UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Security Smell: Una Systematic Literature Review

RELATORE

Prof. Fabio Palomba

Università degli Studi di Salerno

CANDIDATO

Alessandro Tortora

Matricola: 0512106428

Anno Accademico 2021-2022

Questa tesi è stata realizzata nel

Ai miei genitori, a mio fratello e ad Alessia, per aver sempre creduto in me.

Abstract

In un periodo in cui lo sviluppo software assume un ruolo sempre più centrale in ambito informatico, diventa indispensabile garantire la sicurezza dei sistemi che vengono sviluppati. Nella scrittura del codice di produzione, quindi, è di fondamentale importanza prestare attenzione a non generare errori di programmazione - o bad coding practice - che possano compromettere la sicurezza del sistema. Molte volte questi errori sono causati da piccoli difetti nella programmazione che vengono chiamati security smell. I security smell possono, alle volte, non avere conseguenze negative, ma meritano comunque sempre attenzione in quanto possono essere indicatori di un problema. Infatti, molto spesso sono la causa di debolezze nella sicurezza di un sistema che può essere attaccato da utenti malintenzionati attratti da uno o più security smell. L'obiettivo di questo lavoro di tesi è quindi quello di analizzare i security smell, le infrastrutture, le metodologie e le tecniche di sviluppo in cui essi sono maggiormente frequenti ed individuare delle tecniche di programmazione che possano evitare l'insorgere dei security smell. Tale analisi è stata condotta mediante una revisione sistematica della letteratura nella quale sono stati analizzati venti security smell, quattro ambiti nei quali sono maggiormente frequenti e numerose tecniche ad hoc per mitigarli.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

1.1 Contesto Applicativo

Ai giorni nostri, i software sono strumenti fondamentali utilizzati ricorrentemente nel quotidiano che consentono di risolvere una o più esigenze specifiche. Per questo motivo, progettazione e sviluppo software sono attività importanti e richiedono una certa attenzione al fine di garantire un buon servizio per chi ne fruisce. A tal proposito, è necessario, in fase di sviluppo, non introdurre errori nel codice che possono generare problemi di sicurezza. Infatti, in fase di programmazione, possono essere introdotte vulnerabilità dovute a scelte errate nell'implementazione di codice oppure dovute all'utilizzo di istruzioni che facilitano lo svolgimento di attacchi informatici da parte di utenti malintenzionati. Un esempio di un attacco portato a buon fine grazie alle vulnerabilità prodotte in fase di programmazione è quello condotto da Robert Morris nel 1988, noto come Internet Worm: si stima che abbia colpito circa 6000 computer. Per questi motivi, conoscere meglio le debolezze che possono essere introdotte nella programmazione di codice e che possono inficiare la sicurezza di un software è di notevole importanza. In questa revisione sistematica della letteratura, quindi, andremo ad approfondire il concetto dei security smell.

1.1.1 Cosa sono i security smell?

I security smell sono schemi di codifica ricorrenti indicativi di debolezze della sicurezza introdotti inavvertitamente dagli sviluppatori e che possono potenzialmente portare a violazioni della sicurezza. Un security smell può non avere conseguenze negative, ma merita comunque sempre attenzione poiché può essere indicatore di un problema [1].
I security smell sono diversi dalle vulnerabilità, in quanto sono schemi di codifica che indicano difetti. Una vulnerabilità, infatti, viene definita come una debolezza che può essere utilizzata per modificare e accedere a dati non intenzionalmente, interrompere la corretta esecuzione di un software e consentire al completamento di azioni senza le dovute autorizzazioni [2]. Nonostante ciò, i security smell possono portare (o meno) ad una suscettibilità, ma meritano, sempre, attenzione, ispezione, e se necessario, mitigazione [2].

1.2 Motivazioni e Obiettivi

Come si evince dal paragrafo precedente, l'obiettivo principale di questa tesi di ricerca è quello di approfondire il concetto dei security smell in quanto possono compromettere la sicurezza di un software. Ad oggi in letteratura non vi sono numerosi studi condotti sui security smell; l'obiettivo di questo studio quindi si fonda su questa evidenza, con lo scopo di fornire maggiori indicazioni circa i security smell nei vari ambiti in cui sono stati maggiormente riscontrati, e, laddove possibile, prevenirli o mitigarli.

