) was to implement the set of the rules as plugins to the `puppet-lint` architecture[19], through the `puppet-lint` check API[20]. This API facilitates the integration of new checkers in `puppet-lint`İn addition, it allows the user to suppress warnings and disable or enable checkers—which are regarded as important features by the community. All security checks were developed as plugins to `puppet-lint` (Table 7). These checks are applied to the Abstract Syntax Tree (AST) of a Puppet manifest which is generated by an internal tokenizer[21]. InfraSecure was implemented in `Ruby` and its `CLI` is available online[22]. The codebase of the linter is available at https://github.com/TQRG/puppet-lint-infrasecure and open to future contributions.

4.2 Design Choices

This section describes the design choices of our analysis, guided by the distinct cycles of feedback as described in Section 3 and Section 5; also, illustrated in Figure 2.

Variable/Attribute Assignments (VASS). From the preliminary analysis performed in Section 3, we have noticed security-related code smells being detected in logical conditions. For instance, `if has_key($userdata, 'env')` shows a logical condition that was incorrectly flagged as a hard-coded secret issue. Aiming to reduce the number of incorrect predictions, we implemented a rule to search for variable and attribute assignments in Puppet manifests—`isVarAssign(token)` and `isAtrAssign(token)` (cf. Table 7).

Reasoning about the token value (TOKVAL). Some of the rules did not reason about `token.value`. For hard-coded secrets, the linter only checks if the token value is not empty. While manually validating the samples used in our studies, we found false positives of hard-coded secrets. For instance, `aws_admin_username = downcase($::operatingsystem)` which does not store any actual secret. SLIC flagged this case as a hard-coded secret because the value assigned to the variable `aws_admin_username` is not empty. However, the rule needs to reason not only about the length of the right-hand side of the variable assignment but also about the type of token and value. InfraSecure locates variable and attribute assignments in the AST and considers that secrets are usually stored in `:STRING` and `:SSTRING` tokens. In addition, we defined a database of known credentials (`isUserDefault(token.value)`)—credentials that are not considered secrets by the community[23]—and, invalid secrets (`invalidSecret(token.value)`) which are consider as non-valid values for hard-coded secrets. The linter ignores all the credentials in this database. Feedback from distinct **owners of OSS Projects** is what drove us to make this decision is presented below:

[User Default]: *"The names of these UNIX accounts are not considered to be secret. They are published openly as part of the PE documentation: https://puppet.com/docs/pe/2019.8/what_gets_installed_and_where.html#user_and_group_accounts_installed"*

[Invalid Secret]: *"This are default users and default as found in every installed fpm package. there is most of the time a wwwrun or a www-data user depending on the system."*

Use of HTTP without TLS is fine sometimes (SAFE). As SLIC, InfraSecure also flags every single occurrence of *http://*, i.e., it recommends to use TLS by default. For example, the tool flags `apturl => "http://deb.debian.org/debian"`, even though it refers to a credible source. Our definition of credible source is a source that can be trusted. However, different companies can have different opinions regarding the credibility of the same source. That is why this rule is customizable. We observed that this type of issues (CWE-319) are prevalent in Puppet files. Applications often use third-party libraries, which are usually configured in Puppet files, and the links to their sources are not necessarily unsafe. Also, depending on the context of an application, the configuration of localhost servers as HTTP may not be a problem. If no sensitive data is communicated, then there is probably no problem using `http`. InfraSecure has a configuration file for safe domains, i.e., domains that are cleared to be use `http`. Thereby, infrastructure teams can customize their own configuration files. The feedback provided in Section 5 from two different **practitioners**, which supported this decision, is presented bellow:

[Whitelist]: *"I think it is fine if localhost is used. Otherwise TLS should be mandatory. All the big financial organizations will not use this check because they cannot create internal certs or use letsencrypt."*

[Whitelist]: *"By default, it's unsafe to not use HTTPS. But for internal testing/development it is acceptable to me to not use HTTPS all the time."*

[19] Puppet-lint website: http://puppet-lint.com/
[20] Puppet-lint check API: http://puppet-lint.com/developer/api/
[21] Puppet-lint tokenizer: http://puppet-lint.com/developer/tokens/
[22] Gem is available at https://rubygems.org/gems/puppet-lint-infrasecure

[23] https://puppet.com/docs/pe/2019.8/what_gets_installed_and_where.html#user_and_group_accounts_installed

Hard-Coded Secret Division in different checkers (SECR). In Section 3, we observed that the hard-coded secrets checker produces the most significant number of alerts. For instance, SLIC assumes a secret is a key, password or username. As mentioned previously in Section 2, the Common Weakness Enumeration list does not consider solo hard-coded usernames as a threat. Practitioners involved in our validation studies shared the same opinion. We analyzed the distribution of the different types of hard-coded secrets and realized that 48% of the secrets detected were usernames. Therefore, in the final version of our tool, we decided to separate the hard-coded secrets checker into three new checkers (one per type of secret). This way, developers can disable the username checker if they find it noisy. We did not delete the original checker; infrastructure teams can use it if they want to collect all the different types of secrets simultaneously. Feedback provided in Section 5 from a **practitioner** supported this decision:
[Username]: *"The main security issue is having the password hard-coded. About having the user hard-coded, it is possible to allow that as an initial setting that should be changed during the first configuration and, in that case, it is not so much a security issue."*

4.3 Rules

InfraSecure detects 12 different security smells in Puppet manifests. Table 5b presents the AST patterns that are searched in the AST for relevant nodes/sequences of nodes; and, table 5a presents the string patterns used to validate the information in those nodes. Table 7 shows the syntactic pattern matching rules per weakness which leverage the two sets of patterns mentioned before.

