Share, But Be Aware: Security Smells in Python Gists

Md Rayhanur Rahman
North Carolina State University
mrahman@ncsu.edu

Akond Rahman
Tennessee Tech. University
akond.rahman.buet@gmail.com

Laurie Williams
North Carolina State University
lawilli3@ncsu.edu

***Abstract*—Github Gist is a service provided by Github which is used by developers to share code snippets. While sharing, developers may inadvertently introduce security smells in code snippets as well, such as hard-coded passwords. Security smells are recurrent coding patterns that are indicative of security weaknesses, which could potentially lead to security breaches. *The goal of this paper is to help software practitioners avoid insecure coding practices through an empirical study of security smells in publicly-available GitHub Gists*. Through static analysis, we found 13 types of security smells with 4,403 occurrences in 5,822 publicly-available Python Gists. 1,817 of those Gists, which is around 31%, have at least one security smell including 689 instances of hard-coded secrets. We also found no significance relation between the presence of these security smells and the reputation of the Gist author. Based on our findings, we advocate for increased awareness and rigorous code review efforts related to software security for Github Gists so that propagation of insecure coding practices are mitigated.**

***Index Terms*—GitHub, Gist, Python, Security, Security Smell, Static Analysis, Software Security**

I. Introduction

GitHub Gist[1] is an online platform that allows GitHub users to store and share *Gists* with developers. Gists are code snippets and related textual information prepared for explaining and demonstrating a programming concept or task [1]. Similar to GitHub repositories, Gists are also version controlled and clone-able. GitHub Gists have gained in popularity including more than 17.7 million Gists written in 260 programming languages. As these Gists are used and shared by the developers, coding practices in these Gists are also likely to be spread across software projects.

Gists can enable the propagation of insecure coding practices. Let us consider Figure 1 as an example showing a Python Gist[2] where, we observe a hard-coded password in line 6. Hard-coded password in software artifacts is considered as a software security weakness ('CWE-798: Use of Hard-coded Credentials') by Common Weakness Enumeration (CWE) [2]. According to CWE [2], "*If hard-coded passwords are used, it is almost certain that malicious users will gain access to the account in question*". Such recurrence of insecure coding practices are called security smell which are indicative of security weakness [3]. The existence and persistence of these smells in GitHub Gists leaves the possibility of another practitioner using these GitHub Gists with security smells, potentially propagating the use of insecure coding practices. Hence existence of the security smells in Gists along with its relation with author should be investigated to raise awareness among the software practitioners while using and sharing the Gists.

```
3  import wikitools
4
5  username = 'Legoktm'
6  password = 'hunter2'
7  messages = ['Signupstart', 'Signupend']
8  reason = '...'
```

Fig. 1. An example of hard-coded password in a publicly-available Python Gist.

The goal of this paper is to help software practitioners avoid insecure coding practices through an empirical study of security smells in publicly-available GitHub Gists.

We answer the following research questions (RQs):

- **RQ1**: What type of security smells appear in GitHub Gists?
- **RQ2**: How frequently do security smells appear in GitHub Gists?
- **RQ3**: How does a Gist user's reputation relates to the presence of security smells in GitHub Gists?

We answer our research questions by collecting and analyzing GitHub Gists written in Python. We select Python because Python is ranked as the top-most programming language by IEEE Spectrum [4] in 2018. We first apply qualitative analysis [5] to find out what security smells exist in Python Gists. Next, we apply static analysis using Bandit[3] and our developed static analysis tool to identify the existence and frequency of the security smells for $5,822$ publicly-available Python Gists. We also apply hypothesis testing to investigate the relationship between Gist author reputation and presence of security smells in Python Gists.

We list our contribution as following:

- A list of 13 security smell types found in Python Gists;
- An analysis on how frequently security smells appear in Python Gists; and
- An analysis which quantifies the relationship between Gist author reputation and presence of security smells in Python Gists

[1]https://gist.github.com/

[2]https://gist.github.com/legoktm/7781996

[3]https://github.com/PyCQA/bandit

II. Related Work

Prior research has demonstrated the existence of insecure coding practices in code snippets typically found in the Internet. In [6], Rahman et al. analyzed insecure coding practices in Stack Overflow answers of Python related questions and found out that 7.1% of the answers have at least one insecure coding practice. In [7], Unruh et al. found that vulnerabilities exist in web application development tutorials propagating across the projects through the lack of carefulness and attention from developers. Moreover, Chen et al. [8] performed the comparison of Stack Overflow answers containing both secure and insecure coding examples and found that developer community often does not differentiate between secure and insecure practices. Meng et al. [9] concluded with similar findings from their analysis of secure coding practice of Java-related Stack Overflow answers. These aforementioned findings raise the fact that insecure coding practices in code snippets available online can be a major source of introducing and spreading software security weakness throughout open source and commercial projects. Inspired by these facts, we investigate whether security smells exists in Github Gists and to what extent.

