A Different Kind of Smell: Security Smells in Infrastructure as Code Scripts

Rahman, Akond
Tennessee Technological University

Williams, Laurie
North Carolina State University

***Abstract*—**
In this paper we summarize our recent research findings related to infrastructure as code (IaC) scripts where we have identified 67,801 occurrences of security smells that include 9,175 hard-coded passwords. We hope our paper will facilitate awareness amongst practitioners who use IaC.
Keywords: ansible, chef, empirical study, puppet, security smell, infrastructure as code

■ **THE INTRODUCTION** System administrators (sysadmins) are involved in critical tasks, including setting up user accounts, installing package dependencies to maintain development and deployment environments, and fulfilling information technology (IT) support management. For a long time, sysadmins developed and maintained custom BASH/Perl scripts to perform their tasks [9]. Recently, with the wide-spread availability of cloud computing resources, manually executing custom scripts has become error prone and time consuming. Imagine you want to create a text file with the text 'Hello world!'. One way you can achieve this task is to open a file, write in the content, and setting the permission settings using chmod or by changing the properties of the file. This process is manual, and would not scale if you want to create the same file across 1,000 Amazon Web Services (AWS) instances, as you have to login to these instances one at a time, and perform the above-mentioned steps. With the practice of infrastructure as code (IaC) sysadmins can now execute a single IaC script once, and execute the task of creating a text file in 1,000 AWS instances.

Infrastructure as code (IaC) scripts help practitioners to provision and configure their development environment and servers at scale [9]. Commercial IaC tool vendors, such as Chef and Puppet, provide programming syntax and libraries so that programmers can specify configuration and dependency information as scripts. Similar to software source code, IaC scripts are treated as ‘first class citizens’ i.e. these scripts undergo software quality activities, such as linting and testing, and also maintained in a version control system (VCS) system [9].

The information technology (IT) industry has experienced benefits with the usage of IaC. For example, the Intercontinental Exchange (ICE), which runs millions of financial transactions daily [10], maintains 75% of its 20,000 servers using IaC scripts [7]. The use of IaC scripts has helped ICE decrease the time needed to provision development environments from 1∼2 days to 21 minutes [7]. A wide variety of tools exist to implement IaC, e.g., Ansible, Chef, Puppet, SaltStack, and Terraform. According to a RightScale 2019 State of the Cloud report, which conducted a survey with 786 practitioners, Ansible is perceived as the most popular tool to implement IaC [1].

Despite these reported benefits, in the open source software (OSS) domain, IaC scripts may be susceptible to security weaknesses, such as hard-coded passwords. Hard-coded passwords can give the pathway to a malicious user to attack the system. Anecdotes like this example, motivated us to look into this problem in detail, taking a systematic approach.

Our paper summarizes our findings from our recent work [10], [12] in which we answer the following questions:

- What categories of security smells appear in Infrastructure as Code scripts?
- How frequently do security smells appear in Infrastructure as Code scripts?
- What do practitioners think about the identified security smells in Infrastructure as Code scripts?

We build on our previously-published research to synthesize and disseminate our findings to a practitioner-focused community. Our additional contribution is that we infer actionable insights and lessons that practitioner can discuss and apply to IaC development. These insights and lessons can be leveraged to integrate security early in the development process of IaC.

What categories of security smells appear?

A code smell is a recurrent coding pattern that is indicative of potential maintenance problems [4]. A code smell may not always have bad consequences, but still deserves attention, as a code smell may be an indicator of a problem [4]. Our paper focuses on identifying security smells. Security smells are recurring coding patterns that are indicative of security weakness and warrant further inspection [10].

Security smells are different from vulnerabilities, as they are coding patterns that are indicative of a weakness. A vulnerability is defined as a weakness, which can be used by a party to cause a software to modify or access unintended data, interrupt proper execution, or perform incorrect actions that were not specifically granted to the party. Labeling a coding pattern as a vulnerability necessitates consideration of application context, which is not applicable for security smells. Security smells provide a mechanism for security-focused inspection that may or may not lead to a vulnerability but deserves attention, inspection, and if necessary, mitigation. Let us consider an example of hard-coded secret. A hard-coded secret, such as a hard-coded password is only relevant if the password is used to login to a software system. Understanding whether or not the password is being used for a login can be determined by understanding program context, possibly by using data flow analysis. Our analysis does not capture these contexts, which motivated us to use the term security smell instead of vulnerability.