Hard-coded secrets (CWE-321, CWE-259, CWE-798): The top of the Table 7 contains 4 different rules for hard-coded secrets: one per secret; and a final one which detects all kinds of secrets at the same time (keys, password and usernames). In addition to the design choices, the rules consider that secret values cannot be placeholders (`!isPlaceholder()`, Table 5a).

Use of HTTP without TLS (CWE-319): One of the main findings of our analysis is that HTTP without TLS is not always problematic. Therefore, we created a configurable whitelist where infrastructure teams can add safe domains. The checker will not raise alerts when in the presence of a safe domain (`inWhitelist()`, Table 7). InfraSecure provides a default whitelist with known reliable sources such as http://deb.debian.org/debian. However, this default whitelist will be overwritten if the user configures a new one.

Suspicious Comments (CWE-546): This checker was controversial. It was recognized that it would be valuable to alert developers about comments in their code mentioning functionalities and weaknesses that might hint at attackers. However, keywords such as "todo", "later", and "later2" were considered noisy. We changed the list of keywords in response to the complaints and feedback obtained from the developers (`isSuspiciousWord()`, Table 5a).

Usage of Weak Crypto. Algorithms (CWE-326): InfraSecure searches for *in calls to* functions (`isFunctionCall(), Table 5b`) implementing crypto algorithms such as "md5" and "sha1" in variable and attribute assignments (Table 7).

[24]The `strong_password` https://rubygems.org/gems/strong_password ruby gem is used to determine if a password is strong or not.

Table 5: InfraSecure's list of string and AST patterns.

Rule	String Pattern
isAdmin(*t.value*)	root|admin
isNonSecret(*t.value*)	gpg|path|type|buff|zone|mode|tag|header|scheme|length|guid
isPassword(*t.value*)	pass(word|_|$)|pwd
isUser(*t.value*)	user|usr
isKey(*t.value*)	(pvt|priv)+.*(cert|key|rsa|secret|ssl)+
isPlaceholder(*t.value*)	${.*}\|($)?.*::.*(::)?
hasCyrillic(*t.value*)	^(http(s)?://)?.*\p{Cyrillic}+
isInvalidIPBind(*t.value*)	^((http(s)?://)?0.0.0.0(:\d{1,5})?)$
isSuspiciousWord(*t.value*)	hack|fixme|ticket|bug|checkme|secur|debug|defect|weak
isWeakCrypto(*t.value*)	^(sha1|md5)
isCheckSum(*t.value*)	checksum|gpg
isHTTP(*t.value*)	^http://.+
isUserDefault(*t.value*)	pe-puppet|pe-webserver|pe-puppetdb|pe-postgres|pe-console-services|pe-orchestration-services|pe-ace-server|pe-bolt-server
invalidSecret(*t.value*)	undefined|unset|www-data|wwwrun|www|no|yes|[]|undef|true|false|changeit|changeme|none
isStrongPwd(*t.value*) [24]	StrongPassword::StrengthChecker(*t.value*)
isEmptyPassword(*t.value*)	t.value == ""
isVersion(*t.value*)	.*_version

(a) String patterns are applied to token values.

Rule	AST Pattern
isVariable(*t*)	t.type == `:VARIABLE` ∨ t.type == `:NAME`
isString(*t*)	t.type == `:STRING` ∨ t.type == `:SSTRING`
isVarAssign(*t*)	isVariable(*t.prev_code_token*) ∧ t.type == `:EQUALS` ∧ isString(*t.next_code_token*)
isAtrAssign(*t*)	isVariable(*t.prev_code_token*) ∧ t.type == `:FARROW` ∧ isString(*t.next_code_token*)
isResource(*t*)	(t.prev_code_t.type == `:NAME` ∧ t.type == `:LBRACE` ∧ t.next_code_t.type == `:SSTRING`) ∨ (t.prev_code_t.type == `:LBRACE` ∧ t.type == `:SSTRING`)
isFunctionCall(*t*)	(t.type == `:NAME` ∧ t.next_code_token.type == `:LPAREN`) ∨ t.type == `:FUNCTION_NAME`
isComment(*t*)	t.type is in (`:COMMENT`, `:MLCOMMENT`, `:SLASH_COMMENT`)

(b) Patterns applied to the Abstract Syntax Tree (AST).

Invalid IP Address Binding (CWE-284): We found cases where the invalid IP `0.0.0.0` was in descriptions and commands. For instance, SLIC flags `description => 'Open up postgresql for access to sensu from 0.0.0.0/0'`. IPs follow a dot-decimal notation, i.e., they should not include letters. InfraSecure uses a less naive regex than the string pattern (`isInvalidIPBind()`, Table 5a).