1.3 Struttura della tesi

Il seguente lavoro di tesi si compone di quattro capitoli:

- **Capitolo 1: Introduzione**, che illustra il contesto applicativo e gli obiettivi di questa revisione sistematica della letteratura;
- **Capitolo 2: Metodologie di ricerca**, che illustra le domande di ricerca e le strategie di ricerca utilizzate per raggiungere l'obiettivo principale;
- **Capitolo 3: Analisi dei risultati**, che illustra i risultati raggiunti per ciascuna domanda di ricerca;
- **Capitolo 4: Conclusioni**, che fornisce una sintesi della ricerca e introduce possibili lavori futuri.

CAPITOLO 2

Design dello studio

L'obiettivo dello studio è esaminare la letteratura di ricerca che ha approfondito il tema dei security smell, debolezze nella progettazione del codice che possono compromettere la sicurezza dei sistemi in cui sono presenti[1]. Questo studio ha lo scopo di fornire ai lettori una panoramica sui security smell e le loro tipologie, gli ambiti in cui sono maggiormente presenti, con quale frequenza si presentano, e come poter ovviare a questi difetti nella programmazione. Per raggiungere tale scopo, è stata condotta una Systematic Literature Review (SLR), che è un processo di sintesi attraverso il quale i documenti di ricerca esistenti su un argomento di interesse vengono sistematicamente identificati, selezionati e analizzati dalla critica per esaminare l'insieme degli studi di ricerca che hanno contribuito alla definizione e all'evoluzione del campo d'interesse.

2.1 Obiettivi e Domande della ricerca

L'obiettivo specifico di ricerca della revisione sistematica della letteratura è riportato di seguito:

◎ **Obiettivo:** Identificare le tipologie di security smell e le loro caratteristiche.

Questo obiettivo ha guidato la definizione delle domande di ricerca:

🔍 **RQ$_1$.** *Quali security smell sono stati definiti dalla letteratura esistente?*

Questa domanda mira ad affrontare quali sono i security smell definiti dalla letteratura esistente. Il termine security smell è strettamente correlato alla terminologia informatica "bad coding practice"[1]. Per rispondere a questa domanda si è proceduto ad eseguire la search-query sui tre database di ricerca scelti. I security smell raccolti sono stati analizzati e catalogati. Nel capitolo 3 saranno discussi i risultati inerenti a questa domanda di ricerca.

🔍 **RQ$_2$.** *In quali infrastrutture, metodologie o tecniche di sviluppo sono più frequenti i security smell, e come cambiano in essi?*

Questa domanda mira ad affrontare in quali infrastrutture, metodologie o tecniche di sviluppo sono più frequenti i security smell e come possono variare in essi. Attraverso un'approfondita analisi dei documenti sono stati trovati diversi ambiti in cui si manifestano più frequentemente i security smell. Nel capitolo 3 saranno discussi i risultati inerenti a questa domanda di ricerca.

🔍 **RQ$_3$.** *Quali tecniche si possono adottare per correggere il codice affinché non si presentino security smell?*

Questa domanda mira ad affrontare le tecniche che si possono adottare per correggere il codice affinché si possa evitare l'insorgere di security smell in esso.

2.2 Query di ricerca

Uno dei passaggi metodologici chiave di una revisione sistematica della letteratura è l'identificazione di termini di ricerca appropriati che possono aiutare a recuperare un insieme completo di informazioni. A tal proposito, è stata adottata la seguente strategia:

1. Per ogni domanda di ricerca sono state estratte le parole chiave più rilevanti;
2. È stato utilizzato l'operatore booleano OR per comporre la query di ricerca;

Il risultato della query di ricerca è il seguente:

"Security Smell" $(\vee)$ **"Security Antipattern"** $(\vee)$ **"Security Anti-pattern"** $(\vee)$ **"Security Design issue"**

Come si evince dalla query, sono stati messi in OR $(\vee)$ tutte le parole chiave scelte inerenti al tema dei security smell. Prima di continuare la revisione sistematica della letteratura, sono state analizzate attentamente tutte le parole chiave inserite nella query di ricerca con lo scopo di verificare se queste potessero fornire una mappatura completa dei termini che sono maggiormente utilizzati in letteratura circa il tema dei security smell. Questo passaggio è stato fondamentale in quanto ha confermato che tutte le keyword inerenti ai security smell sono state individuate e non è stato quindi necessario modificare la query di ricerca.