Methodology

To identify security smells, we apply manual analysis with a set of 3,339 Ansible, Chef, and Puppet scripts to determine security smells. A summary of the collected scripts is available in Table 1. The 29 repositories used in our manual analysis were downloaded on November 2016 by cloning the master branches. The Puppet-related repositories were collected from Mozilla, whereas, the Ansible and Chef-related repositories were collected from Openstack.

To derive what security smells appear, we first apply a qualitative analysis technique called open coding on 1,726 scripts to identify security smells. Next, we map each identified smell to a possible security weakness defined by the Mitre Common Weakness Enumeration (CWE) [5].

Table 1. Summary Statistics of Collected Scripts to Determine Security Smells

Lang.	Duration	Repository Count	Org.	Script Count
Ansible	2014-02 to 2016-11	16	Openstack	1,101
Chef	2011-05 to 2016-11	11	Openstack	855
Puppet	2014-09 to 2016-11	2	Mozilla	1,383

Findings

We found nine security smells from our collection of Ansible, Chef, and Puppet scripts. Not all smells appear for all three tools. A complete mapping of each security smell and the tool it appears for is listed in Table 2.

Admin by default: This smell is the recurring pattern of specifying default users as administrative users. The smell can violate the 'principle of least privilege' property [6], which recommends practitioners design and implement a system in a manner so that, by default, the least amount of access necessary is provided to any entity.

Empty password: This smell is the recurring pattern of using a string of length zero for a password. An empty password is indicative of a weak password. An empty password does not always lead to a security breach, but makes it easier to guess the password. For example, if your MySQL server allows 'root' access from a remote machine and the superuser 'root' has an empty password then anyone can connect with your MySQL server without a password with all privileges. An empty password is different from using no passwords. In SSH key-based authentication, instead of passwords, public and private keys can be used. Our definition of empty password does not include usage of no passwords and focuses on attributes/variables that are related to passwords and assigned an empty string. Empty passwords are not included in hard-coded secrets because for a hard-coded secret, a configuration value must be a string of length one or more.

Hard-coded secret: This smell is the recurring pattern of revealing sensitive information such as user name and passwords as configurations in IaC scripts. IaC scripts provide the opportunity to specify configurations for the entire system, such as configuring user name and password, setting up SSH keys for users, specifying authentications files (creating key-pair files for Amazon Web Services). However, in the process programmers can hard-code these pieces of information into scripts. We consider three types of hard-coded secrets: hard-coded passwords, hard-coded user names, and hard-coded private cryptography keys.

Unrestricted IP address binding: This smell is the recurring pattern of assigning the address 0.0.0.0 for a database server or a cloud service/instance. Binding to the address 0.0.0.0 may cause security concerns as this address can allow connections from every possible network. Such binding can cause security problems as the server, service, or instance will be exposed to all IP addresses for connection. For example, practitioners have reported how binding to 0.0.0.0 facilitated security problems for MySQL, Memcached (cloud-based cache service) and Kibana (cloud-based visualization service). We acknowledge that an organization can opt to bind a database server or cloud instance to 0.0.0.0, but this case may not be desirable overall.

Missing Default in Case Statement This smell is the recurring pattern of not handling all input combinations when implementing a case conditional logic. Because of this coding pattern, an attacker can guess a value, which is not handled by the case conditional statements and trigger an error. Such error can provide the attacker unauthorized information for the system in terms of stack traces or system error.

No integrity check This smell is the recurring pattern of not checking repository content that is being downloaded using checksums and gpg signatures. By not checking for integrity, a developer assumes the downloaded content is secure and has not been corrupted by a potential attacker.

Suspicious comment: This smell is the recurring pattern of putting information in comments about the presence of defects, missing functionality, or weakness of the system. If you put a comment in your scripts that include certain keywords such as 'TODO', 'FIXME', and 'HACK', along with putting bug information in comments then it may reveal about missing functionalities of your system. However, these keywords make a comment 'suspicious' i.e., indicating missing functionality about the system.