Empty Password (CWE-258): Empty passwords are located the same way as hard-coded secrets, i.e., by focusing on variable and attribute assignments (Table 7). The rule `isEmptyPassword()` (Table 5a) verifies if the password is empty.

Table 6: Performance of InfraSecure v0.1.0.

InfraSecure v0.1.0	*proportional*			*uniform*		
Rule	#TP	#FP	Pr.	#TP	#FP	Pr.
Hard-coded secrets	118	22	0.84	24	4	0.86
Use of HTTP without TLS	8	17	0.32	9	23	0.28
Suspicious comments	5	2	0.71	6	10	0.38
Use of Weak Crypto. Algorithms	5	2	0.71	23	2	0.92
Invalid IP Address Binding	6	0	1.00	28	1	0.97
Empty Password	4	2	0.67	21	15	0.58
Admin by default	1	1	0.50	20	15	0.57
Total	147	46	0.76	131	70	0.65

Admin By Default (CWE-250): These issues are also located by focusing on variable and attribute assignments (Table 7). The rule `isAdmin()`, table 5a, verifies if the user is "admin" or "root".

Homograph Attacks (CWE-1007): Typosquatting attacks, also known as URL hijacking, is a social engineering attack that purposely uses misspelt domains for malicious purposes; and are the cause of many supply chain attacks [14]. This checker is important because malicious actors can use homoglyphs to modify reliable sources for malicious sources (`hasCyrillic()`, Table 5a).

Weak Password (CWE-521): InfraSecure searches for passwords in the same way it searches for Empty Passwords and Hard-Coded Passwords. The only difference is the password value validation (`isStrongPwd()`, Table 5a) which is performed by an external package (`strong_password`) that implements an adaptation of a PHP algorithm developed by Thomas Hruska [20].

Malicious Dependencies (CWE-829): We produced a database of malicious dependencies for Puppet modules by crossing CVEs information and vulnerable products names with third-party libraries that can be configured in Puppet manifests. We used the National Vulnerability Database (NVD) to collect the CVEs and respective vulnerable products—from the list of Known Affected Software Configurations. To get the list of products used by Puppet, we used the Forge API[25]. Our database integrates malicious dependencies for 33 different packages (e.g., `rabbitmq`, `apt`, `cassandra`, `postgresql`, etc). The checker searches for resource configurations (`isResource()`, Table 5b) and verifies if the a configured version of the software integrates our database of malicious dependencies for Puppet (`isMalicious()`, Table 7).

4.4 Proof of Concept: InfraSecure v0.1.0

As a proof of concept, two of the design choices described in Section 4.2 were implemented in the first version, InfraSecure v0.1.0, to ascertain whether precision could be enhanced. In particular, we focused on implementing variable and attribute assignments (VASS) and reasoning about the token value (TOKVAL), to reduce the number of incorrect detections.

In our preliminary analysis with students (see Section 3, Table 3), we observed that the precision of SLIC was 64%. By implementing the two design choices mentioned before, we increased precision by 12 per cent points—when comparing SLIC's precision in the *proportional* set (64%) with the precision of the first version of InfraSecure in the same dataset (76%), Table 6. As these changes were successful w.r.t. precision, we decided to implement the other improvements and conduct a new study with practitioners to collect more feedback about the tool and the anti-patterns covered.

5 PRACTITIONERS EVALUATION

InfraSecure was validated with practitioners experienced in security or configuration management technologies. We built an experiment to validate the warnings of the new tool. The experiment was shared with the Puppet communities on Slack (`puppetcommunity.slack.com`) and Reddit (`r/puppet`). We found 14 participants by this mean. Later, we leveraged Prolific[26] [33] to gather more participants based on their experience and programming knowledge. In this experiment, a total of 339 warnings were validated by 131 practitioners. Furthermore, our improvements increased the precision of the tool from 28% to 83%. As illustrated in Figure 2, we run two cycles of feedback collection and iteratively improve the tool with the feedback collected. This section describes 1) the methodology conducted with practitioners to validate the InfraSecure warnings; and 2) the results obtained from running the practitioners' experiment.

5.1 Study Design

In this section, we detail how the validation study of InfraSecure was designed, and the population leveraged to conduct it. InfraSecure was improved based on the problems collected through the preliminary study and the validations with the maintainers of the software—which led to InfraSecure v0.1.0. To validate the new tool, we surveyed practitioners with experience in security, configuration management tools and programming knowledge by following recent recommendations to run studies on Prolific [33]. After the pre-screening, the practitioners were asked to validate and give feedback on 3 different warnings generated by InfraSecure.

Practitioners Recruitment. The participants were obtained using two distinct routes: 1) By sharing the study with online Puppet communities such as `puppetcommunity.slack.com` (slack) and `r/puppet` (reddit); 2) By using the Prolific platform to gather practitioners with experience in security, configuration management tools and programming skills. Both communities integrate a considerable amount of members: slack has over 9k members, and Reddit has around 4.7k members. However, only 14 members participated in our study. Therefore, we used Prolific to collect more practitioners with experience in security and configuration management tools outside of these two communities. Prolific participants were monetarily compensated for answering each survey, while the participants collected in the communities were not.

