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

Akond Rahman | Tennessee Technological University
Laurie Williams | North Carolina State University

In this article, 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 work will facilitate awareness among practitioners who use IaC.

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

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., they undergo software quality activities, such as linting and testing, and are maintained in a version control system.[9]

The IT industry has experienced benefits from the use of IaC. For example, the Intercontinental Exchange (ICE), which runs millions of financial transactions every day,[10] maintains 75% of its 20,000 servers by using IaC scripts.[7] The incorporation of IaC scripts has helped ICE to shorten the time needed to provision development environments, from 1 ~ 2 days to 21 min.[7] A wide variety of tools exists to implement IaC, e.g., Ansible, Chef, Puppet, SaltStack, and Terraform. According to a 2019 RightScale state-of-the-cloud report, which included a survey of 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 provide a pathway to a malicious user to attack a system, which motivated us to look into the problem in detail, taking a systematic approach. This article summarizes the findings from our recent work,[10, 12] in which we answer the following questions:

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

Digital Object Identifier 10.1109/MSEC.2021.3065190
Date of current version: 29 March 2021

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 practitioners can discuss and apply to IaC development. This information 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] It may not always have bad consequences, but it still deserves attention, as it may indicate trouble.[4] Our article focuses on identifying security smells. Those are recurring coding patterns that imply security weaknesses and warrant further inspection.[10]

Security smells are different from vulnerabilities, as they are coding patterns that point to flaws. A *vulnerability* is defined as a weakness that can be used to modify and access unintended data, interrupt proper software execution, and enable a party to perform actions without permission. Labeling a coding pattern as a vulnerability necessitates consideration of the application context, which is not relevant to security smells. They provide a mechanism for security-focused inspection that may or may not lead to a susceptibility but deserve attention, inspection, and, if necessary, mitigation. Let us consider a hard-coded secret as an example. A hard-coded secret, such as a hard-coded password, is relevant only if the password is used to log in to a software system. Understanding whether or not the password is being used for a login can be determined by knowing the 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 to a set of 3,339 Ansible, Chef, and Puppet scripts. A summary of the collected scripts is available in Table 1. The 29 repositories that we used were downloaded in November 2016 by cloning master branches. The Puppet-related repositories were collected from Mozilla, whereas the Ansible- and Chef-related repositories were gathered from Openstack. To identify security smells, we first apply a qualitative analysis technique called *open coding* on 1,726 scripts. Next, we map each smell to a possible security weakness defined by the Mitre Common Weakness Enumeration (CWE).[5]

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 provided in Table 2. It includes the following:

- *Administration by default*: This smell relates to the recurring pattern of specifying default users as administrative users. It can violate the "principle of least privilege" property,[6] which recommends that practitioners design and implement a system in a manner so that, by default, the least necessary amount access is provided to any entity.
- *Empty password*: This is the recurring pattern of using a string of length zero for a password. An empty password indicates weakness. It does not always lead to a security breach but can be more easily guessed. For example, if your My Structured Query Language (MySQL) server allows "root" access from a remote machine and the superuser "root" has an empty password, then anyone can connect to the server without an all-privilege password. An empty password differs from having no passwords. In secure-shell (SSH) key-based authentication, instead of passwords, public and private keys can be used. Our definition of *empty password* does not include the absence of 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, where 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 names and passwords, as configurations in IaC scripts. IaC scripts provide an opportunity to specify configurations for an entire system, such as setting up SSH keys for users and specifying authentications files (creating key-pair files for AWS). However, in the process, programmers can

Table 1. Summary statistics for collected scripts to determine security smells.

Language	Duration	Repository count	Organization	Script count
Ansible	February 2014–November 2016	16	Openstack	1,101
Chef	May 2011–November 2016	11	Openstack	855
Puppet	September 2014–November 2016	2	Mozilla	1,383

hard code these pieces of information into scripts. We consider three types of hard-coded secrets: passwords, user names, and private cryptography keys.

