The Seven Sins: Security Smells in Infrastructure as Code Scripts

Akond Rahman, Chris Parnin, and Laurie Williams
North Carolina State University, Raleigh, North Carolina
Email: aarahman@ncsu.edu, cjparnin@ncsu.edu, williams@csc.ncsu.edu

Abstract*—Practitioners use infrastructure as code (IaC) scripts to provision servers and development environments. While developing IaC scripts, practitioners may inadvertently introduce security smells. Security smells are recurring coding patterns that are indicative of security weakness and can potentially lead to security breaches. *The goal of this paper is to help practitioners avoid insecure coding practices while developing infrastructure as code (IaC) scripts through an empirical study of security smells in IaC scripts.

We apply qualitative analysis on 1,726 IaC scripts to identify seven security smells. Next, we implement and validate a static analysis tool called *S*ecurity *L*inter for *I*nfrastructure as *C*ode scripts (*SLIC*) to identify the occurrence of each smell in 15,232 IaC scripts collected from 293 open source repositories. We identify 21,201 occurrences of security smells that include 1,326 occurrences of hard-coded passwords. We submitted bug reports for 1,000 randomly-selected security smell occurrences. We obtain 104 responses to these bug reports, of which 67 occurrences were accepted by the development teams to be fixed. We observe security smells can have a long lifetime, e.g., a hard-coded secret can persist for as long as 98 months, with a median lifetime of 20 months.

***Index Terms*—devops, infrastructure as code, security smell**

I. Introduction

Infrastructure as code (IaC) scripts help practitioners to provision and configure their development environment and servers at scale [1]. IaC scripts are also known as configuration scripts [2] [1] or configuration as code scripts [1] [3]. Commercial IaC tool vendors, such as Chef [1] and Puppet [4], provide programming syntax and libraries so that programmers can specify configuration and dependency information as scripts.

Fortune 500 companies [2], such as Intercontinental Exchange (ICE) [3], use IaC scripts to maintain their development environments. For example, ICE, which runs millions of financial transactions daily [4], maintains 75% of its 20,000 servers using IaC scripts [5]. The use of IaC scripts has helped ICE decrease the time needed to provision development environments from 1∼2 days to 21 minutes [5].

However, IaC scripts can be susceptible to security weakness. Let us consider Figure 1 as an example. In Figure 1, we present a Puppet code snippet extracted from the 'aeolus-configure' open source software (OSS) repository [5]. In this code snippet, we observe a hard-coded password using the 'password' attribute. A hard-coded string 'v23zj59an' is assigned as password for user 'aeolus'. Hard-coded passwords in software artifacts is considered as a software security weakness ('CWE-798: Use of Hard-coded Credentials') by Common Weakness Enumerator (CWE) [6]. According to CWE [6], "If hard-coded passwords are used, it is almost certain that malicious users will gain access to the account in question".

IaC scripts similar to Figure 1, which contain hard-coded credentials or other *security smells*, can be susceptible to security breaches. Security smells are recurring coding patterns that are indicative of security weakness. A security smell does not always lead to a security breach, but deserves attention and inspection. Existence and persistence of these smells in IaC scripts leave the possibility of another programmer using these smelly scripts, potentially propagating use of insecure coding practices. We hypothesize through systematic empirical analysis, we can identify security smells and the prevalence of the identified security smells.

The goal of this paper is to help practitioners avoid insecure coding practices while developing infrastructure as code (IaC) scripts through an empirical study of security smells in IaC scripts.

We answer the following research questions:

- **RQ1**: What security smells occur in infrastructure as code scripts? (Section III)
- **RQ2**: How frequently do security smells occur in infrastructure as code scripts? (Section VI)
- **RQ3**: What is the lifetime of the identified security smell occurrences for infrastructure as code scripts? (Section VI)
- **RQ4**: Do practitioners agree with security smell occurrences? (Section VI)

We answer our research questions by analyzing IaC scripts collected from OSS repositories. We apply qualitative analysis [7] on 1,726 scripts to determine security smells. Next, we construct a static analysis tool called **S**ecurity **L**inter for **I**nfrastructure as **C**ode scripts (**SLIC**) to automatically identify the occurrence of these security smells in 15,232 IaC scripts collected by mining 293 OSS repositories from four sources: Mozilla [6], Openstack [7], Wikimedia Commons [8], and GitHub [9].

[1] https://www.chef.io/chef/
[2] http://fortune.com/fortune500/list/
[3] https://www.theice.com/index
[4] https://www.theice.com/publicdocs/ICE_at_a_glance.pdf
[5] https://github.com/aeolusproject/aeolus-configure
[6] https://hg.mozilla.org/
[7] https://git.openstack.org/cgit
[8] https://gerrit.wikimedia.org/r/
[9] https://github.com/

Preprint

```
postgres::user{"aeolus":                                  Hard-coded password
              password => "v23zj59an",
              roles => "CREATEDB",
              require => Service["postgresql"] }
```

Fig. 1: An example IaC script with hard-coded password.

We calculate smell density and lifetime of each identified smell occurrence in the collected IaC scripts. We submit bug reports for 1,000 randomly-selected smell occurrences to assess the relevance of the identified security smells.

Contributions: We list our contributions as following:

- A derived list of seven security smells with definitions;
- An evaluation of how frequently security smells occur in IaC scripts along with their lifetime;
- An empirically-validated tool (SLIC) that automatically detects occurrences of security smells; and
- An evaluation of how practitioners perceive the identified security smells.

We organize the rest of the paper as following: we provide background information with related work discussion in Section II. We describe the methodology and the definitions of seven security smells in Section III. We describe the methodology to construct and evaluate SLIC in Section IV. In Section V, we describe the methodology for our empirical study. We report our findings in Section VI, followed by a discussion in Section VII. We describe limitations in Section VIII, and conclude our paper in Section IX.

II. Background and Related Work

We provide background information with related work discussion in this section.

A. Background

IaC is the practice of automatically defining and managing network and system configurations and infrastructure through source code [1]. Companies widely use commercial tools such as Puppet, to implement the practice of IaC [1] [8] [9]. For example, using IaC scripts application deployment time for Borsa Istanbul, Turkey's stock exchange, reduced from ∼10 days to an hour [10]. With IaC scripts Ambit Energy increased their deployment frequency by a factor of 1,200 [11].

Typical entities of Puppet include manifests [4]. Manifests are written as scripts that use a .pp extension. In a single manifest script, configuration values can be specified using variables and attributes. Puppet provides the utility 'class' that can be used as a placeholder for the specified variables and attributes. For better understanding, we provide a sample Puppet script with annotations in Figure 2. For attributes configuration values are specified using the '=>' sign, whereas, for variables, configuration values are provided using the '=' sign. A single manifest script can contain one or multiple attributes and/or variables. In Puppet, variables store values and have no relationship with resources. Attributes describe the desired state of a resource. Similar to general purpose programming languages, code constructs such as

```
#This is an example Puppet script                          Comment
class (`example'                                  Attribute 'token'
){
   token => 'XXXYYYZZZ'                                  Variable '$os_name'
   $os_name = 'Windows'                                  Case conditional
   case $os_name{                                        Calling function
                                                         'getAuth()'
     'Solaris': { auth_protocol => `http' }
     'CentOS': { auth_protocol => getAuth() }
     default: { auth_protocol => `https' }
   }
}
```

Fig. 2: Annotation of an example Puppet script.

functions/methods, comments, and conditional statements are also available for Puppet scripts.