Pre-Screening. Prolific is a platform where you can find participants to perform online research. As recommended in research on recruiting practitioners for user studies on prolific [33], we performed a pre-screening of the population to collect adequate participants for this study, i.e., participants with security and configuration management experience; and programming knowledge. Prolific has filters dedicated to the industry where the participants work or belong. We sent the pre-screening survey to prolific users working in the following industries: "Computer and Electronics

[25]Forge API is available at https://forgeapi.puppet.com/

[26]Prolific Platform: https://www.prolific.co/

Table 7: InfraSecure rules to detect security smells.

CWE	Weakness Name	Rule
CWE-321	Hard-coded Key	(isVarAssign(t) ∨ isAtrAssign(t)) ∧ isKey(*t.prev_code_token*) ∧ isNonSecret(*t.prev_code_token*) ∧ !isPlaceholder(*t.next_code_token*)
CWE-259	Hard-coded Password	(isVarAssign(t) ∨ isAtrAssign(t)) ∧ isPassword(*t.prev_code_token*) ∧ isNonSecret(*t.prev_code_token*) ∧ !isPlaceholder(*t.next_code_token*) ∧ !isUserDefault(*t.next_code_token*) ∧ !invalidSecret(*t.next_code_token*)
CWE-798	Hard-coded Usernames	(isVarAssign(t) ∨ isAtrAssign(t)) ∧ isUser(*t.prev_code_token*) ∧ isNonSecret(*t.prev_code_token*) ∧ !isPlaceholder(*t.next_code_token*) ∧ !isUserDefault(*t.next_code_token*) ∧ !invalidSecret(*t.next_code_token*)
CWE-798	Hard-coded Secrets	(isVarAssign(t) ∨ isAtrAssign(t)) ∧ (isKey(*t.prev_code_token*) ∨ isPassword(*t.prev_code_token*) ∨ isUser(*t.prev_code_token*)) ∧ !isPlaceholder(*t.next_code_token*) ∧ !isUserDefault(*t.next_code_token*) ∧ !invalidSecret(*t.next_code_token*)
CWE-319	Use of HTTP without TLS	(isVarAssign(t) ∨ isAtrAssign(t)) ∧ isHTTP(*t.next_code_token*) ∧ !inWhitelist(*t.next_code_token*)
CWE-546	Suspicious Comments	isComment(t) ∧ isSuspiciousWord(t)
CWE-326	Use of Weak Crypto. Algo.	(isVarAssign(*t.prev_code_token*) ∨ isAtrAssign(*t.prev_code_token*) ∨ isFunctionCall(*t.next_code_token*)) ∧ !isCheckSum(*t.prev_code_token*) ∧ isWeakCrypto(*t.next_code_token*)
CWE-284	Invalid IP Address Binding	(isVarAssign(t) ∨ isAtrAssign(t)) ∧ isInvalidIPBind(*t.next_code_token*)
CWE-258	Empty Password	(isVarAssign(t) ∨ isAtrAssign(t)) ∧ isPassword(*t.prev_code_token*) ∧ isEmptyPassword(t.prev_code_token)
CWE-250	Admin by default	(isVarAssign(t) ∨ isAtrAssign(t)) ∧ isNonSecret(*t.prev_code_token*) ∧ isUser(*t.prev_code_token*) ∧ !isPlaceholder(*t.next_code_token*) ∧ isAdmin(*t.next_code_token*)
CWE-1007	Homograph Attacks	(isVarAssign(t) ∨ isAtrAssign(t)) ∧ hasCyrillic(*t.next_code_token*)
CWE-521	Weak Password	(isVarAssign(t) ∨ isAtrAssign(t)) ∧ isPassword(*t.prev_code_token*) ∧ isStrongPwd(*t.next_code_token*)
CWE-829	Malicious Dependencies	isResource(t) ∧ isVersion(*t.prev_code_token*) ∧ isMalicious(*t.next_code_token*)

`inWhitelist(t.value)` verifies if the URL is in the list of configurable safe domains/whitelist. If the URL is in the whitelist, an alert should not be raised.
`isMalicious(t.value)` verifies if the software package version configured in the puppet script is in the database of malicious dependencies.

Manufacturing", "Information Services and Data Processing", "Product Development", "Research laboratories", "Scientific or Technical Services", "Software". The participants were asked to answer the following questions: 1) Do you have any kind of experience with configuration management tools? **Choices:** Puppet, Ansible, Terraform, Chef, Other; 2) Experience in Security (Number of Years); 3) Experience in Infrastructure as a Service (Number of Years); and three programming language questions about different puppet configurations. Due to space constraints, we do not present the questions here, but they are available in our replication package: study/practitioners/pre-screening/puppet-study-form.pdf.