Table 2. Mapping of Security Smell and Corresponding IaC Tool

Security Smell	Tool	Corresponding CWE
Admin by default	Chef, Puppet	CWE-250: Execution with Unnecessary Privileges
Empty password	Ansible, Puppet	CWE-258: Empty Password in Configuration File
Hard-coded secret	Ansible, Chef, Puppet	CWE-798: Use of Hard-coded Credentials
Invalid IP address binding	Ansible, Chef, Puppet	CWE-284: Improper Access Control
Missing Default in Case	Chef	CWE-478: Missing Default Case in Switch Statement
No Integrity Check	Ansible, Chef	CWE-353: Missing Support for Integrity Check
Suspicious comment	Ansible, Chef, Puppet	CWE-546: Suspicious Comment
Use of HTTP Without TLS	Ansible, Chef, Puppet	CWE-319: Cleartext Transmission of Sensitive Information
Use of Weak Crypto. Algorithm	Ansible, Puppet	CWE-326: Inadequate Encryption Strength

Use of HTTP Without TLS: This smell is the recurring pattern of using HTTP without the Transport Layer Security (TLS). Such use makes the communication between two entities less secure, as without TLS, use of HTTP is susceptible to man-in-the-middle attacks [13]. For example, if you connect with your MySQL server without TLS then the connection between your system with MySQL server is not private and secure because any third party can listen and modify information on this communication.

Use of Weak Crypto. Algorithms: This smell is the recurring pattern of using weak cryptography algorithms, such as MD5 and SHA-1 for encryption purposes. When weak algorithms, such as MD5 and SHA1, are used for hashing that may not lead to a breach, but using MD5 for password setup may.

How frequently do security smells appear?

As the next step, we want to identify if these security smells are prevalent in the OSS domain. By doing such empirical analysis we can not only understand the current state of security smell prevalence in IaC scripts, but also create benchmarks that practitioners can leverage. We conduct quantitative analysis by first constructing a static analysis tool, and then applying the tool on OSS repositories with IaC scripts.

Methodology

Static Analysis Tool: We construct a tool called Security Linter for Infrastructure as Code (SLIC), to automatically identify security smells for Ansible, Chef, and Puppet. SLIC uses a set of rules to identify security smells in Ansible, Chef, and Puppet scripts. The benefit of these rules is that practitioners can build their own tools that they are using for configuration and automated infrastructure management within their organization. The source code of the tool is available online [8], [11].

Repository collection: We apply a systematic filtering criteria to filter out repositories needed for frequency analysis. Summary attributes of the collected repositories are available in Table 3. All 1,094 repositories used for automated analysis were downloaded on April 2019 by cloning the master branches.

Metrics: First, we apply SLIC to determine the security smell occurrences for each script. Second, we calculate two metrics described below:

- ***Smell Density***: We use smell density to measure the frequency of a security smell x, for every 1000 lines of code (LOC). We measure smell density using Equation 1.

$$\text{Smell Density }(x) = \frac{\text{Total occurrences of } x}{\text{Total line count for all scripts}/1000} \tag{1}$$

- ***Proportion of Scripts (Script%)***: We use the metric 'Proportion of Scripts' to quantify how many scripts have at least one security smell. This metric refers to the percentage of scripts that contain at least one occurrence of smell x.

The two metrics characterize the frequency of security smells differently. The smell density metric is more granular, and focuses on the content of a script as measured by how many smells occur for every 1000 LOC. The proportion of scripts metric is less granular and focuses on the existence of at least one of the seven security smells for all scripts.

Findings

To quantify frequency of security smells in IaC scripts we collect 14,253 Ansible, 36,070 Chef, and 10,774 Puppet scripts respectively, collected from 365, 449, and 280 repositories. We

Table 3. Summary Attributes of the Datasets

	Ansible		Chef		Puppet	
Attribute	**GH**	**OST**	**GH**	**OST**	**GH**	**OST**
Repository Count	349	16	438	11	219	61
Total File Count	498,752	4,487	126,958	2,742	72,817	12,681
Total Script Count	13,152	1,101	35,132	938	8,010	2,764
Tot. LOC (IaC Scripts)	602,982	52,239	1,981,203	63,339	424,184	214,541

observe our identified security smells to exist across all datasets. For Ansible, in our GitHub and Openstack datasets we observe respectively 25.3% and 29.6% of the total scripts to contain at least one of the six identified security smells. For Chef, in our GitHub and Openstack datasets we observe respectively 20.5% and 30.4% of the total scripts to contain at least one of the eight identified security smells. For Puppet, in GitHub and Openstack datasets we observe proportion of scripts to be respectively, 29.3% and 32.9%. Hard-coded secret is the most prevalent security smell with respect to occurrences, smell density, and proportion of scripts contain hard-coded secrets. A complete breakdown of frequency-related findings for Ansible, Chef, and Puppet is respectively presented in Tables 4, 5, and 6.