- *Unrestricted Internet Protocol (IP) address binding*: This is the recurring pattern of assigning the address 0.0.0.0 for a database server or a cloud service/instance. Binding to 0.0.0.0 may cause security concerns, as it can enable connections from every possible network. It can cause security problems since the server, service, and instance will be exposed to all IP addresses. For example, practitioners have reported how binding to 0.0.0.0 facilitated security problems for MySQL, Memcached (a cloud-based cache service), and Kibana (a 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 doing so may not be desirable overall.
- *Missing default in case statement*: This smell involves the recurring pattern of not handling all input combinations when implementing case-conditional logic. Because of this coding pattern, an attacker can guess a value that is not handled by case conditional-statements and trigger an error, which can provide unauthorized system information in terms of stack traces and faults.
- *No integrity check*: This concerns the recurring pattern of not checking downloaded repository content via checksums and GNU Privacy Guard (GPG) signatures. By not verifying integrity, a developer assumes that the content is secure and that it has not been corrupted by a potential attacker.
- *Suspicious comment*: This is the recurring pattern of putting information in comments about the presence of defects, missing functionality, and system weakness. Including certain keywords, such as "TODO," "FIXME," and "HACK," with bug information may reveal missing functionalities. These keywords make a comment "suspicious."
- *Using HTTP without Transport Layer Security (TLS)*: This makes communication between two entities less secure, as without TLS, HTTP is susceptible to man-in-the-middle attacks.[13] For example, if you connect to your MySQL server without TLS, the connection will not be private and secure because any third party can listen in and modify information.
- *Using weak cryptography algorithms*: This smell involves weak cryptography algorithms, such as message–digest algorithm 5 (MD5) and Secure Hash Algorithm 1, for encryption. Weak algorithms for hashing 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 whether these security smells are prevalent in the OSS domain. By doing such empirical analysis, we can understand the current state of security smell prevalence in IaC scripts and create benchmarks that practitioners can leverage. We conduct quantitative analysis by constructing a static tool and applying it to OSS repositories with IaC scripts.

Methodology

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

Table 2. Security smell mapping and corresponding IaC tools.

Security smell	Tool	Corresponding CWE
Administrator 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 cryptography algorithm	Ansible, Puppet	CWE-326: Inadequate encryption strength

IP: Internet Protocol; TLS: Transport Layer Security.

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

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

- *Smell density*: We use smell density to measure the frequency of a security smell x for every 1,000 lines of code (LoC). We measure smell density by using (1):

$$\text{Smell density}\,(x) = \frac{\text{Total occurrences of } x}{\frac{\text{Total line count for all scripts}}{1,000}}. \quad (1)$$

- *Proportion of scripts*: We use this metric to quantify the percentage of scripts with at least one occurrence of security smell x.

The two metrics characterize the frequency of security smells differently. Smell density is more granular and focuses on the content of a script as measured by how many smells occur for every 1,000 LOC. The proportion of scripts is less granular and focuses on the existence of at least one of the seven security smells for all scripts.

Findings

To quantify the frequency of security smells in IaC scripts, we collect 14,253 Ansible, 36,070 Chef, and 10,774 Puppet scripts from 365, 449, and 280 repositories, respectively. We observe that our identified security smells exist across all data sets. For Ansible, in our GitHub and Openstack data sets, we find that, respectively, 25.3 and 29.6% of the total scripts contain at least one of the six identified security smells. For Chef, in our GitHub and Openstack data sets, we see that, respectively, 20.5 and 30.4% of the total scripts contain at least one of the eight identified security smells. For Puppet, in GitHub and Openstack data sets, we note the 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 the proportion of scripts. Complete breakdowns of the frequency-related findings for Ansible, Chef, and Puppet are in Tables 4–6.

Occurrences. Security smell occurrences are enumerated in Table 4 for all six data sets. In Table 4 "N/A" indicates that a security smell category is not applicable for a tool. For example, administrator by default is not applicable for Ansible, as the category was not identified during qualitative analysis. In the case of Ansible scripts, we observe 18,353 occurrences of security smells; for Chef, we detect 28,247, and for Puppet, we note 17,756. For Ansible, we identify 15,131 occurrences of hard-coded secrets, of which 55.9, 37, and 7.1% are hard-coded keys, user names, and passwords, respectively. For Chef, we locate 15,363 occurrences, of which 47, 8.9%, and 44.1% are hard-coded keys, user names, and passwords, respectively. For Puppet, we detect 14,444 instances, of which 68.6, 22.9, and 8.5% are hard-coded keys, user names, and passwords, respectively.