B. Related Work

Prior research has investigated what categories of bad coding practices can have security consequences in non-IaC domains, such as Android applications and Java frameworks. Meng et al. [12] studied bad coding practices related to the security of Java Spring Framework in Stack Overflow, and reported 9 out of 10 SSL/TLS-related posts to discuss insecure coding practices. Fahl et al. [13] investigated inappropriate use of SSL/TLS protocols, such as, trusting all certificates and stripping SSL, for Android applications. Using MalloDroid, Fahl et al. [13] identified 8% of the studied 13,500 Android applications to inappropriately use SSL/TLS. Felt et al. [14] used Stowaway to study if Android applications follow the principle of least privilege. They reported 323 of the studied 900 Android applications to be over-privileged. Ghafari et al. [15] analyzed 70,000 Android applications to identify the frequency of security smells. They reported 50% of the studied applications to contain at least three security smells. Egele et al. [16] used a static analysis tool called Cryptolint to analyze if cryptography APIs are used inappropriately, for example, using constant encryption keys and using static seeds to seed pseudo-random generator function. Egele et al. [16] observed at least one inappropriate use for 88% of 11,748 Android applications.

The above-mentioned work highlights that other domains such as Android are susceptible to inappropriate coding practices that have security consequences. Yet, for IaC scripts we observe lack of studies that investigate coding practices with security consequences. For example, Sharma et al. [2], Schwarz [17], and Bent et al. [18], in separate studies investigated code maintainability aspects of Chef and Puppet scripts. Hanappi et al. [19] investigated how convergence of IaC scripts can be automatically tested, and proposed an automated model-based test framework. Jiang and Adams [8] investigated the co-evolution of IaC scripts and other software artifacts, such as build files and source code. They reported IaC scripts to experience frequent churn. Rahman and Williams [20] characterized defective IaC scripts using text mining and created prediction models using text feature met-

Preprint

rics. Rahman et al. [21] surveyed practitioners to investigate which factors influence usage of IaC tools.

III. Security Smells

We describe the methodology to derive security smells in IaC scripts, followed by the definitions and examples for the identified security smells.

A code smell is a recurrent coding pattern that is indicative of potential maintenance problems [22]. A code smell may not always have bad consequences, but still deserves attention, as a code smell may be an indicator of a problem [22]. Our paper focuses on identifying security smells. Security smells are recurring coding patterns that are indicative of security weakness. A security smell is different from a vulnerability, as a vulnerability always enables a security breach [23], whereas a security smell does not.

A. RQ1: What security smells occur in infrastructure as code scripts?

Data Collection: We collect 1,726 Puppet scripts that we use to determine the security smells. We collect these scripts from the master branches of 74 repositories, downloaded on July 30, 2017. We collect these 74 repositories from the three organizations Mozilla, Openstack and Wikimedia Commons. We use Puppet scripts to construct our dataset because Puppet is considered as one of the most popular tools for configuration management [8] [9], and has been used by companies since 2005 [24].

Qualitative Analysis: We first apply a qualitative analysis technique called descriptive coding [25] on 1,726 Puppet scripts to identify security smells. Next, we map each identified smell to a possible security weakness defined by CWE [6]. We select qualitative analysis because we can (i) get a summarized overview of recurring coding patterns that are indicative of security weakness; and (ii) obtain context on how the identified security smells can be automatically identified. We select the CWE to map each smell to a security weakness because CWE is a list of common software security weaknesses developed by the security community [6].

Figure 3 provides an example of our qualitative analysis process. We first analyze the code content for each IaC script and extract code snippets that correspond to a security weakness as shown in Figure 3. From the code snippet provided in the top left corner, we extract the raw text: '$db_user = 'root'. Next we generate the initial category 'Hard-coded user name' from the raw text '$db_user = 'root'' and '$vcenter_user = 'user''. Finally, we determine the smell 'Hard-coded secret' by combining initial categories. We combine these two initial categories, as both correspond to a common pattern of specifying user names and passwords as hard-coded secrets. Upon derivation we observe 'Hard-coded secret' to be related to 'CWE-798: Use of Hard-coded Credentials' and 'CWE-259: Use of Hard-coded Password' [6].

Our qualitative analysis process to identify seven security smells took 565 hours. The first author conducted the qualitative analysis.

Verification of CWE Mapping by Independent Raters: The first author derived the seven smells, and the derivation process is subject to the rater's judgment. We mitigate this limitation by recruiting two independent raters who are not authors of the paper. These two raters, with background in software security, independently evaluated if the identified smells are related to the associated CWEs. We provide the raters the smell names, one example of each smell, and the related CWEs for each smell. The two raters independently determined if each of the smells are related to the provided CWEs. Both raters mapped each of the seven security smells to the same CWEs. We observe a Cohen's Kappa score of 1.0 between raters.

B. Answer to RQ1: What security smells occur in infrastructure as code scripts?

Using our methodology we identify seven security smells, each of which we describe in this section. The names of the smells are presented alphabetically. Examples of each security smell is presented in Figure 4.

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 [26], which recommends practitioners to design and implement system in a manner so that by default the least amount of access necessary is provided to any entity. In Figure 4, two of the default parameters are '$power_username', and '$power_password'. If no values are passed to this script, then the default user will be 'admin', and can have full access. The smell is related with 'CWE-250: Execution with Unnecessary Privileges' [6].

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. The smell is similar to the weakness 'CWE-258: Empty Password in Configuration File' [6]. An empty password is different from using no passwords. In SSH key-based authentication, instead of passwords, public and private keys can be used [27]. 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. In Figure 4, we provide six examples of hard-coded secrets. We consider three types of hard-coded secrets: hard-coded passwords, hard-coded user names, and hard-coded private cryptography keys. Relevant weaknesses to the smell

Preprint

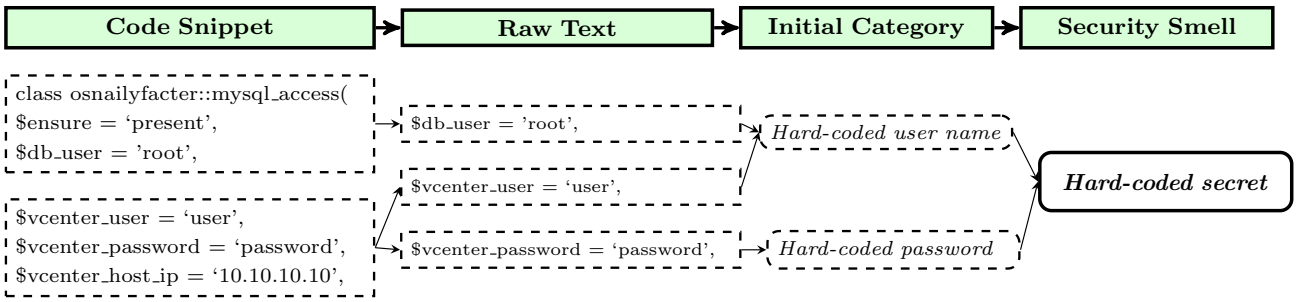

Fig. 3: An example of how we use qualitative analysis to determine security smells in IaC scripts.