2.3 Sorgenti di ricerca

Una volta definita la query di ricerca, sono stati selezionati i database da utilizzare durante l'esecuzione della ricerca. La corretta identificazione di tali database è indispensabile al fine di avere una revisione sistematica della letteratura di successo. Per questo motivo, sono stati selezionati i seguenti database di ricerca:

ACM Digital Library (https://dl.acm.org/)

Scopus (https://www.scopus.com/)

IEEEXplore (https://ieeexplore.ieee.org/)

Questi database sono utilizzati per condurre revisioni sistematiche della letteratura e garantiscono quindi una copertura completa della ricerca pubblicata, consentendoci quindi di accedere all'intero set di articoli.

2.4 Criteri di selezione delle risorse

I criteri di esclusione e inclusione consentono la selezione di risorse che affrontano le domande di ricerca di una revisione sistematica della letteratura. Nel contesto di questo lavoro di tesi, sono stati identificati e applicati i seguenti criteri di "Esclusione/Inclusione".

2.4.1 Criteri di esclusione

Le risorse che hanno soddisfatto i seguenti vincoli sono state escluse dallo studio condotto:

- Articoli non scritti in inglese;
- Articoli con lunghezza inferiore a 7 pagine;
- Documenti duplicati;
- Articoli il cui testo non era disponibile;
- Scarsa comprensibilità;
- Non coerente con gli argomenti trattati.
- Fonte non specificata

Attraverso l'uso di questi filtri si vanno ad escludere tutti i risultati non inerenti alla nostra ricerca, come, ad esempio, articoli di cui non si conosce la provenienza.

2.4.2 Criteri di inclusione

Nello studio sono stati inclusi i documenti che:

- Esplicitano quali security smell vengono trattati;
- Esplicitano gli ambiti in cui i security smell sono stati riscontrati e la loro frequenza in essi;
- Esplicitano tecniche per la risoluzione dei security smell;

2.5 Estrazione dei dati

Dopo aver identificato tutte le risorse finali da considerare, sono state estratte le informazioni di maggior rilievo dagli articoli analizzati per rispondere alle domande di ricerca. Nello specifico, è stata definita una tabella chiamata "Data Extraction Form" (vedi Tabella 2.1) nella quale, per ogni articolo preso in considerazione, sono state estratte le informazioni più rilevanti al fine di rispondere alle domande di ricerca. Infatti, le informazioni riportate nel Data Extraction Form, sono:

1. Nome Paper
2. Autori
3. Database
4. Categorie analizzate (Infrastrutture, Metodologie o Tecniche di sviluppo)
5. Linguaggio di Programmazione
6. Tipo di studio condotto
7. Risultati individuati

Queste informazioni, quindi, ci permetteranno di rispondere in modo esaustivo alle domande di ricerca e di ottenere i risultati attesi.
A tal proposito, sono stati generati dei grafici che saranno riportati nel capitolo Analisi dei Risultati.

Data Extraction Form						
Nome Paper	Autori	Database	Categorie Analizzate	Linguaggio	Tipo di Studio	Risultati

Tabella 2.1: Informazioni estratte da ogni articolo.

2.6 Analisi dei dati

In questa sezione viene chiarito come i dati, una volta estratti, sono stati sintetizzati, combinati ed analizzati per rispondere alle tre domande di ricerca (RQ).

2.6.1 Sintesi dei dati

I dati sono stati analizzati e sintetizzati prendendo in considerazione solo la parte degli articoli che risponde alle domande di ricerca.
In particolare, per ogni documento, sono stati considerati:

- Gli articoli nei quali sono stati analizzati e spiegati i security smell;
- Gli articoli nei quali sono state analizzate le infrastrutture, le metodologie o le tecniche di sviluppo in cui si presentano maggiormente i security smell e con che frequenza si manifestano;
- Gli articoli nei quali sono state analizzate tecniche comuni per la risoluzione dei security smell;

2.6.2 Combinazione dei dati

I dati sono stati combinati andando ad inserire quest'ultimi in una tabella (Coding) nella quale sono state riportate le domande di ricerca (RQ1, RQ2, RQ3).