Smell density. In Table 5, we report the smell density for all six data sets and observe the dominant security smell to be hard-coded secret. "N/A" indicates that a category is not applicable for a tool. For example, administrator by default is not pertinent to Ansible, as the category was not identified during our manual analysis.

Proportion of scripts. In Table 6, we detail the proportion of scripts for the six data sets. "N/A" indicates that a category is not applicable for a tool; e.g., administrator by default does not apply to Ansible.

Table 3. Summary attributes of the data sets.

	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
Total LoC (IaC scripts)	602,982	52,239	1,981,203	63,339	424,184	214,541

GH: GitHub; OS: OpenStack.

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

- Approximately 17.9 ~ 32.9% of the IaC scripts in our data set include at least one security smell category.
- For every 1,000 IaC LoC, security smells appear in 13.3 ~ 31.5 LoC in IaC scripts.
- With respect to script proportion and smell density, the most frequent security smell category is hard-coded secret. In our data sets, 6.8 ~ 24.8% of the collected scripts include at least one hard-coded secret. Furthermore, for every 1,000 LoC in IaC scripts, 7.1 ~ 25.6 hard-coded secrets appear. Hard-coded passwords are common: on average, one hard-coded password appears in every seven IaC scripts.
- Considering script proportion and smell density, the use of weak cryptography algorithms is less than one across all data sets.

What Do Practitioners Think About the Identified Security Smells?

Once we had empirical evidence showing that security smells are rampant in OSS IaC scripts, we wanted to get feedback from OSS practitioners about a random

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

	Ansible		Chef		Puppet	
Smell name	**GH**	**OST**	**GH**	**OST**	**GH**	**OST**
Administrator 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 cryptography algorithm	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**
Administrator 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	25.6	16.5
Missing default in switch	N/A	N/A	0.5	1	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 cryptography algorithm	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

subset of the identified issues. We provide our methodology and results in the following.

Methodology

We gathered feedback by using bug reports to understand how practitioners perceived the identified security smells. First, we randomly selected 500 occurrences of security smells for each Ansible, Chef, and Puppet script. Second, we posted a bug report for each occurrence, describing the following items: the smell name, a brief description, the related CWE, and the script where the smell occurred. Third, we determined that a practitioner agreed with a security smell occurrence if 1) he or she replied to a bug report and explicitly said that he or she concurred or 2) he or she fixed the security smell, as determined by rerunning SLIC. We reported disagreements if the practitioner 1) closed the issue report without discussion or 2) explicitly reported a security smell as a false positive or irrelevant. If no actions were taken, we reported "no response."

Findings

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

Table 6. The proportion of scripts with at least one smell for Ansible, Chef, and Puppet.

	Ansible		Chef		Puppet	
Smell name	GH	OST	GH	OST	GH	OST
Administrator 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	3.6	3.8	N/A	N/A
Suspicious comment	6.3	8	6.6	9.3	5.9	7.2
Unrestricted IP address	0.5	0.4	1.1	1	1.7	2.9
Use of HTTP without TLS	3.7	3	4.9	6.9	6.3	8.5
Use of weak cryptography algorithm	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