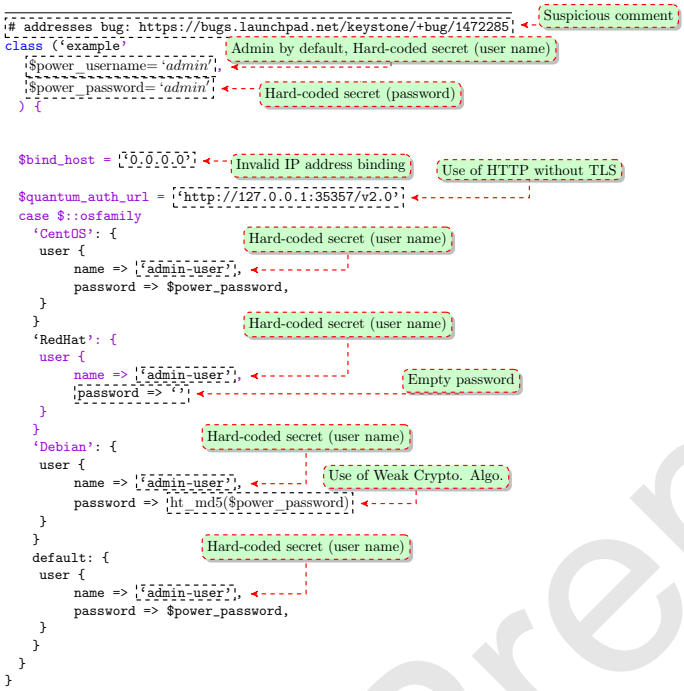

Fig. 4: An annotated script with all seven security smells. The name of each security smell is highlighted on the right.

are 'CWE-798: Use of Hard-coded Credentials' and 'CWE-259: Use of Hard-coded Password' [6]. For source code, practitioners acknowledge the existence of hard-coded secrets and advocate for tools such as CredScan [10] to scan source code.

We acknowledge that practitioners may intentionally leave hard-coded secrets such as user names and SSH keys in scripts, which may not be enough to cause a security breach. Hence this practice is security smell, but not a vulnerability.

Invalid 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 [28]. 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 [11](database server), Memcached [12](cloud-based cache service) and Kibana [13](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. This smell is related to improper access control as stated in the weakness 'CWE-284: Improper Access Control' [6].

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. The smell is related to 'CWE-546: Suspicious Comment' [6]. Examples of such comments include putting keywords such as 'TODO', 'FIXME', and 'HACK' in comments, along with putting bug information in comments. Keywords such as 'TODO' and 'FIXME' in comments are used to specify an edge case or a problem [29]. However, these keywords make a comment 'suspicious' i.e., indicating missing functionality about the system.

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 [30]. For example, as shown in Figure 4, the authentication protocol is set to 'http' for the branch that satisfies the condition 'RedHat'. Such usage of HTTP can be problematic, as the 'admin-user' will be connecting over a HTTP-based protocol. An attacker can eavesdrop on the communication channel and may guess the password of user 'admin-user'. This security smell is related to 'CWE-319: Cleartext Transmission of Sensitive Information' [6]. The motivation for using HTTPS is to protect the privacy and integrity of the exchanged data. Information sent over HTTP may be encrypted, and in such case 'Use of HTTP without TLS' may not lead to a security attack.

Use of weak cryptography algorithms: This smell is the recurring pattern of using weak cryptography algorithms, such

[10] https://blogs.msdn.microsoft.com/visualstudio/2017/11/17/managing-secrets-securely-in-the-cloud/

[11] https://serversforhackers.com/c/mysql-network-security

[12] https://news.ycombinator.com/item?id=16493480

[13] https://www.elastic.co/guide/en/kibana/5.0/breaking-changes-5.0.html

Preprint

TABLE I: An Example of Using Code Snippets To Determine Rule for ‘Hard-coded secret’

Code Snippets	Output of Parser
$keystone_db_password = ‘keystone_pass’,	<VARIABLE, ‘$keystone_db_password’, ‘keystone_pass’ >
$glance_user_password = ‘glance_pass’,	<VARIABLE, ‘$glance_user_password’, ‘glance_pass’ >
$rabbit_password = ‘rabbit_pw’,	<VARIABLE, ‘$rabbit_password’, ‘rabbit_pw’ >
user =>‘jenkins’	<ATTRIBUTE, ‘user’, ‘jenkins’ >
$ssl_key_file = ‘/etc/ssl/private/ssl-cert-gerrit-review.key’	<VARIABLE, ‘$ssl_key_file’, ‘/etc/ssl/private/ssl-cert-gerrit-review.key’ >

as MD4 and SHA-1 for encryption purposes. MD5 suffers from security problems, as demonstrated by the Flame malware in 2012 [31]. MD5 is susceptible to collision attacks [32] and modular differential attacks [33]. In Figure 4, we observe a password is being set using the ‘ht_md5’ method provided by the ‘htpasswd’ Puppet module [14]. Similar to MD5, SHA1 is also susceptible to collision attacks [15]. This smell is related to ‘CWE-327: Use of a Broken or Risky Cryptographic Algorithm’ and ‘CWE-326: Inadequate Encryption Strength’ [6]. When weak algorithms such as MD5 are used for for hashing that may not lead to a breach, but using MD5 for password setup may.

```
1 #This is an example Puppet script
2 $token = `XXXYYYZZZ'
3 $os_name = `Windows'
4 auth_protocol => `http'
5 $vcenter_password = `password'
```

a

Line#	Output of Parser
1	<COMMENT, ‘This is an example Puppet script’>
2	<VARIABLE, ‘token’, ‘XXXYYYZZZ’>
3	<VARIABLE, ‘os_name’, ‘Windows’>
4	<ATTRIBUTE, ‘auth_protocol’, ‘http’>
5	<VARIABLE, ‘vcenter_password’, ‘password’>

b

Fig. 5: Output of the ‘Parser’ component in SLIC. Figure 5a presents an example IaC script fed to Parser. Figure 5b presents the output of Parser for the example IaC script.

IV. Security Linter for Infrastructure as Code Scripts (SLIC)

We first describe how we construct SLIC, then we describe how we evaluate SLIC with respect to smell detection accuracy.

A. Description of SLIC

SLIC is a static analysis tool for detecting security smells in IaC scripts. SLIC has two components:

Parser: The Parser parses an IaC script and returns a set of tokens. Tokens are non-whitespace character sequences extracted from IaC scripts, such as keywords and variables. Except for comments, each token is marked with its name, token type, and any associated configuration value. Only token type and configuration value are marked for comments. For example, Figure 5a provides a sample script that is fed into SLIC. The output of Parser is is expressed as a vector, as shown in Figure 5b. For example, the comment in line#1, is expressed as the vector ‘<COMMENT, ‘This is an example Puppet script’>’. Parser provides a vector representation of all code snippets in a script.