III. Security Smells

In this section, we describe how we derive security smells from Python Gists followed by the definitions and examples of the 13 derived security smells.

Fig. 2. Examples of all security smells

RQ1: What type of security smells appear in Python Gists?

A. Derivation of Security Smells through Qualitative Analysis:

We first obtain a list of insecure coding practices from these six sources: (i) CWE [2]; (ii) Openstack Security Guidelines for Python [10]; (iii) Bandit [11]; (iv) Sonar Source Python Security Hotspots [12]; (v) prior work on security smells [3]; and prior work on insecure coding practices in Stack Overflow [6]. We select these sources as these sources already have an existing list of insecure coding practices for multiple programming languages with examples of coding patterns. Next, the first two authors of this paper individually map each of those identified smells to a potential security weakness indexed in CWE [2]. We mapped CWE as it is a community-driven database for software security weaknesses and vulnerabilities [2]. The goal of creating this database is to understand security weaknesses in software, create automated tools so that security weaknesses in software can be automatically identified and repaired, and create a common baseline standard for security weakness identification, mitigation, and prevention efforts [2]. The mapping agreement score was associated with Cohen's kappa score of 1 which indicates a perfect agreement [8]. We performed such analysis to get a summarized overview of recurring coding patterns and syntactic context for automatic identification of those security smells. We have chosen CWE mapping of security smells as CWE provides a list of common software security weaknesses which are developed and maintained by software security experts [2].

B. Answer to RQ1

In this section, we describe the 13 identified security smells obtained from the aforementioned analysis. The smells are listed in alphabetical order. Examples of the security smells are provided in Figure 2.

Bad File Permission: This smell is the recurring pattern of using *chmod* POSIX API to grant low levels of restrictions with group executable and world writeable access. This smell can lead to vulnerabilities such as information disclosure and code execution. This smell is related with CWE-732: Incorrect Permission Assignment for Critical Resource [2].

Command Injection: This smell refers to the occurrence of calling a process using *popen*, *subprocess*, *os.system* and taking arguments from variables or user inputs. Command injection facilitates attackers invoking external executables and injecting shell commands if sanitation of inputs is not performed carefully. This smell is related to CWE-77: Command Injection [2].

Constructing SQL Statement Upon User Input: This smell is the recurring pattern of using SQL statements based on user inputs in Python Gists. This will leave room for the malicious users to insert malicious SQL queries. This smell is related to CWE-89: SQL Injection [2].

Deployment with Debug Flag set to TRUE: This smell is the occurrence of delivering production code with *DEBUG* feature enabled. As this feature is used by the developers to find bugs efficiently in the code through system and application logs, error reports, faults and traces, it is possible for malicious users to gain security critical information. This smell is related with CWE-215: Information Exposure Through Debug Information [2].

Empty Password: This smell refers to the usage of password strings with zero lengths and thus indicates a default or weak password. Although this smell does not lead to security breaches, it makes guessing the password an easier task. The smell is similar to the weakness CWE-258: Empty Password in Configuration File [2].

Exec Statement: This smell occurs when the Python environment dynamically executes arbitrary codes based on user inputs using *exec* statement. Codes executing OS commands dynamically might help an attacker to execute unexpected or dangerous commands. This smell is related with CWE-95: Improper Neutralization of Directives in Dynamically Evaluated Code and CWE-77: Command Injection [2].

Hard-coded IP Address Bindings: This smell is the recurring pattern of assigning the IP address $0.0.0.0$ or other hard-coded values for remote server address. Such practice might allow connection from every possible sources and thus can be a source of vulnerability. In addition, hard-coded binding of IP address is another bad practice from secure coding point of view because, malicious users can discover a security sensitive IP address from decompiling the code and thus launch attacks. This smell is related to CWE-200: Information Disclosure [2].

Hard-coded Secrets: This smell is the recurring pattern of exposing sensitive information, such as user name and passwords in Python Gists. We consider three types of hard-coded secrets: hard-coded passwords/tokens, hard-coded user names, and hard-coded private cryptography keys. This smell is related with CWE-798: Use of Hard-coded Credentials [2].