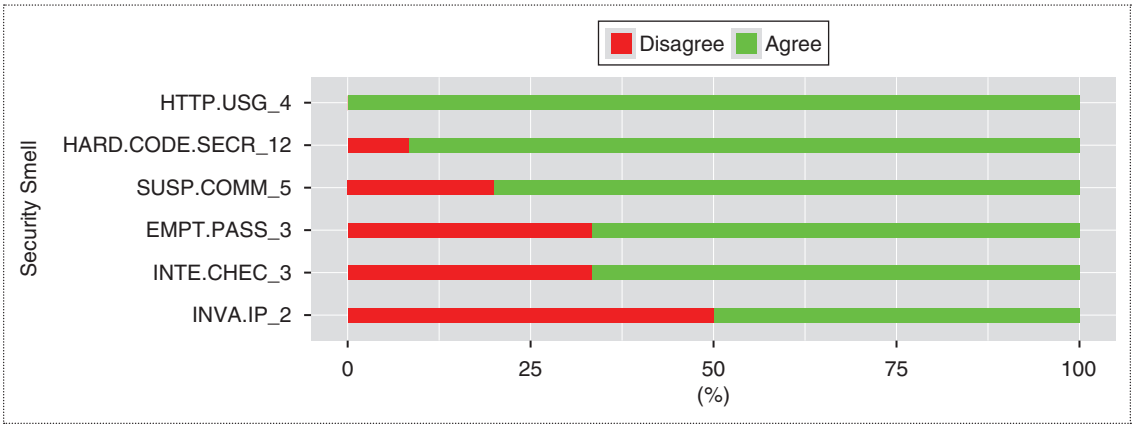

Figure 1. Feedback for 29 smell occurrences for Ansible. Practitioners agreed with 82.7% of the selected smell occurrences.

Reasons for practitioner agreement. In their responses, practitioners provided reasoning for why smells appeared. For one occurrence of HTTP without TLS in a Chef script, one practitioner suggested the availability of an 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 instance of hard-coded secret in an Ansible script, one practitioner agreed, stating possible solutions: "I agree that it [the hard-coded secret] could be in an Ansible vault or something dedicated to secret storage." When accepting a smell occurrence, practitioners also suggested how the issue could be mitigated. For example, for an incidence 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 disagreement. 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 if it is used for testing purposes. One practitioner stated that "the code in question is an integration test. The username and password are not used anywhere else, so this should be no issue." This anecdotal evidence suggests that while developing IaC scripts, practitioners may consider only their own development context and not realize how another practitioner may perceive the use of security smells as an acceptable practice. For one occurrence of HTTP without TLS in a Puppet script, one practitioner said: "It's using HTTP on local host; what's the risk?"

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

What Did We Learn?

IaC script security smells are prevalent, with the most common category being hard-coded secret. Our

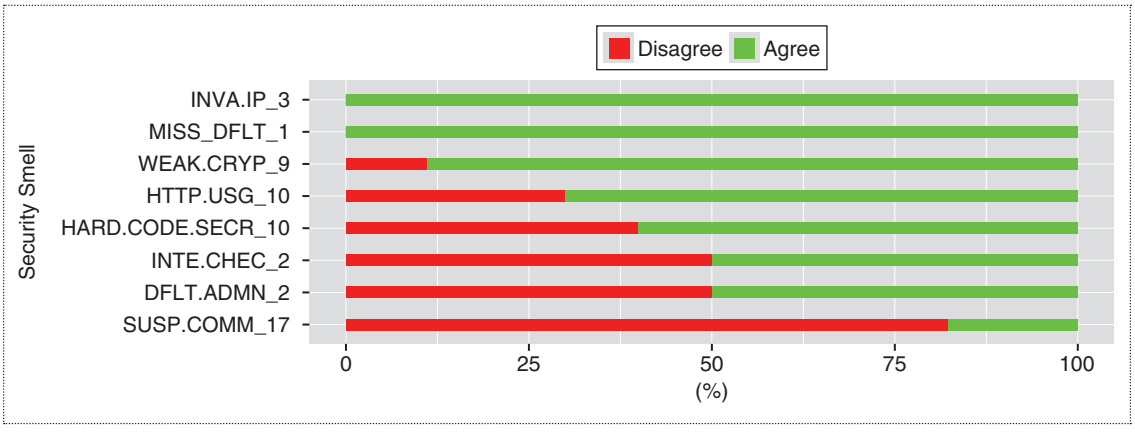

Figure 2. Feedback for 54 smell occurrences for Chef. Practitioners agreed with 55.5% of the selected smell occurrences.