2.6.3 Analisi dei dati

I dati, infine, sono stati analizzati per rispondere alle domande di ricerca (RQ1, RQ2 e RQ3)
I dati e gli articoli relativi all'esecuzione dell'analisi della letteratura sono disponibili e accessibili online.

CAPITOLO 3

Analisi dei risultati

Negli ultimi anni i security smell hanno iniziato a ricevere particolare attenzione. Una delle motivazioni risiede nello sviluppo di strumenti in grado di rilevare tali smell: infatti Saveedra e Ferreira [3] hanno proposto un framework chiamato GLITCH in grado di rilevare automaticamente i security smell in ambito IaC (Infrastructure as Code). Attualmente GLITCH supporta il rilevamento di nove security smell negli script scritti in Ansible, Chef o Puppet, tre linguaggi che differiscono tra loro per quanto riguarda l'ordine di esecuzione, la manutenzione del codice e la necessità di installare un software aggiuntivo, lo stile e la sintassi [4].
Rahman et al. hanno invece costruito uno strumento di analisi statistica chiamato Security Linter for Infrastructure as Code scripts (SLIC) per identificare automaticamente la presenza dei security smell negli script IaC [1] [2].
Siddiq et al. hanno utilizzato due analizzatori statici (Bandit e Pylint) per rilevare i security smell nei campioni di set di dati di addestramento utilizzati per la generazione di codice Python [5].
Gadient et al. hanno invece analizzato manualmente applicazioni al fine di individuare security smell. Per quanto riguarda le applicazioni Mobile, le app analizzate manualmente sono state estratte dai repository software F-Droid e AndroZoo [7] [6].
Cheh et al. hanno utilizzato come caso di studio le specifiche API dell'Open Bank

Project, standard globale e soluzione API open source per l'open banking utilizzato in tutto il mondo da più di 110000 sviluppatori. Sono state estratte ed analizzate manualmente 304 chiamate API [8]. Infine Islamÿ et al. hanno analizzato manualmente 28 applicazioni Spring al fine di individuare security smell. Esse sono ospitate su GitHub: 8 di 28 sono applicazioni aziendali del mondo reale e le restanti 20 sono progetti dimostrativi con un esempio di utilizzo del framework Spring. La loro analisi, inoltre, ha scoperto 6 tipi di anti-pattern di sicurezza [9].
Questo capitolo illustra i risultati ottenuti per ogni domanda di ricerca e i risultati relativi all'obiettivo principale.

3.1 Esecuzione del processo di ricerca

Una volta definiti i blocchi fondamentali della revisione sistematica della letteratura, si è proceduto alla sua esecuzione. In particolare, il processo di esecuzione è stato così svolto:

1. È stata eseguita la query di ricerca sui tre database selezionati. La query di ricerca ha prodotto un totale di 40 risultati, ottenuti come segue: 9 risultati utilizzando ACM Digital Library, 6 utilizzando IEEEXplore e 25 utilizzando Scopus. Il primo passaggio è stato completato scaricando tutti i documenti selezionati ed archiviandoli in un ambiente locale per effettuare un'indagine più rapida.

2. Ciascuno dei documenti scelti è stato sottoposto all'applicazione dei criteri di esclusione. In questa fase, infatti, sono stati scansionati tutti gli articoli e sono stati applicati i filtri dei criteri di esclusione: sono stati considerati il titolo, l'abstract e le parole chiave di ciascun articolo. Complessivamente, sono stati esclusi 28 articoli, e quindi, 12 sono passati alla fase successiva.

3. Ciascuno dei documenti rimanenti è stato sottoposto all'applicazione dei criteri di inclusione. In questa fase, sono stati scansionati tutti gli articoli rimanenti e sono stati applicati i filtri dei criteri di inclusione. A differenza della fase precedente, nella quale i filtri erano stati applicati su titolo, abstract e parole

chiave, in questa l'inclusione è stata valutata sull'intero articolo. Come risultato di questa fase, sono stati scartate 3 fonti. Solo 9 articoli, quindi, sono stati inclusi nella revisione sistematica.