[14]https://forge.puppet.com/leinaddm/htpasswd

[15]https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html

Rule Engine: We take motivation from prior work [35] [36] and use a rule-based approach to detect security smells. We use rules because (i) unlike keyword-based searching, rules are less susceptible to false positives [35] [36]; and (ii) rules can be applicable for IaC tools irrespective of their syntax. The Rule Engine consists of a set of rules that correspond to the set of security smells identified in Section III-A. The Rule Engine uses the set of tokens extracted by Parser and checks if any rules are satisfied.

We can abstract patterns from the smell-related code snippets, and constitute a rule from the generated patterns. We use Table I to demonstrate our approach. The ‘Code Snippet’ column presents a list of code snippets related to ‘Hard-coded secret’. The ‘Parser Output’ column represents vectors for each code snippet. We observe that the vector of format ‘<VARIABLE, NAME, CONFIGURATION VALUE >’ and ‘<ATTRIBUTE, NAME, CONFIGURATION VALUE >’, respectively, occurs four times and once for our example set of code snippets. We use the vectors from the output of ‘Parser’ to determine that variable and attribute are related to ‘Hard-coded secret’. The vectors can be abstracted to construct the following rule: ‘$(isAttribute(x) \vee isVariable(x)) \wedge (isUser(x.name) \vee isPassword(x.name) \vee isPvtKey(x.name)) \wedge (length(x.value) > 0)$’. This rule states that ‘for an IaC script, if token x is a variable or an attribute, and a string is passed as configuration value for a variable or an attribute which is related to user name/password/private key, then the script contains the security smell ‘Hard-coded secret’. We apply the process of abstracting patterns from smell-related code snippets to determine the rules for the seven security smells.

A programmer can use SLIC to identify security smells for one or multiple Puppet scripts. The programmer specifies a directory where script(s) reside. Upon completion of analysis, SLIC generates a comma separated value (CSV) file where the count of security smell for each script is reported, along with the line number of the script where the smell occurred. We implement SLIC using API methods provided by puppet-lint [17].

Rules to Detect Security Smells: We present the rules needed for the ‘Rule Engine’ of SLIC in Table II. The string patterns need to support the rules in Table II are listed in Table III. The ‘Rule’ column lists rules for each smell that is executed by Rule Engine to detect smell occurrences. To detect whether or not a token type is a variable ($isVariable(x)$),

[17]http://puppet-lint.com/

Preprint

TABLE II: Rules to Detect Security Smells

Smell Name	Rule
Admin by default	$(isParameter(x)) \wedge (isAdmin(x.name) \wedge isUser(x.name))$
Empty password	$(isAttribute(x) \vee isVariable(x)) \wedge ((length(x.value) == 0 \wedge isPassword(x.name))$
Hard-coded secret	$(isAttribute(x) \vee isVariable(x)) \wedge (isUser(x.name) \vee isPassword(x.name) \vee isPvtKey(x.name)) \wedge (length(x.value)>0)$
Invalid IP address binding	$((isVariable(x) \vee isAttribute(x)) \wedge (isInvalidBind(x.value))$
Suspicious comment	$(isComment(x)) \wedge (hasWrongWord(x) \vee hasBugInfo(x))$
Use of HTTP without TLS	$(isAttribute(x) \vee isVariable(x)) \wedge (isHTTP(x.value))$
Use of weak crypto. algo.	$(isFunction(x) \wedge usesWeakAlgo(x.name))$

TABLE III: String Patterns Used for Functions in Rules

Function	String Pattern
$hasBugInfo()$ [34]	'bug[#nt]$*$[0-9]+','show_bug\.cgi?id=[0-9]+'
$hasWrongWord()$ [16]	'bug', 'hack', 'fixme', 'later', 'later2', 'todo'
$isAdmin()$	'admin'
$isHTTP()$	'http:'
$isInvalidBind()$	'0.0.0.0'
$isPassword()$	'pwd', 'pass', 'password'
$isPvtKey()$	'[pvt|priv]+*[cert|key|rsa|secret|ssl]+'
$isUser()$	'user'
$usesWeakAlgo()$	'md5', 'sha1'

an attribute ($isAttribute(x)$), a function ($isFunction(x)$), or a comment ($isComment(x)$), we use the token vectors generated by Parser. Each rule includes functions whose execution is dependent on matching of string patterns. We apply a string pattern-based matching strategy similar to prior work [37] [38], where we check if the value satisfies the necessary condition. Table III lists the functions and corresponding string patterns. For example, function 'hasBugInfo()' will return true if the string pattern 'show_bug\.cgi?id=[0-9]+' or 'bug[#nt]$*$[0-9]+' is satisfied.

B. Evaluation of SLIC

We evaluated the detection accuracy of SLIC by constructing an oracle dataset.

Oracle Dataset: We construct the oracle dataset by applying closed coding [25], where a rater identifies a pre-determined pattern. In the oracle dataset, 140 scripts are manually checked for security smells by at least two raters. The raters apply their knowledge related to IaC scripts and security, and determine if a certain smell appears for a script. To avoid bias, we did not include any raters as part of deriving smells or constructing SLIC.

We made the pattern identification task available to the students using a website. In each task, a rater determines which of the security smells identified in Section III-A occur in a script. We used graduate students as raters to construct the oracle dataset. We recruited these students from a graduate-level course conducted in the university. We obtained institutional review board (IRB) approval for the student participation. Of the 58 students in the class, 28 students agreed to participate. We assigned 140 scripts to the 28 students to ensure each script is reviewed by at least two students, where each student does not have to rate more than 10 scripts. We used balanced block design to assign 140 scripts from our collection of 1,726 scripts. We observe agreements on the rating for 79 of 140 scripts (56.4%), with a Cohen's Kappa of 0.3. According to Landis and Koch's interpretation [39], the reported agreement is 'fair'. In the case of disagreements between raters for 61 scripts, the first author resolved the disagreements.

Upon completion of the oracle dataset, we evaluate the accuracy of SLIC using precision and recall for the oracle dataset. Precision refers to the fraction of correctly identified smells among the total identified security smells, as determined by SLIC. Recall refers to the fraction of correctly identified smells that have been retrieved by SLIC over the total amount of security smells.

Accuracy of SLIC for Oracle Dataset: We report the detection accuracy of SLIC with respect to precision and recall in Table IV. As shown in the 'No smell' row, we identify 113 scripts with no security smells. The rest of the 27 scripts contained at least one occurrence of the seven smells. The count of occurrences for each security smell along with SLIC's precision and recall for the oracle dataset are provided in Table IV. For example, in the oracle dataset, we identify one occurrence of 'Admin by default' smell. The precision and recall of SLIC for one occurrence of admin by default is respectively, 1.0 and 1.0. SLIC generates zero false positives and one false negative for 'Hard-Coded secret'. For the oracle dataset average precision and recall of SLIC is 0.99.