Hard-coded tmp Directories: This smell is the recurring use of hard-coded tmp directory to save data that can not be stored in memory or to pass to external programs that must read from a file. This smell leads to a number of security problems, such as malicious users guessing the tmp directory name and writing to that directory containing the temporary file and thus enables them to effectively hijack the location through a symbolic link. This smell is related to CWE-377: Insecure Temporary File [2].

Ignoring Except Block: This smell refers to the practice of catching an exception and ignoring it silently through either *pass* or *continue* statements. It represents potential security issues as errors raised by the application might be caused from service disruption/interference attacks. This smell is related to CWE-703: Improper Check or Handling of Exceptional Conditions [2].

No Integrity Check: This smell refers to the practice of downloading .iso, .tar, .tar.gz, .dmg, .deb, .bin, .rpm, and .zip files from Internet and not checking the integrity (checksum or *sha*) of those files. Thus, malicious users can easily attack the system by providing corrupted file and injecting malicious payloads. This smell is related to CWE-353: Missing Support for Integrity Check and CWE-434: Unrestricted Upload of File with Dangerous Type [2].

No Certificate Validation: This smell is the recurring pattern of requesting content without any certificate (TLS) verification. This smell facilitate attackers to use an invalid and/or expired certificates to claim to be a trusted host. This smell is related to CWE-599: Missing Validation of OpenSSL Certificate [2].

Use of HTTP without TLS: This smell occurs when a statement makes HTTP call without using TLS and thus leads to a less secure connection. Without TLS, any connection is vulnerable to man-in-the-middle attacks, eavesdropping and guessing the credentials. This security smell is related to CWE- 319: Cleartext Transmission of Sensitive Information [2].

The presence of these smells indicates that Python Gists are susceptible to having software security weakness which can propagate through the sharing of Gists.

IV. Empirical Study and Findings

A. Datasets

We conduct our empirical study by building upon Horton and Parnin's research on Github Gists [1]. Horton and Parnin constructed a curated dataset of $10,259$ publicly-available Python Gists, of which $5,822$ Gists are executable. We use the set of publicly-available $5,822$ Python Gists because (i) these Gists are executable, which could be indicative of usability amongst practitioners; and (ii) these Gists are curated based on popularity [1].

B. Static Analysis

As discussed in Section III-A, we performed qualitative analysis to derive security smells in the Python Gists and to get the syntactic contexts to automatically identify the smell occurrences in the Gists. For the purpose of identifying smell occurrences, we have used two static analyzers. The first of those is Bandit [4], a static analyzer for security smells in Python scripts [11]. However, as Bandit does not cover all of the smells mentioned in section III-B, we build another static analysis tool to identify 6 smells that can not be captured by Bandit. Similar to Bandit, our tool also uses the Python *ast* module [13] and captures smell occurrences while traversing the abstract syntax tree of the Python Gists. These following 6 smells are identified by the second analyzer: Use of Command Line Arguments which is a part of Command Injection, Empty Passwords, Debug Flag Set to True, Hard-coded Secret, No Integrity Check and Use of HTTP without TLS; rest of the smells are captured by the Bandit. In Table I, the rules are described that are used for identification of these 6 smells. We derive these rules from observing the coding patterns in the Gists. The identification of these smells includes detecting and analyzing variable/object-attribute/dictionary-key names, their corresponding string values, presence of strings containing HTTP URLs, use of python default method calls to request HTTP call, to take input from command line arguments and to apply checksum on downloaded files.

C. Oracle Dataset

The first author of this paper constructed an oracle dataset to evaluate the performance of our tool(s) for security smell

[4]https://github.com/PyCQA/bandit

TABLE I
RULES FOR THE CUSTOM SMELL DETECTOR TOOL

Smell	Rules
Debug Flag Set to True	$(isVariableName(x) \vee isObjectAttributeName(x) \vee isDictionaryKeyName(x)) \wedge (x = DEBUG \vee x = DEBUG\_PROPAGATE\_EXCEPTIONS) \wedge value(x) = True$
Empty Password	$(isVariableName(x) \vee isObjectAttributeName(x) \vee isDictionaryKeyName(x)) \wedge isCommonPasswordName(x) \wedge length(value(x)) = 0$
Hard-coded Secret	$(isVariableName(x) \vee isObjectAttributeName(x) \vee isDictionaryKeyName(x)) \wedge (isCommonPasswordName(x) \vee isCommonUserName(x) \vee isCommonIdName(x) \vee isCommonTokenName(x) \vee isCommonKeyName(x)) \wedge length(value(x)) > 0$
No Integrity Check	$isHttpDownload(x) = True \wedge useOfChecksumLibrary(x) = False$
Use of Command Line Arguments	$isFunctionCallName(x) = sys.argv \vee isFunctionCallName(x) = ArgumentParser \vee isFunctionCallName(x) = argparse$
Use of HTTP without TLS	$isHttpCall(x) = True \wedge isTLSused(x) = False$