We obtained a total of 431 responses from 8 different industries. Then, we ordered those participants by priority where priority is the count of experience in 1) at least one configuration management tool (*CMEXP*), 2) security (*SECEXP*), 3) infrastructure as a service (*INFRAEXP*); and 4) score in the programming questions (*SCORE*). Priority was calculated as follows $0.3 * ((CMEXP + SECEXP + INFRAEXP)/3) + 0.7 * (SCORE/3)$ and varies between 0 to 3. A priority of 3 means the participant is adequate for the study, whereas a priority of 0 means the participant is not adequate. For the validation study, we only invited participants with priority equal to or greater than 1.5—which represented 53% of the initial responses (227 out of 431 participants).

Validation Form. We built a form online to share with the Puppet communities and practitioners. The initial page of the form explains the study's goal and asks the participant for her profession/career, number of years of experience in security, and number of years of experience in infrastructure/puppet. The goal of the study is to validate the output of our new tool: InfraSecure. Therefore, participants are required to validate 3 different warnings (one at each time). For each warning, the form presents a description of the issue and the piece of code where the issue is located (cf. Figure 4). Participants have to evaluate the issue and provide their validation: "Yes, I agree", if the warning reports a security issue; "No, I disagree", if it reports a false security issue; or, "I'm not sure", when unsure. The participant can also provide additional feedback on the problem.

Warnings Dataset. For this experiment, we validated the output of 9 different rules (Table 7), where the warnings for weak passwords and malicious dependencies were mostly validated in the second round of feedback collection. We ran the InfraSecure over a total of 1050 GitHub projects—collected from the dataset used in the preliminary study (Section 3). We created a uniform sample with 50 warnings per rule (i.e., a total of 450 warnings).

Metrics. We report the number of true positives (TPs), the number of false positives (TPs), the number of "Unsure" responses and Precision—calculated as described in Section 3.1.2.

5.2 Results

We obtained a total of 131 participants: 74 in the first round of feedback; and 57 in the second round. Due to the lack of responses from participants, we were only able to validate 342 out of the 450 initial number of warnings. Table 8 shows the distribution of warnings and precision obtained for the final version of InfraSecure (v1.1.0). At the end of this study, InfraSecure reported a precision of 83%, where 54 of the warnings were False Positives.

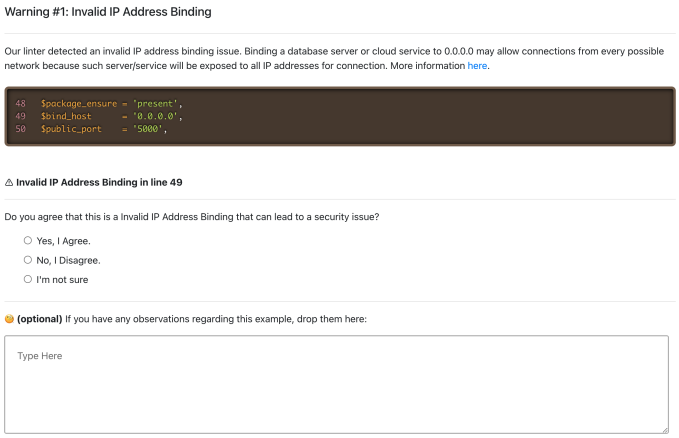

Figure 4: Example of the form presented to the practicioner for warning validation.

Table 8: Performance of InfraSecure (v1.1.0). (Validation with Practitioners)

Rule	#TP	#FP	#Unsure	Precision
Hard-coded secrets	28	8	3	0.78
Use of HTTP without TLS	32	3	2	0.91
Suspicious Comments	16	15	7	0.52
Use of Weak Crypto. Algo.	33	3	6	0.92
Invalid IP Address Binding	26	8	6	0.77
Empty Password	33	3	1	0.92
Admin by default	30	6	6	0.83
Malicious Dependencies	25	6	3	0.81
Weak Password	32	2	0	0.94
Total	255	54	34	0.83

Part of the feedback obtained in this experiment was documented in Section 4.2 and 4.3. Table 9 shows the evolution of the tool's precision with the different iterations of feedback. In contrast to Table 6, where we report the precision of v0.1.0 for the alerts validated by students; here, in Table 9, we report the precision of v0.1.0 based on the practitioners' feedback, i.e., leveraging the alerts validated by practitioners (instead of students). It is important to note that the implementation of v0.1.0 focused on understanding variable and attribute assignments and reasoning about the token value to reduce the number of incorrect detections. These two improvements affected all checkers. The remaining versions of the tool focused on addressing specific false positives, extending the ruleset and adding the safe domains feature. In the comparison provided in Table 9, we observe an increase of precision—from 76% to 83%—by conducting different cycles of feedback collection. In addition, feedback was essential to extend the ruleset. This study with practitioners led us to create 3 new rules to detect weak passwords; typosquatting attacks; and malicious dependencies (being the last two the root causes of many supply chain attacks [14, 15]).

Summary: Results show that working side-by-side with the community will help the authors of the tools develop better linters, as proposed before by a Google study [34]. Using this feedback approach, we improved the linter's precision and the final ruleset.

Table 9: Precision obtained in different cycles of feedback collection for InfraSecure.