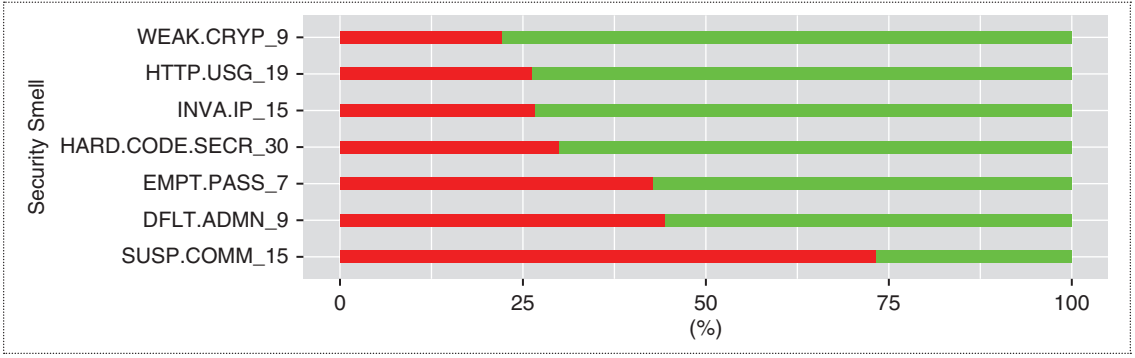

Figure 3. Feedback for 104 smell occurrences for Puppet. Practitioners agreed with 63.4% of the selected smell occurrences.

findings show that well-known security smells that appear for existing languages, such as Java, also appear for IaC. One possible explanation attributes the prevalencve of hard-coded secrets to the nature of IaC development; i.e., if practitioners set up user accounts, user names and passwords are likely to be specified in IaC scripts. Another possible explanation could concern the lack of credential tool availability and usage: perhaps practitioners do not have adequate tools to manage passwords and usernames in IaC scripts. They also may not be aware of credential tool management that already exists for IaC, such as Vault. A lack of cybersecurity awareness could be another possible explanation. Practitioners who develop IaC scripts may not know about the consequences of security smells, and they may have contributed to the prevalence of such issues.

Implications for Practitioners

Our research can have implications for practitioners, which we describe in the following.

Look Before You Push

Practitioners can leverage our work to prioritize the coding patterns they should look for to mitigate security issues. If an organization uses code review as part of its IaC development process, team members can use the security smells listed here as a guide. If a team does not use code reviews, it may benefit from adopting such a process to flag security smells. Some of the identified security smells in IaC scripts are examples of security misconfigurations that have caused large-scale breaches, as happened during the Cloud Hospitality attack, where more than 10 million people's personal data were exposed due to a cloud-based misconfiguration.[8] Security-focused code review can help practitioners mitigate the risk of attacks that could be caused by IaC scripts, which are heavily used in automated configuration management.

Early Mitigation

For each security smell category, we list mitigation techniques that developers can adopt while creating IaC scripts, as in the following:

- *Administrator by default*: We advise practitioners to create user accounts that have the minimum possible security privilege and use those accounts as defaults. 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 offer 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 by using tools such as CredScan and SLIC.
- *Invalid IP address binding*: To mitigate this smell, we advise programmers to systematically allocate their IP addresses based on services and resources that need to be provisioned. For example, incoming and outgoing connections for a database containing sensitive information can be restricted to certain IP addresses and ports.
- *Missing default in case statement*: We advise programmers to always add a default "else" block so that unexpected inputs do not trigger events that can expose information about a system.
- *No integrity check*: Since IaC scripts are used to download and install packages and repositories at scale, we advise practitioners to always check content by computing hashes and checking GPG signatures.
- *Suspicious comment*: We acknowledge that in OSS development, programmers may introduce suspicious comments to facilitate collaborative development and provide clues to why corresponding code changes are made.
- *Use of HTTP without TLS*: We advocate for companies to adopt HTTP with TLS by leveraging resources provided by tool vendors. We encourage 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 instructors and researchers to pursue efforts to educate the IaC community. Our suggestions include conducting hands-on workshops and sharing tutorials at practitioner-oriented conferences. Currently, the field of DevOps and IaC has garnered a lot of interest. IaC practitioners frequently organize workshops and conferences to discuses their experiences and the challenges they face. We urge researchers to participate in conferences and underline the importance of integrating cybersecurity in IaC. Practitioners are more receptive to the experiences of their peers, and these venues could help to disseminate our findings to them.

Teams that use automated pipelines to deploy their provisioned systems may benefit from SLIC, as the tool is automated. Furthermore, developers can use SLIC

to identify security smells as they develop these scripts. If a continuous integration system is employed, automated checks can be added to the continuous integration system so that security smells are flagged and the integration of submitted code changes is rejected until the issues are resolved. SLIC is susceptible to generating false positives and false negatives that can prevent wide-escale adoption among practitioners. As documented in our research, the SLIC precision and recall can be as lows as 72[3] and 75%,[12] respectively. We are taking these limitations into account and investing effort to apply strategies such as taint tracking so that there are fewer false positives and false negatives. ■

Acknowledgments

We thank the Practical and Actionable Software Engineering Research group, Tennessee Technological University, for its valuable feedback. This research was partially funded by the National Science Foundation, under grant 2026869, and the National Security Administration Science of Security Lablet (grant H98230-17-D-0080) at North Carolina State University.