TABLE IV: SLIC's Accuracy for the Oracle Dataset

Smell Name	Occurr.	Precision	Recall
Admin by default	1	1.00	1.00
Empty password	2	1.00	1.00
Hard-coded secret	24	1.00	0.96
Invalid IP address binding	4	1.00	1.00
Suspicious comment	17	1.00	1.00
Use of HTTP without TLS	9	1.00	1.00
Use of weak crypto. algo.	1	1.00	1.00
No smell	113	0.99	1.00
Average		0.99	0.99

Dataset and Tool Availability: The source code of SLIC and all constructed datasets are available online [40].

V. Empirical Study

A. Research Questions

We investigate the following research questions:

- RQ2: How frequently do security smells occur in infrastructure as code scripts?
- RQ3: What is the lifetime of the identified security smell occurrences for infrastructure as code scripts?

Preprint

TABLE V: OSS Repositories Satisfying Criteria (Sect. V-B)

	GH	MOZ	OST	WIK
Initial Repo Count	3,405,303	1,594	1,253	1,638
Criteria-1 (11% IaC Scripts)	6,088	2	67	11
Criteria-2 (Not a Clone)	4,040	2	61	11
Criteria-3 (Commits/Month $\geq$ 2)	2,711	2	61	11
Criteria-4 (Contributors $\geq$ 10)	219	2	61	11
Final Repo Count	219	2	61	11

- RQ4: Do practitioners agree with security smell occurrences?

B. Datasets

We conduct our empirical study with four datasets of Puppet scripts. Three datasets are constructed using repositories collected from three organizations: Mozilla, Openstack, and Wikimedia. The fourth dataset is constructed from repositories hosted on GitHub. To assess the prevalence of the identified smells and increase generalizability of our findings, we include repositories from Github, as companies tend to host their popular OSS projects on GitHub [41] [42].

As advocated by prior research [43], OSS repositories needs to be curated. We apply the following criteria to curate the collected repositories:

- **Criteria-1**: At least 11% of the files belonging to the repository must be IaC scripts. Jiang and Adams [8] reported for OSS repositories, which are used in production, IaC scripts co-exist with other types of files, such as Makefiles. They observed a median of 11% of the files to be IaC scripts. By using a cutoff of 11% we assume to collect repositories that contain sufficient amount of IaC scripts for analysis.
- **Criteria-2**: The repository is not a clone.
- **Criteria-3**: The repository must have at least two commits per month. Munaiah et al. [43] used the threshold of at least two commits per month to determine which repositories have enough software development activity. We use this threshold to filter repositories with short activity.
- **Criteria-4**: The repository has at least 10 contributors. Our assumption is that the criteria of at least 10 contributors may help us to filter out irrelevant repositories.

As shown in Table V, we answer RQ2 using 15,232 scripts collected from 219, 2, 61, and 11 repositories, respectively, from GitHub, Mozilla, Openstack, and Wikimedia. We clone the master branches of the 293 repositories. Summary attributes of the collected repositories are available in Table VI.

TABLE VI: Summary Attributes of the Datasets

Attribute	GH	MOZ	OST	WIK
Repository Type	Git	Mercurial	Git	Git
Repository Count	219	2	61	11
Total File Count	72,817	9,244	12,681	9,913
Total Puppet Scripts	8,010	1,613	2,764	2,845
Tot. LOC (Puppet Scripts)	424,184	66,367	214,541	135,137

C. Analysis

Sanity Check for SLIC's Accuracy: With respect to accuracy, SLIC may have high accuracy on the oracle dataset, but not on the complete dataset. To mitigate this limitation and assess SLIC's accuracy performance on the complete dataset, we perform a sanity check for a randomly-selected set of 250 scripts collected from four datasets. We manually inspect each of the 250 scripts for security smells. Next, we run SLIC on the collected scripts. Finally, we report the precision and recall of SLIC for the selected 250 scripts.

The first author performed manual inspection. From manual inspection we observe 40 occurrences of smells: 29 occurrences of hard-coded secrets; 8 occurrences of suspicious comments; and 3 occurrences of invalid IP address binding. Precision of SLIC for hard-coded secrets, suspicious comments, and invalid IP address binding is respectively, 0.78, 0.73, and 1.00. The recall of SLIC for hard-coded secrets, suspicious comments, and invalid IP address binding is respectively 1.00, 0.95, and 1.00. SLIC generated eight and three false positives respectively for 'Hard-coded secret', and 'Suspicious comment'. SLIC generated one false negative for 'Hard-coded secret'. The recall is $\geq$0.95, which indicates SLIC's ability to detect most existing smells, but may overestimate the frequency of smell occurrences.

1) RQ2: How frequently do security smells occur in infrastructure as code scripts?: RQ2 focuses on characterizing how frequently security smells are present. First, we apply SLIC to determine the security smell occurrences for each script. Second, we calculate two metrics described below:

- ***Smell Density***: Similar to prior research that have used defect density [44] [45] and vulnerability density [46], 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.

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

- ***Proportion of Scripts (Script%)***: Similar to prior work in defect analysis [20] [47], 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.

2) RQ3: What is the lifetime of the identified security smell occurrences for infrastructure as code scripts?: In RQ3, we focus on identifying the lifetime i.e., amount of time a security smell persists for the same script. A security smell that persists for a long time can facilitate attackers. We answer RQ3 by executing the following steps:

Step-1: For each smell occurrence s existent in script x for month m_i, we determine s to persist for month m_{i+1} if,

- s occurs for script x; and

Preprint

TABLE VII: Example of a Persistent Security Smell

Month	Code Snippet	Output of Parser	Persist?
2012-03	'bind_address' = '0.0.0.0'	<ATTRIBUTE, 'bind_address', '0.0.0.0' >	N/A
	$bind_host='0.0.0.0'	<VARIABLE, '$bind_host', '0.0.0.0' >	N/A
2012-04	'bind_address' = '0.0.0.0'	<ATTRIBUTE, 'bind_address', '0.0.0.0' >	**YES**
	$admin_bind_host = '0.0.0.0'	<VARIABLE, '$admin_bind_host', '0.0.0.0' >	**NO**

- s occurs with the same configuration value; and
- s occurs for the same type of token in the script such as, attribute, comment, function, or variable.

We further demonstrate our approach using an example, as shown in Table VII. In Table VII, we provide code snippets that relate to two instances of 'Invalid IP address binding'. In this example, as shown in the first row, we notice two occurrences of 'Invalid IP address binding', for the same script. Both of the smells occurred in March 2012. In the second row of Table VII, we observe for April 2012 two code snippets for which 'Invalid IP address binding' occurred. The first code snippet, as indicated by 'YES', represents a persistent smell because the smell occurred with the same attribute 'bind_address'.

Step-2, we repeat Step-1, for all smell occurrences for months, m_i, where $i = 2, 3, ..., N$, representing all months for the script the security smell occurred.

Step-3, we determine which smells have consecutive occurrences throughout the lifetime of the script. For each smell with consecutive occurrences, we calculate the difference between the time the smell first and last occurred. For example, for script x, if smell s occurs for months March 2012 and April 2012, then the lifetime of smell s will be one month. To avoid systematic overestimating of lifetime, we do not mark multiple attributes or variables with the same configuration values as persisting smells.