Participants	version	Precision
Research Team, Owners of OSS Projects, PuppetLabs, Voxpupuli	v0.1.0	76%
Practitioners (cycle 1)	v1.0.0	79%
Practitioners (cycle 2)	v1.1.0	83%

5.3 Discussion & Limitations

This paper reports our approach to improve the ruleset of an IaC security iteratively linter in different cycles of feedback collection. However, the tool can still be improved with more sophisticated techniques such as data-flow analysis, which would fulfil the following feedback: *In puppet, pre-defining a password as empty does not mean it is empty (e.g.,* `$ssl_password = ''`*). Many times these variables are changed later. Thus, for each empty password, InfraSecure verifies if the same variable was changed within the same file. If it was, then the linter will not raise an issue.*.

In addition, some engineers suggested that usernames should be only reported as hard-coded secrets when paired with a password/key. For this, we must match the different pairs of credentials in a puppet manifest. To sum up, there are still opportunities to improve the precision and recall of InfraSecure. We reached out to owners of highly active GitHub projects that use Puppet reporting warnings detected by InfraSecure. Two owners mentioned that since the apps are not in production, they did not consider the issues relevant. Even after improving the linter to detect the anti-patterns correctly, some problems are still not problematic. This happens because the linter does not have context regarding the software's usage, which will always be a source of False Positives. In the future, we will continue to search for solutions to make the linter more context-aware since this is a known problem of linters.

6 ETHICAL STANDARDS AND COMPLIANCE

This section discusses compliance with the ACM Policy for Research Involving Humans,[27] which ensures that the ethical and legal standards are met when research has human participants.

Informed Consent. One of the principles is to ensure that participants are informed about the fact that they are participating in a study. In our study, consent was collected differently for each experiment: for the first one, the research team agreed to participate in the study; for the OSS maintainers experiment, we used the puppet community slack to communicate and discuss the investigation with the maintainers; finally, for the practitioners' experiment, we asked survey participants if they agreed to participate in our different surveys at the beginning of the pre-screening phase.

Data Privacy. For all experiments, we ensured that the participants' private information was protected by not providing the participants' personal data (e.g., GitHub usernames of the OSS maintainers, prolific participants' names, ages, nationalities, etc.) in our replication package.

Spam. As mentioned in Section 3.2, we carefully organized the issues to minimize the amount of messages sent to maintainers [3,

[27]The ACM Policy for Research Involving Humans description is available at https://www.acm.org/publications/policies/research-involving-human-participants-and-subjects (Accessed October 13, 2022)

17]. Security smells of the same kind were all reported in a single GitHub issue. In addition, we designed the issues to be actionable by providing personalized fix suggestions and adding references that document the detected problems.

Full Disclosure in Security. Fully disclosing vulnerabilities on GitHub issues allows hackers to exploit unfixed vulnerabilities, creating risks for software users. Ideally, vulnerability disclosure should be performed confidentially. Yet, GitHub does not provide a feature to report them privately. Therefore, carefully performing full disclosure is accepted by OSS maintainers [2, 7]. Some OSS projects adopt security mailing lists. In those cases, disclosure should be performed through those mailing lists.

7 THREATS TO VALIDITY

This section presents potential threats to the validity of this study.

Internal Validity: As with any implementation, the scripts that we developed to run the tools and collect the metrics reported in the paper are potentially not bug-free. However, the scripts and outputs are open-source for other researchers and potential users to check the validity of the results.

Construct Validity: A potential threat is the manual analysis of the warnings raised by SLIC in the Puppet scripts, which can potentially be mislabeled. We tried to mitigate this by running a kappa analysis between the two co-authors. For the experiments with the OSS maintainers, we inferred the validations of the alerts from their comments. Although both co-authors inferred the validations, and a kappa analysis was performed, we risk our inference being incorrect. It is also important to mention that even though we made an effort to collect feedback from experienced humans, their judgement can also not be 100% accurate, which can introduce error in the precision values reported.

External Validity: A potential threat to external validity is related to the fact that the set of Puppet scripts we have considered in this study may not accurately represent the whole set of vulnerabilities that can happen during development. We attempt to reduce the selection bias by gathering a large collection of real, openly available (hence, reproducible) Puppet scripts. Another potential threat is that we could have missed the latest updates to SLIC. To mitigate this risk, we contacted the authors of SLIC to confirm that the version available is true to the most recent one.

8 RELATED WORK

As IaC has become popular and prevalent, researchers have dedicated efforts to improve its quality. Jiang et al. conducted an empirical study on Puppet scripts to gain a deep understanding of the characteristics of such scripts and how they evolve over time [21]. Bent et al. investigated the quality and maintainability aspects of Puppet scripts [35]. Furthermore, Rahman et al. proposed prediction models (based on text mining) to classify defective IaC scripts [31]. Palma et al. created a catalog of software metrics for IaC scripts [11]. In addition, recent work has been developed to detect malicious packages published on registry maintainers such as npm and ruby gems [14]. Building and introducing linters earlier in the software development life-cycle shift security left and decreases the probability of shipping malicious packages.