Occurrences: The occurrences of the security smells are presented in the 'Occurrences' column of Table 4 for all six datasets. In Table 4 'N/A' indicates the security smell category is not applicable for the tool. For example, admin be default is not applicable for Ansible, as the category was not identified during qualitative analysis. The 'Combined' row presents the total smell occurrences. In the case of Ansible scripts, we observe 18,353 occurrences of security smells, and for Chef, we observe 28,247 occurrences of security smells. For Puppet we observe 17,756 occurrences of security smells. For Ansible, we identify 15,131 occurrences of hard-coded secrets, of which 55.9%, 37.0%, and 7.1% are respectively, hard-coded keys, user names, and passwords. For Chef, we identify 15,363 occurrences of hard-coded secrets, of which 47.0%, 8.9%, and 44.1% are respectively, hard-coded keys, user names, and passwords. For Puppet, we identify 14,444 occurrences of hard-coded secrets, of which 68.6%, 22.9%, and 8.5% are respectively, hard-coded keys, user names, and passwords.

Smell Density: In Table 5, we report the smell density for all datasets. The 'Combined' row presents the smell density for each dataset when all seven security smell occurrences are considered. For all six datasets, we observe the dominant security smell to be 'Hard-coded secret'. In Table 5 'N/A' indicates the security smell category is not applicable for a tool. For example, 'admin by default' is not applicable for Ansible as the smell category was not identified as part of our manual analysis.

Proportion of Scripts (Script%): In Table 6, we report the proportion of scripts (Script %) values for each of the six datasets. The 'Combined' row represents the proportion of scripts in which at least one of the identified smells appear. In Table 6 'N/A' indicates the security smell category is not applicable for a tool, e.g., 'admin by default' is not applicable for Ansible.

Our findings related to security smell frequency can be summarized as following:

- Approximately, 17.9%∼32.9% of IaC scripts in our dataset include at least one category of security smell.
- For every 1,000 lines of IaC code, security smells appear in 13.3∼31.5 LOC in IaC scripts.
- With respect to script proportion and smell density, the most frequently occurring security smell category is hard-coded secret. In our datasets, 6.8%∼24.8% of the collected scripts include at least one hard-coded secret. Furthermore, for every 1,000 lines of code for IaC scripts, 7.1%∼25.6 hard-coded secrets to appear. Hard-coded passwords are common: on average, 1 hard-coded passwords appear in 7 IaC scripts.
- Considering script proportion and smell density, the use of weak cryptography algorithms, is less than 1.0 across all datasets.

What do practitioners think about identified security smells?

Now that we have empirical evidence that shows security smells are rampant in OSS IaC

Table 4. Smell Occurrences for Ansible, Chef, and Puppet scripts

	Ansible		Chef		Puppet	
Smell Name	**GH**	**OST**	**GH**	**OST**	**GH**	**OST**
Admin by default	N/A	N/A	301	61	52	35
Empty password	298	3	N/A	N/A	136	21
Hard-coded secret	14,409	722	14,160	1,203	10,892	3,552
Missing default in switch	N/A	N/A	953	68	N/A	N/A
No integrity check	194	14	2,249	132	N/A	N/A
Suspicious comment	1,421	138	3,029	161	758	305
Unrestricted IP address	129	7	591	19	188	114
Use of HTTP without TLS	934	84	4,898	326	1,018	460
Use of weak crypto algo.	N/A	N/A	94	2	177	48
Combined	17,385	968	26,275	1,972	13,221	4,535

Table 5. Smell Density for Ansible, Chef, and Puppet scripts

	Ansible		Chef		Puppet	
Smell Name	**GH**	**OST**	**GH**	**OST**	**GH**	**OST**
Admin by default	N/A	N/A	0.1	0.9	0.1	0.1
Empty password	0.49	0.06	N/A	N/A	0.3	0.1
Hard-coded secret	23.9	13.8	7.1	19.0	25.6	16.5
Missing default in switch	N/A	N/A	0.5	1.0	N/A	N/A
No integrity check	0.3	0.2	1.1	2.1	N/A	N/A
Suspicious comment	2.3	2.6	1.5	2.5	1.7	1.4
Unrestricted IP address	0.2	0.1	0.3	0.3	0.4	0.5
Use of HTTP without TLS	1.5	1.6	2.4	5.1	2.4	2.1
Use of weak crypto algo.	N/A	N/A	0.05	0.03	0.4	0.1
Combined	28.8	18.5	13.3	31.5	25.3	29.6

Table 6. Proportion of Scripts With At Least One Smell for Ansible, Chef, and Puppet scripts