4. Infine, è stata effettuata l'estrazione dei dati, che è risultata abbastanza immediata dato l'esiguo numero di articoli inclusi nella revisione e l'applicazione di inclusion/exclusion criteria ben scelti.

3.2 Analisi dei risultati ottenuti - RQ1

In questo paragrafo si fornirà la risposta alla prima domanda di ricerca.

RQ1. Quali security smell sono stati definiti dalla letteratura esistente?

Per rispondere alla domanda di ricerca (RQ1) sono stati analizzati i dati che si riferiscono alla presenza dei security smell. Come detto nel Capitolo 2, l'obiettivo di RQ1 è stato quello di ottenere una panoramica generale dei security smell che maggiormente sono stati riscontrati sia nelle repository analizzate presenti su GitHub, Mozzilla, OpenStack, Wikimedia, F-Droid, sia su applicazioni Android open/closed-source.

Il nostro studio evidenzia l'interesse nell'analizzare i security smell, la loro frequenza nei codici dei diversi ambiti in cui si presentano e le tecniche che si possono adottare al fine di evitarli. Pertanto, come vedremo più approfonditamente nel paragrafo 3.3, i security smell individuati sono stati categorizzati per infrastrutture, metodologie e tecniche di sviluppo alle quali appartengono. Questi ambiti sono: "Infrastructure as Code (IaC)", "Mobile", "Spring Security", "Programmazione generale per la generazione di codice."

Quindi, i security smell che sono stati identificati sono:

3.2.1 IaC: Infrastructure as Code [1, 2, 3, 4]

- Admin by default (CWE-250): accade quando viene assegnato admin come utente predefinito;

- Password vuota (CWE-258): accade quando viene utilizzata una stringa di lunghezza zero come password;
- Segreto hard-coded (CWE 259, CWE 798): accade quando vengono rilevate informazioni sensibili come nome utente o password negli script;
- Indirizzo IP senza restrizioni (CWE-284): accade quando viene assegnato 0.0.0.0 come indirizzo IP;
- Commento sospetto (CWE-546): accade quando vengono inseriti nei commenti informazioni sui difetti o vulnerabilità;
- Uso di HTTP senza SSL/TLS (CWE-319): accade quando non vengono utilizzati i layers SSL/TLS;
- Nessun controllo di integrità (CWE-353): accade quando non viene controllato il contenuto scaricato;
- Uso di algoritmi di crittografia deboli (CWE 326, CWE 327): accade quando vengono utilizzati algoritmi di crittografia classificati "deboli", come SHA-1;
- Default mancante nella dichiarazioni di istruzioni switch-case (CWE-478): accade quando non si tiene conto di tutte le combinazioni di input nel momento in cui si implementa una logica condizionale di tipo switch-case;

3.2.2 Programmazione generale per la generazione di codice [5]

- Uso di assert (CWE-703): accade quando viene fatto uso ricorrente di assert nel codice;
- Funzioni in lista nera (CWE-78, CWE-330): accade solitamente quando vengono utilizzate funzioni che non gestiscono il problema della sincronizzazione tra processi;
- Sottoprocesso senza shell Uguale Vero (CWE-78): accade quando viene utilizzato il modulo subprocess, spesso in Python, per eseguire una nuova applicazione;

3.2.3 Mobile [6, 7, 8]

- Canale di trasporto non sicuro: accade quando si sceglie di utilizzare HTTP al posto di HTTPS;
- Divulgazione del codice sorgente: accade quando i messaggi di errore lasciano trapelare informazioni sensibili sul sistema in esecuzione;
- Divulgazione di informazioni sulla versione: accade quando le intestazioni HTTP forniscono informazioni sensibili sull'architettura o sulla configurazione del sistema in esecuzione;
- Mancanza di controllo degli accessi: accade quando si può effettuare l'accesso senza alcun tipo di identificazione o mediante un'identificazione debole;
- Reindirizzamenti HTTPS mancanti: accade quando non si viene indirizzati a connessioni crittografate anche se supportate;
- HSTS mancante: accade quando i server non sfruttano la funzione HSTS;

3.2.4 Spring Security [9]

- Utilizzo di BCrypt con forza predefinita 10: accade quando si utilizza BCrypt come funzione di hashing e si utilizza una forza crittografica minore di 16, spesso 10;
- Richieste illimitate durante invocazione API: accade quando non si limitano le richieste degli utenti durante le invocazioni API;
- Segreto hard-coded (CWE 259, CWE 798): accade quando vengono rilevate informazioni sensibili come nome utente o password negli script;
- Uso di algoritmi di crittografia deboli (CWE 326, CWE 327): accade quando vengono utilizzati algoritmi di crittografia classificati "deboli", come MD5;

Nella figura 3.1 possiamo notare la percentuale con cui i security smell individuati sono presenti nelle infrastrutture, metodologie o tecniche di sviluppo citate.