There are several linters available for security but only the subject of this paper, SLIC, focuses in IaC scripts for Puppet [28]. The authors started by demonstrating the linter in the context of Puppet scripts, and later, the authors reproduced the same study for Chef and Ansible and created new tools for those technologies [30]. A major issues with linters is their lack of precision [9, 16, 23, 25, 27, 36]: low precision entails low reliability for developers. Previous research has shown the impact of this issue on the developers' workflow and stressed it is essential to create precise tools; otherwise, the developers will not use them [4, 8, 12, 19, 22, 34]. As mentioned before, this study aims to gain a better understanding of the current capabilities of the only IaC security linter for Puppet and shed some light on how to move forward.

9 CONCLUSIONS & FUTURE WORK

In this study, we observed through a comprehensive study that security linters for IaC scripts still need to be improved to be adopted by the industry. This paper leverages community expertise to address the challenge of improving the precision of such linting tools. We focused on precision as it is critically important in this domain—false security warnings can be very disruptive. More precisely, we interviewed professional developers of Puppet scripts to collect their feedback on the root causes of imprecision of the state-of-the-art security linter for Puppet. From that feedback, we developed a linter adjusting 7 rules of an existing linter ruleset and adding 3 new rules. We conducted a new study with 131 professional developers, showing an increase in precision from 28% to 83%. Following the findings of a Google study [34], we show that authors of linters can improve their own tools if they focus on the users' feedback. The takeaway messages of this paper are that (i) it is feasible to tune security linters to produce acceptable precision; and, that (ii) involving practitioners in discussions is an effective way to guide the improvement of those linters.

The observations that we made throughout this work pave the way for the following future work: extend InfraSecure to detect other security vulnerabilities, integrate the tool with methods for automated patching, and port InfraSecure to other configuration management tools.

ACKNOWLEDGEMNTS

We thank the Puppet Community for their support and involvement in the different studies. This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia with reference UIDB/50021/2020 and a PhD scholarship (ref. SFRH/BD/143319/2019). This work is partially supported by INES (www.ines.org.br); CNPq grant 465614/2014-0; CAPES grant 88887.136410/2017-00; FACEPE grants APQ-0399-1.03/17 and PRONEX APQ/0388-1.03/14.

REFERENCES

[1] 2020. Palo Alto Networks. Unit 42 Cloud Threat Report 2H 2020. https://www.paloaltonetworks.com/prisma/unit42-cloud-threat-research.

[2] 2022. Ask an Ethicist: Vulnerability Disclosure. https://ethics.acm.org/integrity-project/ask-an-ethicist/ask-an-ethicist-vulnerability-disclosure/.

[3] Sebastian Baltes and Stephan Diehl. 2016. Worse Than Spam: Issues In Sampling Software Developers. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'16)*. Association for Computing Machinery, New York, NY, USA, Article 52, 6 pages. https://doi.org/10.1145/2961111.2962628

[4] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[5] Zdravko Botev and Ad Ridder. 2014. Variance reduction. *Wiley StatsRef: Statistics Reference Online* (2014), 1–6.

[6] Antônio Carvalho, Welder Luz, Diego Marcílio, Rodrigo Bonifácio, Gustavo Pinto, and Edna Dias Canedo. 2020. C-3PR: A Bot for Fixing Static Analysis Violations via Pull Requests. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'20)*. IEEE, 161–171. https://doi.org/10.1109/SANER48275.2020.9054842

[7] Andrew Cencini, Kevin Yu, and Tony Chan. 2005. Software Vulnerabilities: Full-, Responsible-, and Non-Disclosure. https://courses.cs.washington.edu/courses/csep590/05au/whitepaper_turnin/software_vulnerabilities_by_cencini_yu_chan.pdf

[8] Foteini Cheirdari and George Karabatis. 2018. Analyzing False Positive Source Code Vulnerabilities Using Static Analysis Tools. In *IEEE International Conference on Big Data (Big Data'18)*. 4782–4788. https://doi.org/10.1109/BigData.2018.8622456

[9] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. 332–343. https://doi.org/10.1145/2970276.2970347

[10] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46. https://doi.org/10.1177/001316446002000104

[11] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian Andrew Tamburri. 2020. Towards a catalogue of software quality metrics for infrastructure code. *Journal of Systems and Software* (2020), 110726. https://doi.org/10.1016/j.jss.2020.110726

[12] Carlo Dimastrogiovanni and Nuno Laranjeiro. 2016. Towards Understanding the Value of False Positives in Static Code Analysis. In *Latin-American Symposium on Dependable Computing (LADC'16)*. 119–122. https://doi.org/10.1109/LADC.2016.25

[13] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. https://doi.org/10.1145/3338112

[14] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *Network and Distributed Systems Security Symposium (NDSS'21)*. https://doi.org/10.14722/ndss.2021.23055

[15] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2021. Containing Malicious Package Updates in npm with a Lightweight Permission System. In *IEEE/ACM International Conference on Software Engineering (ICSE'21)*. 1334–1346. https://doi.org/10.1109/ICSE43902.2021.00121