	Ansible		Chef		Puppet	
Smell Name	**GH**	**OST**	**GH**	**OST**	**GH**	**OST**
Admin by default	N/A	N/A	0.3	2.1	0.6	1.1
Empty password	1.1	0.2	N/A	N/A	1.4	0.5
Hard-coded secret	19.2	22.4	6.8	15.9	21.9	24.8
Missing default in switch	N/A	N/A	2.5	6.5	N/A	N/A
No integrity check	1.1	1.0	3.6	3.8	N/A	N/A
Suspicious comment	6.3	8.0	6.6	9.3	5.9	7.2
Unrestricted IP address	0.5	0.4	1.1	1.0	1.7	2.9
Use of HTTP without TLS	3.7	3.0	4.9	6.9	6.3	8.5
Use of weak crypto algo.	N/A	N/A	0.2	0.1	0.9	0.5
Combined	25.3	29.6	20.5	30.4	29.3	32.9

scripts, we wanted to get feedback from OSS practitioners for a random subset of the identified security smells. We provide our methodology and results below:

Methodology

We gather feedback using bug reports on how practitioners perceive the identified security smells. We apply the following procedure:

First, we randomly select 500 occurrences of security smells for each of Ansible, Chef, and Puppet scripts. **Second**, we post a bug report for each occurrence, describing the following items: smell name, brief description, related CWE, and the script where the smell occurred. **Third**, we determine a practitioner to agree with a security smell occurrence if (i) the practitioner replies to the submitted bug report explicitly saying the practitioner agrees; or (ii) the practitioner fixes the security smell occurrence in the specified script as determined by re-running SLIC on IaC scripts, for which we submitted bug reports. If the security smell does not exist in the script of interest, then we determine the smell to be fixed. We report disagreements if the practitioner (i) closes the issue report without discussion; or (ii) explicitly reports the identified security smell instances as a false positive or irrelevant. If no actions are taken related to agreement or disagreement, then we report 'no response'.

Findings

In the case of Ansible scripts, we observe an agreement of 82.7% for 29 smell occurrences. For Chef scripts, we observe an agreement of 55.5% for 54 smell occurrences. In the case of Puppet scripts, we observe an agreement of 63.4% for 104 smell occurrences. The percentage of smells to which practitioners agreed to be fixed for Ansible, Chef, and Puppet is respectively, presented in Figures 1, 2, and 3. For each of the figures, the y-axis represents each smell name followed by the occurrence count. For example, according to Figure 1, for 4 occurrences of 'Use of HTTP without TLS' (HTTP.USG), we observe 100% agreement for Ansible scripts.

Reasons for Practitioner Agreements: In their response, practitioners provided reasoning on why these smells appeared. For one occurrence of 'HTTP without TLS' in a Chef script, one practitioner suggested availability of a HTTPS endpoint saying: "*In this case, I think it was just me being a bit sloppy: the HTTPS endpoint is available so I should have used that to download RStudio packages from the start*". For an occurrence of hard-coded secret in an Ansible script one practitioner agreed stating possible solutions: "*I agree that it [hard-coded secret] could be in an Ansible vault or something dedicated to secret storage.*". Upon acceptance of the smell occurrences, practitioners also suggested how these smells can be mitigated. For example, for an occurrence of 'Unrestricted IP Address' in a Puppet script, one practitioner stated:"*I would accept a pull request to do a default of 127.0.0.1*".

Reasons for Practitioner Disagreements: We observe practitioners to value development context when disagreeing with security smell occurrences. For example, a hard-coded password may not seem to have security implications for practitioners if the hard-coded password is used for testing purposes. One practitioner disagreed stating "*the code in question is an integration test. The username and password is not used anywhere else so this should be no issue.*". These anecdotal evidence suggests that while developing IaC scripts practitioners may only be considering their own development context, and not realizing how another practitioner may perceive use of these security smells as an acceptable practice. For one occurrence of 'HTTP Without TLS' in a Puppet script one practitioner disagreed stating "*It's using http on localhost, what's the risk?*".

The above-mentioned statements from disagreeing practitioners also suggest a lack of security awareness, e.g. if a developer comes across the script of interest mentioned in the above-mentioned paragraph, the developer may perceive the use of hard-coded passwords to be an acceptable practice, potentially propagating the practice of hard-coded secrets. Another practitioner suggested that human intervention is also necessary when dealing with static analysis tool alerts: "*Human intervention is likely the best principled action, here.*"

What Did We Learn?

We learn that for IaC scripts security smells are prevalent. The most frequent security smell category is hard-coded secret. However, hard-coded secrets are not the only security smell category, other categories exist as well for example, use of weak cryptography algorithm and unrestricted IP address. Our findings show that well-known security smells that appear for existing languages, such as Java also appear for IaC.