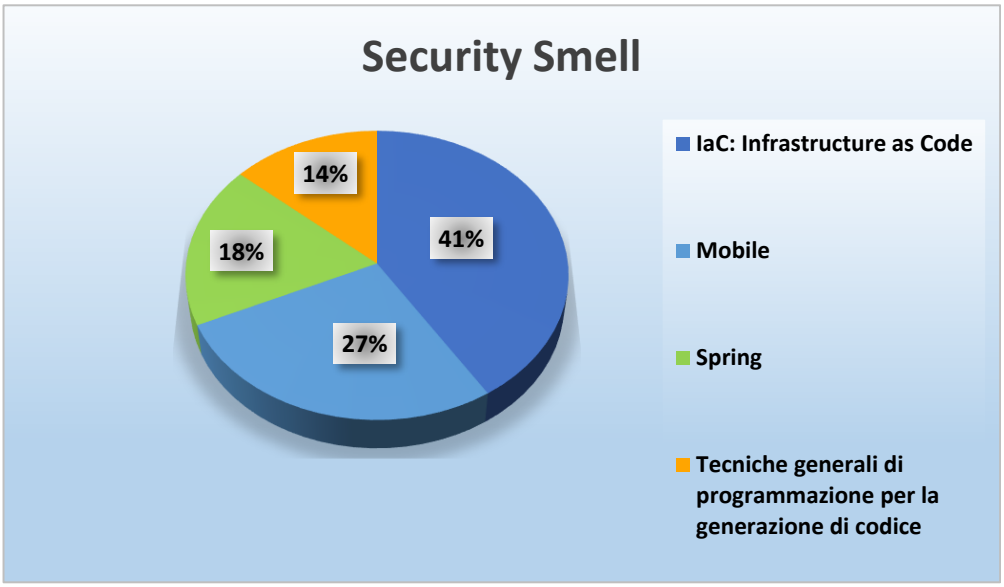

Figura 3.1: Percentuale con cui i security smell individuati sono presenti nelle infrastrutture, metodologie o tecniche di sviluppo citate

3.2.5 Dettagli sui security smell e CWE

Come si evince dal paragrafo 3.2, alcuni security smell individuati sono affiancati da una denominazione, CWE. Il Common Weakness Enumeration (comunemente chiamato CWE) è un sistema di categorie per le debolezze e le vulnerabilità di hardware e software. È sostenuto da un progetto comunitario con l'obiettivo di comprendere i difetti del software e dell'hardware e di creare strumenti automatizzati che possano essere utilizzati per identificare, correggere e prevenire tali difetti. Il progetto è sponsorizzato dal National Cybersecurity FFRDC, gestito da The MITRE Corporation, con il supporto di US-CERT e della National Cyber Security Division del Dipartimento della Sicurezza Interna degli Stati Uniti.
Alcuni security smell, quindi, vengono analizzati e riconosciuti più facilmente dalla comunità della sicurezza in quanto strettamente collegati a debolezze o falle nel codice precedentemente riscontrate dal CWE. Infatti, una mappatura tra uno smell e una debolezza di sicurezza riportata dal CWE può convalidare il processo qualitativo

[4]. I security smell collegati a debolezze riconosciute dal CWE che sono state citate in questa revisione sistematica della letteratura sono:

1. **CWE-250 (Execution with Unnecessary Privileges [1], [2], [3], [4])**: debolezza che viola la proprietà del "Principio del minimo privilegio". Solitamente accade quando viene assegnato admin come utente predefinito. Questa debolezza è collegata al security smell *Admin by Default*.

2. **CWE-258 (Password vuota nei file di configurazione [1], [2], [3], [4])**: debolezza riscontrata quando vi è una stringa di lunghezza zero come password. Questa debolezza è collegata al security smell *Password vuota*.