detection. The first author checked 100 Gists manually for security smells and applied knowledge regarding Python syntax and associated security issues and then determined whether a smell in particular appears in the Gist. After completion of the Oracle dataset, we evaluated the performance of the smell detection through the measurement of *precision* and *recall*. *Precision* refers to the fraction of correctly identified smells among the total identified security smells and *recall* refers to the fraction of correctly identified smells over the total number of security smells. In Table II, the *precision* and recall scores are given for the oracle dataset. *Recall* of all the smells are greater than 0.75 and provides the confidence to identify all the security smells with lower false alarm.

TABLE II
ACCURACY FOR THE ORACLE DATASET

Smell Name	Occurrences	Precision	Recall
Bad File Permission	1	1.00	1.00
Debug Set to True	2	1.00	1.00
Empty Password	2	1.00	1.00
Exec Statement	4	1.00	1.00
Hard-coded IP Address Binding	3	1.00	1.00
Hard-coded Secret	31	0.94	0.91
Constructing SQL upon Input	3	1.00	1.00
Hard-coded tmp Directory	3	1.00	1.00
Ignoring Except Block	13	1.00	1.00
No Integrity Check	7	0.43	0.75
No Certificate Validation	1	1.00	1.00
Command Injection	86	1.00	1.00
Use of HTTP without TLS	71	1.00	1.00
Combined	**227**	**0.97**	**0.98**

D. Empirical Findings

RQ2: How frequently do security smells appear in Python Gists? This question focuses on how frequently do security smells appear in the Python Gists. First, we performed static analysis using two aforementioned tools on the Gists and then we applied these two following metrics on the results: (i) *Smell Density:* This measure is used to quantify the frequency of smells per 1000 lines of code [14] as shown in the equation below.

$$smell\ density\ (x) = \frac{1000 \times total\ occurrence\ of\ x}{total\ line\ counts\ of\ all\ Gists} \quad (1)$$

and (ii) *Proportion of Gists:* This measure is used to quantify how many scripts have at least one security smells. This metric refers to the percentage of scripts that contain at least one occurrence of a particular smell.

TABLE III
SMELL OCCURRENCES, SMELL DENSITY, AND PROPORTION OF GISTS

Smell Name	Occurrences	Density per KLOC	Proportion of Gists(%)
Bad File Permission	2	0.01	0.03
Debug Set to True	35	0.09	0.60
Empty Password	25	0.06	0.43
Exec Statement	11	0.03	0.17
Hard-coded IP Address Binding	47	0.12	0.74
Hard-coded Secret	689	1.77	5.84
Constructing SQL upon Input	43	0.11	0.50
Hard-coded tmp Directory	87	0.22	1.10
Ignoring Except Block	208	0.53	2.58
No Integrity Check	8	0.02	0.05
No Certificate Validation	11	0.03	0.09
Command Injection	2,372	6.10	17.71
Use of HTTP without TLS	865	2.22	8.64
Combined	**4,403**	**11.32**	**31.21**

These two metrics represent the occurrence of security smells differently. The first one is more granular and focuses on the content of a script as measured by how many smells occur for every 1000 LOC. The second one is less granular and focuses on the existence of at least one of the 13 security smells, similar to prior research [3]. In Table III, we observe four types of smells to occur more frequently. These four smells are: Command Injection, Use of HTTP without TLS, Hard-coded Secret, and Ignoring Except Block. On the other hand, smells such as No Certificate Validation, Bad File Permission, Exec Statement and No Integrity Check occurs rarely. Command Injection smell is associated with the highest density and proportion meanwhile, bad file permission is associated with the lowest density and proportion values. In total, there are $4,403$ occurrences of smells, almost 12 smells per 1000 lines of code as well as around 31% of the Gists having at least one smell. These findings indicate that developers should be aware of security smells while they use and share Github Gists otherwise these smell will propagate to other software projects. Although Gists are used as toy examples to demonstrate programming concepts [1], if these Gists are copied and adapted into production code without careful inspection, the production code would have security

weakness. It is also noteworthy that, certain smells such as hard-coded secret and command injection occur a lot more in Gists due to the inherent nature of the demonstration of the corresponding programming tasks and thus high occurrence of these smells are not that alarming unless someone blindly copying these Gists into their codes.