One possible explanation can attribute to the prevalence of hard-coded secrets: the nature of IaC development i.e. if practitioners are setting up user accounts, then user names and passwords are likely to be specified in IaC scripts. Another possible explanation could be lack of credential tool availability and usage: may be practitioners do not have good enough tools to manage the passwords and usernames in IaC scripts. They also may not be aware of credential tool management that already exists for IaC, such as Vault. Lack of cybersecurity awareness could be another possible explanation. Practitioners who develop IaC scripts may not be aware of the consequences of security smells, and they may have contributed to the prevalence of security smells.

Implications for Practitioners

Our research can have implications for practitioners, which we describe below:

Look Before You Push

Practitioners can leverage our work to prioritize which coding patterns they should look for to mitigate security issues. If an organization uses code review as part of their IaC development process, then the team members can use the listed security smells as a guide. If a team does not use the code review process, the team may benefit from adopting a code review process,

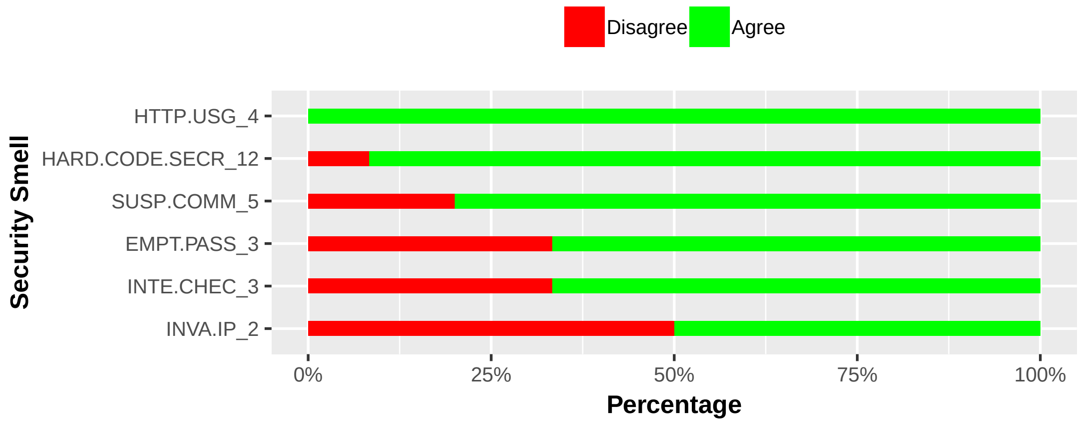

Figure 1. Feedback for 29 smell occurrences for Ansible. Practitioners agreed with 82.7% of the selected smell occurrences.

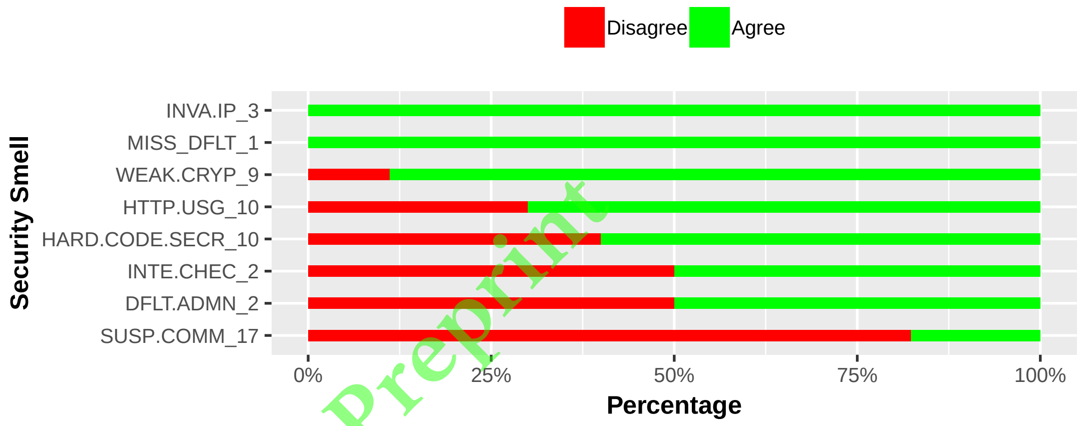

Figure 2. Feedback for 54 smell occurrences for Chef. Practitioners agreed with 55.5% of the selected smell occurrences.

which will flag the security smells. Some of the identified security smells in IaC scripts are examples of security misconfigurations that have been attributed to cause large-scale breaches, as happened for the Cloud Hospitality attack where over 10 million people's personal data was exposed due to a cloud-based misconfiguration [15]. Security-focused code review can help practitioners mitigate such attacks that can be potentially caused by IaC scripts as they are heavily used in automated configuration management.