References

1. A. Rayome. "Ansible overtakes chef and puppet as the top cloud configuration management tool," Techrepublic, 2019. https://www.techrepublic.com/article/ansible-overtakes-chef-and-puppet/ (accessed Jan. 18, 2021).
2. E. Barker, "Guideline for using cryptographic standards in the federal government: Cryptographic mechanisms," National Inst. Standards and Technol., Gaithersburg, MD, Tech. Rep. SP 800-175B, Aug. 2016.
3. F. Bhuiyan and A. Rahman, "Characterizing co-located insecure coding patterns in infrastructure as code scripts," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. Workshop (ASEW)*, 2020, pp. 27–32. doi: 10.1145/3417113.3422154.
4. M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*. New York: Addison-Wesley, 1999.
5. "CWE-common weakness enumeration," MITRE, 2018. https://cwe.mitre.org/index.html (accessed July 2, 2020).
6. Security and privacy controls for federal information systems and organizations, National Inst. of Standards and Technol., Gaithersburg, MD. https://www.nist.gov/publications/ (accessed July 4, 2020).
7. "NYSE and ICE: Compliance, DevOps and efficient growth with Puppet enterprise," Puppet, Tech. Rep., Apr. 2018. https://media.webteam.puppet.com/uploads/2019/11/puppet-CS-NYSE-ICE.pdf
8. T. Seals. "Millions of hotel guests worldwide caught up in mass data leak." Threatpost.com, 2020. https://threatpost.com/millions-hotel-guests-worldwide-data-leak/161044/ (accessed Jan. 18, 2021).
9. A. Rahman, E. Farhana, and L. Williams, "The 'as code' activities: Development anti-patterns for infrastructure as code," *Empirical Softw. Eng.*, 2020. [Online]. Available: https://arxiv.org/pdf/2006.00177.pdf
10. A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *Proc. 41st Int. Conf. Softw. Eng. (ICSE '19)*, 2019, pp. 164–175.
11. A. Rahman, M. Rahman, C. Parnin, and L. Williams, "Dataset for security smells for ansible and chef scripts used in DevOps," Apr. 2020. doi: 10.6084/m9.figshare.8085755.v2.
12. A. Rahman, M. Rayhanur Rahman, C. Parnin, and L. Williams, "Security smells in ansible and chef scripts: A replication study," *ACM Trans. Softw. Eng. Methodol.*, 2020. [Online]. Available: https://arxiv.org/pdf/1907.07159.pdf
13. E. Rescorla, "HTTP over TLS," 2000.
14. J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proc. IEEE*, vol. 63, no. 9, pp. 1278–1308, Sept. 1975. doi: 10.1109/PROC.1975.9939.

Akond Rahman is an assistant professor at Tennessee Technological University, Cookeville, Tennessee, 38505, USA. His research interests include DevOps and secure software development. Rahman received a Ph.D. in computer science and engineering from North Carolina State University (NCSU). He won the Association for Computing Machinery (ACM) Special Interest Group on Software Engineering (SIGSOFT) Doctoral Symposium Award at the 2018 International Conference on Software Engineering (ICSE), the ACM SIGSOFT Distinguished Paper Award at ICSE 2019, and the NCSU Department of Computer Science Distinguished Dissertation Award and the NCSU College of Engineering Distinguished Dissertation Award in 2020. Contact him at arahman@tntech.edu.

Laurie Williams is a Distinguished University Professor in the Department of Computer Science, College of Engineering, North Carolina State University (NCSU), Raleigh, North Carolina, 27695, USA. She is a codirector of the NCSU Science of Security Lablet sponsored by the U.S. National Security Agency. Her research interests include software security, agile software development practices and processes, software reliability, and software testing and analysis. Williams received a Ph.D. from the University of Utah. She is a Fellow of IEEE and the Association for Computing Machinery. Contact her at williams@csc.ncsu.edu.