RQ3: How does a Gist user's reputation relates to the presence of security smells in GitHub Gists? Researchers found from the prior work [15], [16] that there is a quantifiable relationship between the author and the corresponding source code quality. Similarly, we hypothesize whether the presence of security smells in Gists is related to the reputation of the Gist author. We collect the information regarding how many days passed since the author joined Github and how many users follow the particular author in the Github using the Github Rest API v3[5]. Then we calculate author reputation score using equation below.

$$author\ reputation\ score = \frac{number\ of\ followers\ author\ has}{number\ of\ days\ author\ joined} \quad (2)$$

We perform *Mann-Whitney's U Test* to observe any significance in the relation between the author reputation and presence of security smells in Python Gists. To do that, we categorize our Gists into two segments: Gists with no smell; and Gists with at least one smell. After that, we obtain author reputation scores of all of the Gist authors from aforementioned two Gist categories using the Equation 2. Thus we obtain two list of reputation scores: (i) of authors whose written Gists have no smell and (ii) of authors whose written Gists have at least one smell. Then we determine a relationship to exist between author reputation and presence of a security smell if level of significance, $p < 0.05$. To measure the size of the relationship, we use *Cliff's Delta* measurement and find out that although there is a relationship between the author reputation and occurrence of security smells in the Gists, this significance is of negligible (0.041) effect size. Thus we observe author reputation have no significant relation with the presence of insecure coding practices in Gists. Other factors, such as security knowledge could be related to presence of security smells–an area, which could be of interest to researchers.

V. Limitations

We list the limitations of our paper as following. The mapping between CWE indexes and security smell is based upon author's judgment and subjective view. Selection of the oracle dataset is also based upon the selection bias of the authors. The number of Gists used in this work is relatively small due to the unavailability of unrestricted Python Gists collection API from Github. The Gists might have other security smells as well, for example, certain Python methods such as `yaml.load()` could have security weakness which we ignore in this research scope. Moreover, we also ignore context specific insecure coding practices, such as applying regular expression to user inputs. Finally, any static analysis tool is susceptible to invoke false positives and two static analysis tools used in this work is no exception.

[5]https://developer.github.com/v3/

VI. Conclusion

Using Gists, developers can share the implementation of programming concepts to help developers across the Internet. However, security smells residing in the Gists can spread across software projects through usage and share. We investigated the existence of security smells in Python Gists and found 13 types of smells in publicly-available $5,822$ Gists. We identified $4,403$ smell occurrences including 689 hard-coded secrets. From the smell density analysis, we found out that command injection is the most prevalent security smell and bad file permission is the least one. We also found out that author reputation is not significantly related to the presence of smell in Gists. From our findings we emphasize on increased awareness and rigorous code review efforts while using and sharing Github Gists so that propagation of insecure coding practices are mitigated.

References

[1] E. Horton and C. Parnin, "Gistable: Evaluating the executability of python code snippets on github," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018.

[2] "CWE-Common Weakness Enumeration," https://cwe.mitre.org/, [accessed:1/6/19].

[3] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *Proceedings of the 41st International Conference on Software Engineering*, 2019.

[4] "2018 Top Programming Languages," https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages, [accessed:1/6/19].

[5] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*, 2012.

[6] A. Rahman, E. Farhana, and N. Imtiaz, "Snakes in paradise?: Insecure python-related coding practices in stack overflow," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR '19, 2019.

[7] T. Unruh, B. Shastry, M. Skoruppa, F. Maggi, K. Rieck, J.-P. Seifert, and F. Yamaguchi, "Leveraging flawed tutorials for seeding large-scale web vulnerability discovery," in *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

[8] M. Chen, F. Fischer, N. Meng, X. Wang, and J. Grossklags, "How reliable is the crowdsourced knowledge of security implementation?" *arXiv preprint arXiv:1901.01327*, 2019.

[9] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. Arango-Argoty, "Secure coding practices in java: Challenges and vulnerabilities," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018.

[10] "Openstack Docs," https://security.openstack.org/, [accessed:1/6/19].

[11] "Bandit," https://github.com/PyCQA/bandit, [accessed:1/6/19].

[12] "Python Sonar Source," https://rules.sonarsource.com, [accessed:1/6/19].

[13] "Abstract Syntax Trees," https://docs.python.org/3/library/ast.html, [accessed:1/6/19].

[14] 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*, 2005.

[15] M. Greiler, K. Herzig, and J. Czerwonka, "Code ownership and software quality: A replication study," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015.

[16] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.