Early Mitigation

For each category of security smell, we list mitigation techniques that developers can adopt while developing IaC scripts:

- **Admin by default**: We advise practitioners to create user accounts that have the minimum possible security privilege and use that account as default. Recommendations from Saltzer and Schroeder [14] may be helpful in this regard.
- **Empty password**: We advocate against storing empty passwords in IaC scripts. Instead, we suggest the use of strong passwords.
- **Hard-coded secret**: We suggest the following measures to mitigate hard-coded secrets:
  - use tools such as Vault to store secrets
  - scan IaC scripts to search for hard-coded secrets using tools such as CredScan and SLIC.

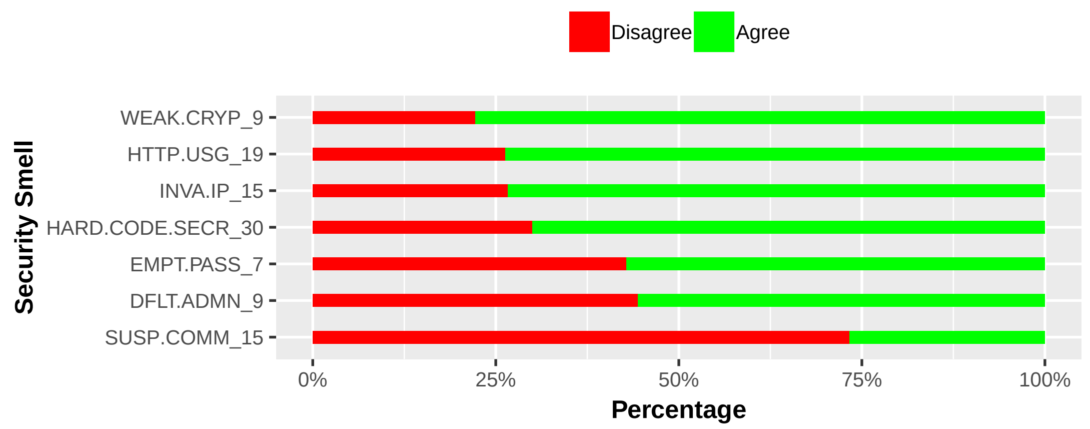

Figure 3. Feedback for 104 smell occurrences for Puppet. Practitioners agreed with 63.4% of the selected smell occurrences.

- **Invalid IP address binding**: To mitigate this smell, we advise programmers to allocate their IP addresses systematically based on which services and resources need to be provisioned. For example, incoming and outgoing connections for a database containing sensitive information can be restricted to a certain IP address and port.
- **Missing default in case statement**: We advise programmers to always add a default 'else' block so that unexpected input does not trigger events, which can expose information about the system.
- **No integrity check**: As IaC scripts are used to download and install packages and repositories at scale, we advise practitioners to always check downloaded content by computing hashes of the content or checking with GPG signatures.
- **Suspicious comment**: We acknowledge that in OSS development, programmers may be introducing suspicious comments to facilitate collaborative development and to provide clues on why the corresponding code changes are made.
- **Use of HTTP without TLS**: We advocate companies to adopt the HTTP with TLS by leveraging resources provided by tool vendors. We advocate for better documentation and tool support so that programmers do not abandon the process of setting up HTTP with TLS.
- **Use of Weak cryptography algorithms**: We advise programmers to use cryptography algorithms recommended by the National Institute of Standards and Technology [2] to mitigate this smell.

Knowledge is Power

We urge educators and researchers to pursue efforts in educating the IaC community. Our suggestions include conducting hands-on workshops and sharing tutorials on practitioner-oriented conferences. Currently, the field of DevOps and IaC has garnered a lot of interests amongst practitioners. IaC practitioners frequently organize workshop and conferences where they discuses their experiences and the challenges they face related to IaC. We urge researchers to participate in these venues and underline the importance of integrating cybersecurity in IaC. Practitioners are more receptive to hear from the experiences of other practitioners, and these venues could help in disseminating our findings to the practitioners.

Tools! We Need Better Tools!!