[16] François Gauthier, Nathan Keynes, Nicholas Allen, Diane Corney, and Padmanabhan Krishnan. 2018. Scalable Static Analysis to Detect Security Vulnerabilities: Challenges and Solutions. In *IEEE Cybersecurity Development (SecDev'18)*. IEEE, 134–134. https://doi.org/10.1109/SecDev.2018.00030

[17] Nicolas E. Gold and Jens Krinke. 2020. Ethical Mining: A Case Study on MSR Mining Challenges. In *International Conference on Mining Software Repositories (MSR'20)*. Association for Computing Machinery, New York, NY, USA, 265–276. https://doi.org/10.1145/3379597.3387462

[18] Michele Guerriero, Martin Garriga, Damian A. Tamburri, and Fabio Palomba. 2019. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In *IEEE International Conference on Software Maintenance and Evolution (ICSME'29)*. 580–589. https://doi.org/10.1109/ICSME.2019.00092

[19] M. Harman and P. O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'18)*. 1–23. https://doi.org/10.1109/SCAM.2018.00009

[20] Thomas Hruska. 2011. How to calculate Password Strength... http://cubicspot.blogspot.com/2011/11/how-to-calculate-password-strength.html.

[21] Yujuan Jiang and Bram Adams. 2015. Co-evolution of Infrastructure and Source Code - An Empirical Study. In *IEEE/ACM Working Conference on Mining Software Repositories (MSR'15)*. 45–55. https://doi.org/10.1109/MSR.2015.12

[22] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *International Conference on Software Engineering (ICSE'13)*. 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[23] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. 2017. Challenges for Static Analysis of Java Reflection - Literature Review and Empirical Study. In *IEEE/ACM International Conference on Software Engineering (ICSE'17)*. 507–518. https://doi.org/10.1109/ICSE.2017.53

[24] Mary L McHugh. 2012. Interrater Reliability: the Kappa Statistic. *Biochemia medica: Biochemia medica* 22, 3 (2012), 276–282.

[25] Tukaram Muske and Alexander Serebrenik. 2016. Survey of Approaches for Handling Static Analysis Alarms. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'16)*. IEEE, 157–166. https://doi.org/10.1109/SCAM.2016.25

[26] Pars Mutaf. 1999. Defending against a Denial-of-Service Attack on TCP. In *Recent Advances in Intrusion Detection, Second International Workshop (RAID'99)*.

[27] Joonyoung Park, Inho Lim, and Sukyoung Ryu. 2016. Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild. In *IEEE/ACM International Conference on Software Engineering Companion (ICSE-C'16)*. 61–70.

[28] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *IEEE/ACM International Conference on Software Engineering (ICSE'19)*. 164–175. https://doi.org/10.1109/ICSE.2019.00033

[29] Akond Rahman, Asif Partho, David Meder, and Laurie Williams. 2017. Which Factors Influence Practitioners' Usage of Build Automation Tools?. In *IEEE/ACM International Workshop on Rapid Continuous Software Engineering (RCoSE'17)*. 20–26. https://doi.org/10.1109/RCoSE.2017.8

[30] Akond Rahman, Md Rayhanur Rahman, Chris Parnin, and Laurie Williams. 2021. Security Smells in Ansible and Chef Scripts: A Replication Study. *ACM Trans. Softw. Eng. Methodol.* 30, 1, Article 3 (Jan. 2021), 31 pages. https://doi.org/10.1145/3408897

[31] Akond Rahman and Laurie Williams. 2018. Characterizing Defective Configuration Scripts Used for Continuous Deployment. In *IEEE International Conference on Software Testing, Verification and Validation (ICST'18)*. IEEE, 34–45. https://doi.org/10.1109/ICST.2018.00014

[32] Akond Rahman and Laurie Williams. 2021. Different Kind of Smells: Security Smells in Infrastructure as Code Scripts. *IEEE Security Privacy* 19, 3 (2021), 33–41. https://doi.org/10.1109/MSEC.2021.3065190

[33] Brittany Reid, Markus Wagner, Marcelo d'Amorim, and Christoph Treude. 2022. Software Engineering User Study Recruitment on Prolific: An Experience Report. In *International Workshop on Recruiting Participants for Empirical Software Engineering (RoPES'22)*.

[34] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (Mar. 2018), 58–66. https://doi.org/10.1145/3188720

[35] Eduard Van der Bent, Jurriaan Hage, Joost Visser, and Georgios Gousios. 2018. How good is your puppet? An empirically defined and validated quality model for puppet. In *International Conference on Software Analysis, Evolution and Reengineering (SANER'20)*. IEEE, 164–174. https://doi.org/10.1109/SANER.2018.8330206

[36] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. 2020. How Developers Engage with Static Analysis Tools in Different Contexts. *Empirical Software Engineering* 25, 2 (2020), 1419–1457. https://doi.org/10.1007/s10664-019-09750-5

[37] Xiaoyun Wang and Hongbo Yu. 2005. How to Break MD5 and Other Hash Functions. In *Advances in Cryptology (EUROCRYPT'05)*, Ronald Cramer (Ed.). 19–35.