3) RQ4: Do practitioners agree with security smell occurrences?: We gather feedback using bug reports on how practitioners perceive the identified security smells. From the feedback we can assess if the identified security smells actually have an impact on how practitioners develop IaC scripts. We apply the following procedure:

First, we randomly select 1,000 occurrences of security smells from the four datasets. **Second**, we post a bug report for each occurrence, describing the following items: smell name, brief description, related CWE, and the script and line number where the smell occurred. We explicitly ask if contributors of the repository agrees to fix the smell instances.

VI. Empirical Findings

We answer the three research questions as following:

A. RQ2: How frequently do security smells occur in infrastructure as code scripts?

We observe our identified security smells to exist across all datasets. For GitHub, Mozilla, Openstack, and Wikimedia respectively, 29.3%, 17.9%, 32.9%, and 26.7% of all scripts include at least one occurrence of our identified smells. Hard-coded secret is the most prevalent security smell with respect to occurrences and smell density. Altogether, we identify 16,952 occurrences of hard-coded secrets, of which 68.3%, 23.9%, and 7.8% are respectively, hard-coded keys, user names, and passwords. A complete breakdown of findings related to RQ2 is presented in Table VIII for our four datasets GitHub ('GH') Mozilla ('MOZ'), Openstack ('OST'), and Wikimedia ('WIK').

Occurrences: The occurrences of the seven security smells are presented in the 'Occurrences' column of Table VIII. The 'Combined' row presents the total smell occurrences. For Github, Mozilla, Openstack, and Wikimedia we respectively observe 13221, 1141, 4507, and 2332 occurrences of security smells.

Smell Density: In the 'Smell Density (per KLOC)' column of Table VIII we report the smell density. The 'Combined' row presents the smell density for each dataset when all seven security smell occurrences are considered. For all four datasets, we observe the dominant security smell is 'Hard-coded secret'. The least dominant smell is 'Admin by default'.

Security smells occur more frequently for scripts hosted in GitHub. For hard-coded secrets, smell density is 1.5 $\sim$ 2.1 times higher for scripts hosted in GitHub than the other three datasets.

Proportion of Scripts (Script%): In the 'Proportion of Scripts (Script%)' column of Table VIII, we report the proportion of scripts (Script %) values for each of the four datasets. The 'Combined' row represents the proportion of scripts in which at least one of the seven smells appear. As shown in the 'Combined' row, percentage of scripts that have at least one of seven smells is respectively, 29.3%, 17.9%, 32.9%, and 26.7% for GitHub, Mozilla, Openstack, and Wikimedia.

B. RQ3: What is the lifetime of the identified security smell occurrences for infrastructure as code scripts?

The persistence of some hard-coded secrets is noticeable across all datasets. A security smell can persist in a script for as long as 98 months. We report our findings for RQ3 in Table IX. The median and maximum lifetime of each security smell is identified for the four datasets. The row 'At least One Smell' refers to the median and maximum lifetime of at least one type of security smell.

For GitHub, Mozilla, Openstack, and Wikimedia the maximum lifetime of a hard-coded secret is respectively, 92, 77, 89, and 98 months. Hard-coded secrets can reside in IaC scripts for a long period of time, which can give attackers opportunity to compromise the system. Awareness and perception can be two possible explanations for lengthy lifetime of security smells. If practitioners are not aware of the consequences of the smells then they may not fix them. Also, if the practitioners do not

Preprint

TABLE VIII: Smell Occurrences, Smell Density, and Proportion of Scripts for the Four Datasets

	Occurrences				Smell Density (per KLOC)				Proportion of Scripts (Script%)			
Smell Name	**GH**	**MOZ**	**OST**	**WIK**	**GH**	**MOZ**	**OST**	**WIK**	**GH**	**MOZ**	**OST**	**WIK**
Admin by default	52	4	35	6	0.1	0.06	0.1	0.04	0.6	0.2	1.1	0.2
Empty password	136	18	21	36	0.3	0.2	0.1	0.2	1.4	0.4	0.5	0.3
Hard-coded secret	10,892	792	3,552	1,716	25.6	11.9	16.5	12.7	21.9	9.9	24.8	17.0
Invalid IP address binding	188	20	114	41	0.4	0.3	0.5	0.3	1.7	0.7	2.9	1.4
Suspicious comment	758	202	305	343	1.7	3.0	1.4	2.5	5.9	8.5	7.2	9.1
Use of HTTP without TLS	1,018	57	460	164	2.4	0.8	2.1	1.2	6.3	1.6	8.5	3.7
Use of weak crypto algo.	177	48	20	26	0.4	0.7	0.1	0.2	0.9	1.1	0.5	0.4
Combined	13,221	1,141	4,507	2,332	31.1	17.2	21.0	17.2	29.3	17.9	32.9	26.7

TABLE IX: Lifetime of Security Smells (Months)

Smell Name	**GH**	**MOZ**	**OST**	**WIK**
	(Med, Max)	(Med, Max)	(Med, Max)	(Med, Max)
Admin by default	(30.0, 73.0)	(41.0, 41.0)	(15.0, 89.0)	(20.0, 22.0)
Empty password	(21.0, 76.0)	(27.5, 54.0)	(13.5, 89.0)	(18.5, 56.0)
Hard-coded secret	(24.0, 92.0)	(34.0, 77.0)	(15.0, 89.0)	(20.0, 98.0)
Invalid IP address binding	(31.0, 73.0)	(14.0, 77.0)	(22.0, 89.0)	(20.0, 63.0)
Suspicious comment	(21.0, 92.0)	(22.0, 77.0)	(11.0, 89.0)	(20.0, 61.0)
Use of HTTP without TLS	(23.0, 92.0)	(9.0, 77.0)	(13.0, 89.0)	(20.0, 93.0)
Use of weak crypto. algo.	(26.0, 92.0)	(47.0, 77.0)	(23.0, 89.0)	(20.0, 59.0)
At Least One Smell	(24.0, 92.0)	(23.5, 77.0)	(14.0, 89.0)	(20.0, 98.0)

perceive these smells to be consequential then smells may reside in scripts for a long duration.

C. RQ4: Do practitioners agree with security smell occurrences?

From 93 practitioners we obtain 104 responses for the submitted 1000 bug reports. We observe an agreement of 64.4% for 104 smell occurrences. The percentage of smells to which practitioners agreed to be fixed is presented in Figure 6. In y-axis each smell name is followed by the occurrence count. For example, according to Figure 6, for 30 occurrences of 'Hard-coded secret'(HARD_CODE_SECRET), we observe 70.0% agreement. We observe 75.0% or more agreement for two smells: 'Use of HTTP without TLS' and 'Use of weak cryptography algorithms'.

In their response, practitioners provided reasoning on why these smells appeared in the first place. For one occurrence of 'HTTP without TLS', practitioners highlighted the lack of documentation and tool support saying: "*Good catch. This was probably caused by lack of documentation or absence of https endpoint at the time of writing. Should be fixed in next release.*". Upon acceptance of the smell occurrences, practitioners also suggested how these smells can be mitigated.