Teams use an automated pipeline to deploy their provisioned systems, may benefit from SLIC, as the tool is automated. Furthermore, developers can use the tool to identify security smells as they develop these scripts. If a continuous integration system is used, automated checks can be added to the CI system, so that security smells are first flagged, and integration of the submitted code changes are rejected, as long as the security smells are removed.

SLIC is susceptible to generate false positives and false negatives that can prevent wide-scale adoption amongst practitioners. As documented in our research, precision and recall of SLIC can be as lows as respectively, 72% [3] and 75% [12]. We are currently taking these limitations into account, and investing efforts on how to apply strategies, such as taint tracking so that fewer false positives and false negatives are generated by SLIC.

ACKNOWLEDGMENT

We thank the PASER group at Tennessee Technological University (TTU) for their valuable feedback. This research was partially funded by the National Science Foundation (NSF) award # 2026869 and the NSA Science of Security Lablet (award H98230-17-D-0080) at NC State University.

■ REFERENCES

1. Alison Rayome. Ansible overtakes chef and puppet as the top cloud configuration management tool. https://www.techrepublic.com/article/ansible-overtakes-chef-and-puppet/, 2019. [Online; accessed 18-January-2021].
2. Elaine Barker. Guideline for using cryptographic standards in the federal government: Cryptographic mechanisms. Technical report, National Institute of Standards and Technology, Gaithersburg, Maryland, August 2016.
3. F. Bhuiyan and A. Rahman. Characterizing co-located insecure coding patterns in infrastructure as code scripts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, 2020. to appear.
4. Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
5. MITRE. CWE-Common Weakness Enumeration. https://cwe.mitre.org/index.html, 2018. [Online; accessed 02-July-2020].
6. National Institute of Standards and Technology. Security and privacy controls for federal information systems and organizations. https://www.nist.gov/publications/, 2014. [Online; accessed 04-July-2020].
7. Puppet. Nyse and ice: Compliance, devops and efficient growth with puppet enterprise. Technical report, Puppet, April 2018.
8. A. Rahman, C. Parnin, and L. Williams. The Seven Sins: Security Smells in Infrastructure as Code Scripts. 1 2019.
9. Akond Rahman, Effat Farhana, and Laurie Williams. The 'as code' activities: Development anti-patterns for infrastructure as code. *Empirical Softw. Engg.*, 2020. to appear, pre-print: https://arxiv.org/pdf/2006.00177.pdf.
10. Akond Rahman, Chris Parnin, and Laurie Williams. The seven sins: Security smells in infrastructure as code scripts. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 164–175, Piscataway, NJ, USA, 2019. IEEE Press.
11. Akond Rahman, M. Rahman, Chris Parnin, and Laurie Williams. Dataset for Security Smells for Ansible and Chef Scripts Used in DevOps, 4 2020.
12. Akond Rahman, Md. Rayhanur Rahman, Chris Parnin, and Laurie Williams. Security smells in ansible and chef scripts: A replication study. *ACM Trans. Softw. Eng. Methodol.*, 2020. To appear. pre-print: https://arxiv.org/pdf/1907.07159.pdf.
13. Eric Rescorla. Http over tls. 2000.
14. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept 1975.
15. Tara Seals. Millions of hotel guests worldwide caught up in mass data leak. https://threatpost.com/millions-hotel-guests-worldwide-data-leak/161044/, 2020. [Online; accessed 18-January-2021].

Akond Rahman Akond Rahman is an assistant professor at Tennessee Tech University. His research interests include DevOps and Secure Software Development. He graduated with a PhD from North Carolina State University, an M.Sc. in Computer Science and Engineering from University of Connecticut, and a B.Sc. in Computer Science and Engineering from Bangladesh University of Engineering and Technology. He won the ACM SIGSOFT Doctoral Symposium Award at ICSE in 2018, the ACM SIGSOFT Distinguished Paper Award at ICSE in 2019, the CSC Distinguished Dissertation Award, and the COE Distinguished Dissertation Award from NC State in 2020. He actively collaborates with industry practitioners from IBM, Siemens, and others. To know more about his work, visit https://akondrahman.github.io/.

Laurie Williams Laurie Williams is a Distinguished University Professor in the Computer Science Department of the College of Engineering at North Carolina State University (NCSU). Laurie is a co-director of the NCSU Science of Security Lablet sponsored by the National Security Agency. Laurie's research focuses

on software security; agile software development practices and processes; software reliability, and software testing and analysis. Laurie is both an IEEE Fellow and an ACM Fellow. To know more about her work, visit https://collaboration.csc.ncsu.edu/laurie/.