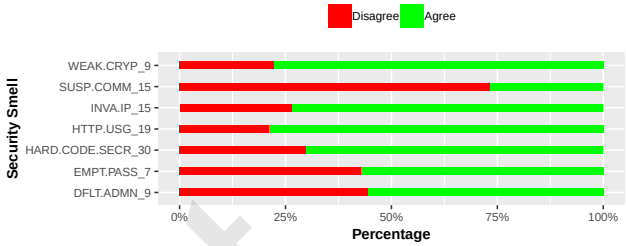

Fig. 6: Feedback for the 104 smell occurrences. Practitioners agreed with 64.4% of the selected smell occurrences.

For example, for an occurrence of 'Invalid IP address binding', one practitioner stated:"*I would accept a pull request to do a default of 127.0.0.1*".

Reasons for Practitioner Disagreements: We observe context to have importance to practitioners. For example, a hard-coded password may not have security implications if the hard-coded password resides in a repository used for training purposes. As one practitioner stated "*This is not publicly used module, but instead used in training only in a non-production environment. This module is designed in a manner to show basic functionality within Puppet Training courses.*". For one occurrence of 'HTTP Without TLS' one practitioner disagreed stating "*It's using http on localhost, what's the risk?*".

The above-mentioned statements from disagreeing practitioners also suggest lack of awareness: the users who use the training module of interest may consider use of hard-coded passwords as an acceptable practice, potentially propagating the practice of hard-coded secrets. Both local and remote sites that use HTTP can be insecure, as considered by practitioners from Google [18] [19]. Possible explanations for disagreements can also be attributed to perception of practitioners: smells in code have subjective interpretation [48], and programmers do not uniformly agree with all smell occurrences [49], [50]. Furthermore, researchers [51] have observed programmers' bias to perceive their code snippets as secure, even if the code snippets are insecure.

[18] https://security.googleblog.com/2018/02/a-secure-web-is-here-to-stay.html

[19] https://developers.google.com/web/fundamentals/security/encrypt-in-transit/why-https

Preprint

VII. Discussion

We suggest strategies on how the identified security smells can be mitigated along with other implications:

A. Mitigation Strategies

Admin by default: We advise practitioners to create user accounts that has the minimum possible security privilege and use that account as default. Recommendations from Saltzer and Schroeder [52] 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 [20] to store secrets
- scan IaC scripts to search for hard-coded secrets using tools such as CredScan and SLIC.

Invalid IP address binding: To mitigate this smell, we advise programmers to allocate their IP addresses systematically based on which services and resources needs 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.

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 [29]. Based on our findings we advocate for creating explicit guidelines on what pieces of information to store in comments, and strictly follow those guidelines through code review. For example, if a programmer submits code changes where a comment contains any of the patterns mentioned for suspicious comments in Table III, the submitted code changes will not be accepted.

Use of HTTP without TLS: We advocate companies to adopt the HTTP with TLS by leveraging resources provided by tool vendors, such as MySQL [21] and Apache [22]. 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 [53] to mitigate this smell.

B. Possible Implications

Guidelines: One possible strategy to mitigate security smells is to develop concrete guidelines on how to write IaC scripts in a secure manner. When constructing guidelines, the IaC community can take Acar et al. [54]'s findings into account, and include easy to understand, task-specific examples on how to write IaC scripts in a secure manner.

[20] https://www.vaultproject.io/

[21] https://dev.mysql.com/doc/refman/5.7/en/encrypted-connections.html

[22] https://httpd.apache.org/docs/2.4/ssl/ssl_howto.html

Prioritizing inspection efforts: From Section VI-A, answers to RQ2 indicates that not all IaC scripts include security smells. Researchers can build upon our findings to explore which characteristics correlate with IaC scripts with security smells. If certain characteristics correlate with scripts that have smells, then programmers can prioritize their inspection efforts for scripts that exhibit those characteristics. Researchers can also investigate if metric-based prediction techniques proposed in prior research [55] [56] can be used to identify IaC scripts that are more likely to include security smells, which can benefit from more scrutiny.

VIII. Threats to Validity

In this section, we discuss the limitations of our paper:

Conclusion Validity: The derived security smells and their association with CWEs are subject to the first author's judgment. We account for this limitation by applying verification of CWE mapping with two student raters who are not authors of the paper. Also, the oracle dataset constructed by the raters are susceptible to subjectivity, as the raters' judgment influences appearance of a certain security smell.

Internal Validity: We acknowledge that other security smells may exist. We mitigate this threat by manually analyzing 1,726 IaC scripts for security smells. In future, we aim to investigate if more security smells exist.

The detection accuracy of SLIC depends on the constructed rules that we have provided in Table II. We acknowledge that the constructed rules are heuristic-driven and susceptible to generating false positives and false negatives.

External Validity: Our findings are subject to external validity, as our findings may not be generalizable. We observe how security smells are subject to practitioner interpretation, and thus the relevance of security smells may vary from one practitioner to another. We construct our datasets using Puppet, which is a declarative language. Our findings may not generalize for IaC scripts that use an imperative form of language. Also, our scripts are collected from the OSS domain, and not from proprietary sources.

IX. Conclusion

IaC scripts help companies to automatically provision and configure their development environment, deployment environment, and servers. Security smells are recurring coding patterns in IaC scripts that are indicative of security weakness and can potentially lead to security breaches. By applying qualitative analysis on 1,726 scripts we identified seven security smells: admin by default; empty password; hard-coded secret; invalid IP address binding; suspicious comment; use of HTTP without TLS; and use of weak cryptography algorithms. We analyzed 15,232 IaC scripts to determine which security smells occur and used a tool called SLIC, to automatically identify security smells that occur in IaC scripts. We evaluated SLIC's accuracy by constructing an oracle dataset. We identified 21,201 occurrences of security smells that included 1,326 occurrences of hard-coded passwords. Based on smell density, we observed the most dominant and least dominant security

Preprint

smell to be respectively, 'Hard-coded secret' and 'Admin by default'. We randomly selected 1,000 occurrences of security smells and observe 64.4% agreement from practitioners for 104 responses. We observed security smells to persist, for example, hard-coded secrets can reside in an IaC script for up-to 98 months. Based on our findings, we recommend concrete guidelines for practitioners to write IaC scripts in a secure manner. We hope our paper will facilitate further security-related research in the domain of IaC scripts.

References

[1] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[2] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 189–200. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901761

[3] A. Rahman, A. Partho, P. Morrison, and L. Williams, "What questions do programmers ask about configuration as code?" in *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering*, ser. RCoSE '18. New York, NY, USA: ACM, 2018, pp. 16–22. [Online]. Available: http://doi.acm.org/10.1145/3194760.3194769

[4] P. Labs, "Puppet Documentation," https://docs.puppet.com/, 2018, [Online; accessed 08-Aug-2018].

[5] Puppet, "Nyse and ice: Compliance, devops and efficient growth with puppet enterprise," Puppet, Tech. Rep., April 2018. [Online]. Available: https://puppet.com/resources/case-study/nyse-and-ice

[6] MITRE, "CWE-Common Weakness Enumeration," https://cwe.mitre.org/index.html, 2018, [Online; accessed 08-Aug-2018].

[7] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[8] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code: An empirical study," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 45–55. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820527

[9] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: A configuration verification tool for puppet," *SIGPLAN Not.*, vol. 51, no. 6, pp. 416–430, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/2980983.2908083

[10] P. Labs, "Borsa istanbul: Improving efficiency and reducing costs to manage a growing infrastructure," Puppet, Tech. Rep., July 2018. [Online]. Available: https://puppet.com/resources/case-study/borsa-istanbul

[11] Puppet, "Ambit energy's competitive advantage? it's really a devops software company," Puppet, Tech. Rep., April 2018. [Online]. Available: https://puppet.com/resources/case-study/ambit-energy

[12] N. Meng, S. Nagy, D. D. Yao, W. Zhuang, and G. A. Argoty, "Secure coding practices in java: Challenges and vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 372–383. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180201

[13] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: An analysis of android ssl (in)security," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 50–61. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382205

[14] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046779

[15] M. Ghafari, P. Gadient, and O. Nierstrasz, "Security smells in android," in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sept 2017, pp. 121–130.

[16] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 73–84. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516693

[17] J. Schwarz, "Code Smell Detection in Infrastructure as Code," https://www.swc.rwth-aachen.de/thesis/code-smell-detection-infrastructure-code/, 2017, [Online; accessed 08-Aug-2018].

[18] E. van der Bent, J. Hage, J. Visser, and G. Gousios, "How good is your puppet? an empirically defined and validated quality model for puppet," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 164–174.

[19] O. Hanappi, W. Hummer, and S. Dustdar, "Asserting reliable convergence for configuration management scripts," *SIGPLAN Not.*, vol. 51, no. 10, pp. 328–343, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/3022671.2984000

[20] A. Rahman and L. Williams, "Characterizing defective configuration scripts used for continuous deployment," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, April 2018, pp. 34–45.

[21] A. Rahman, A. Partho, D. Meder, and L. Williams, "Which factors influence practitioners' usage of build automation tools?" in *Proceedings of the 3rd International Workshop on Rapid Continuous Software Engineering*, ser. RCoSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 20–26. [Online]. Available: https://doi.org/10.1109/RCoSE.2017..8

[22] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[23] N. I. of Standards and T. (NIST), "Csrc-glossary-vulnerability," https://csrc.nist.gov/Glossary/?term=2436, 2018, [Online; accessed 09-Aug-2018].

[24] J. T. McCune and Jeffrey, *Pro Puppet*, 1st ed. Apress, 2011. [Online]. Available: https://www.springer.com/gp/book/9781430230571

[25] J. Saldaña, *The coding manual for qualitative researchers*. Sage, 2015.

[26] N. I. of Standards and T. (NIST), "Security and privacy controls for federal information systems and organizations," https://www.nist.gov/publications/security-and-privacy-controls-federal-information-systems-and-organizations-including-0, 2014, [Online; accessed 18-Aug-2018].

[27] T. Ylonen and C. Lonvick, "The secure shell (ssh) protocol architecture," 2006.

[28] P. Mutaf, "Defending against a denial-of-service attack on tcp." in *Recent Advances in Intrusion Detection*, 1999.

[29] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer, "Todo or to bug: Exploring how task annotations play a role in the work practices of software developers," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 251–260. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368123

[30] E. Rescorla, "Http over tls," 2000.

[31] L. of Cryptography and S. S. (CrySyS), "skywiper (a.k.a. flame a.k.a. flamer): A complex malware for targeted attacks," Laboratory of Cryptography and System Security, Budapest, Hungary, Tech. Rep., May 2012. [Online]. Available: http://www.crysys.hu/skywiper/skywiper.pdf

[32] B. den Boer and A. Bosselaers, "Collisions for the compression function of md5," in *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology*, ser. EUROCRYPT '93. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1994, pp. 293–304. [Online]. Available: http://dl.acm.org/citation.cfm?id=188307.188356

[33] X. Wang and H. Yu, "How to break md5 and other hash functions," in *Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 19–35. [Online]. Available: http://dx.doi.org/10.1007/11426639_2

[34] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147

[35] I. K. Ratol and M. P. Robillard, "Detecting fragile comments," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 112–122. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155581

Preprint

[36] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: Bugs or bad comments?*/," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 145–158. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294276

[37] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: An empirical study," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 257–268. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635880

[38] S. Bugiel, S. Nurnberger, T. Poppelmann, A.-R. Sadeghi, and T. Schneider, "Amazon IA: When elasticity snaps back," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 389–400. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046753

[39] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977. [Online]. Available: http://www.jstor.org/stable/2529310

[40] A. Authors, "Dataset for Paper: The Seven Sins-Security Smells in Infrastructure as Code Scripts," 8 2018. [Online]. Available: https://figshare.com/s/9f4439a60ffcfc035453

[41] R. Krishna, A. Agrawal, A. Rahman, A. Sobran, and T. Menzies, "What is the connection between issues, bugs, and enhancements?: Lessons learned from 800+ software projects," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '18. New York, NY, USA: ACM, 2018, pp. 306–315. [Online]. Available: http://doi.acm.org/10.1145/3183519.3183548

[42] A. Agrawal, A. Rahman, R. Krishna, A. Sobran, and T. Menzies, "We don't need another hero?: The impact of "heroes" on software development," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '18. New York, NY, USA: ACM, 2018, pp. 245–253. [Online]. Available: http://doi.acm.org/10.1145/3183519.3183549

[43] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, pp. 1–35, 2017. [Online]. Available: http://dx.doi.org/10.1007/s10664-017-9512-6

[44] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 580–586. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062558

[45] J. C. Kelly, J. S. Sherif, and J. Hops, "An analysis of defect densities found during software inspections," *Journal of Systems and Software*, vol. 17, no. 2, pp. 111 – 117, 1992. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0164121292900893

[46] O. H. Alhazmi and Y. K. Malaiya, "Quantitative vulnerability assessment of systems software," in *Annual Reliability and Maintainability Symposium, 2005. Proceedings.*, Jan 2005, pp. 615–620.

[47] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, Jan 2007.

[48] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 33:1–33:39, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629648

[49] M. V. Mantyla and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Softw. Engg.*, vol. 11, no. 3, pp. 395–431, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1007/s10664-006-9002-8

[50] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 101–110. [Online]. Available: http://dx.doi.org/10.1109/ICSME.2014.32

[51] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic apis," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 154–171.

[52] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, Sept 1975.

[53] E. Barker, "Guideline for using cryptographic standards in the federal government: Cryptographic mechanisms," National Institute of Standards and Technology, Gaithersburg, Maryland, Tech. Rep., August 2016. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-175b.pdf

[54] Y. Acar, C. Stransky, D. Wermke, C. Weir, M. L. Mazurek, and S. Fahl, "Developers need support, too: A survey of security advice for software developers," in *2017 IEEE Cybersecurity Development (SecDev)*, Sept 2017, pp. 22–26.

[55] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *2010 Third International Conference on Software Testing, Verification and Validation*, April 2010, pp. 421–428.

[56] A. Rahman, P. Pradhan, A. Partho, and L. Williams, "Predicting android application security and privacy risk with static code metrics," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, May 2017, pp. 149–153.

Preprint