3. **CWE-259, CWE-798 (Use of Hard-Coded Password - Use of Hard-Coded Credentials [1], [2], [3], [4])**: debolezze riscontrate quando vi sono rilevazioni di informazioni sensibili negli script, come nome utente o password. Queste debolezze sono collegate al security smell *Segreto Hard-Coded*.

4. **CWE-284 (Improper Access Control [1], [2], [3], [4])**: debolezza riscontrata quando vi si assegna l'indirizzo 0.0.0.0 per un server di database o un servizio/istanza cloud. Il binding all'indirizzo 0.0.0.0 può infatti causare problemi di sicurezza poiché questo indirizzo può consentire connessioni da ogni rete possibile. Questa debolezza è collegata al security smell *Indirizzo IP senza restrizioni*.

5. **CWE-546 (Commento Sospetto [1], [2], [3], [4])**: debolezza riscontrata quando vengono inseriti nel codice commenti che contengono informazioni sulla presenza di bug, funzionalità mancanti o debolezze nel sistema (ad es. "TODO", "FIXME", "HACK"). Questa debolezza è collegata al security smell *Commento Sospetto*.

6. **CWE-319 (Cleartext Transmission of Sensitive Information [1], [2], [3], [4])**: debolezza riscontrata nell'utilizzo di HTTP senza Transport Layer Security o Secure Sockets Layer. Rende la comunicazione meno sicura in quanto più suscettibile ad attacchi man-in-the-middle (attacco informatico in cui qualcuno segretamente ritrasmette o altera la comunicazione tra due parti che credono di

comunicare direttamente tra di loro.) Questa debolezza è collegata al security smell *Uso di HTTP senza SSL/TLS*.

7. **CWE-353 (Missing support for Integrity Check [2], [3], [4])**: debolezza riscontrata quando non viene controllato il contenuto scaricato (ad esempio attraverso checksum). Questa debolezza è collegata al security smell *Nessun controllo di integrità*.

8. **CWE-326, CWE-327 (Forza di crittografia inadeguata - Uso di un algoritmo crittografico rotto o rischioso [1], [2], [3], [4])**: debolezze riscontrate quando vengono utilizzati algoritmi di crittografia deboli (come MD5 o SHA-1) o quando vengono utilizzati parametri relativi alla forza di crittografia inadeguati. Queste debolezze sono collegate ai security smell *Utilizzo di BCrypt con forza predefinita 10 e Uso di algoritmi di crittografia deboli*.

9. **CWE-478 (Missing Default Case in Switch Statement [2], [3], [4])**: debolezza riscontrata quando non si tiene conto di tutte le combinazioni di input che possono verificarsi nell'implementazione di una logica condizionale di tipo switch-case. A causa di questo schema di codifica, un utente malintenzionato può indovinare un valore che non viene gestito dalle dichiarazioni dello switch-case e generare un errore. Tale errore può fornire all'attaccante informazioni non autorizzate sul sistema in termini di tracce di stack o errori di sistema. Questa debolezza è collegata al security smell *Default mancante nella dichiarazione di istruzioni switch-case*.

10. **CWE-703 (Improper Check or Handling of Exceptional Conditions [5])**: debolezza riscontrata quando viene fatto uso di assert nel codice di produzione (viene infatti considerato una bad practice). Può essere infatti sostituito con una corretta gestione delle clausole try-except. Questa debolezza è collegata al security smell *Uso di assert*.

11. **CWE-330 (Use of Insufficiently Random Values [5])**: BANDIT,[1] , strumento che individua problemi nella sicurezza di uso comune in Python, ha riscontrato

[1] https://bandit.readthedocs.io/en/latest/

che i generatori pseudocasuali standard di Python possono essere utili per generare numeri casuali, ma non sono adatti alla crittografia. Un generatore di numeri pseudocasuali sicuro dal punto di vista crittografico è un generatore di numeri casuali che utilizza meccanismi di sincronizzazione per garantire che due processi non generino lo stesso numero casuale nello stesso momento. La funzione random di Python non è quindi adatta a questo scopo. Questa debolezza è collegata al security smell *Funzioni in lista nera*.

12. **CWE-78 (OS Command Injection [5])**: In Python, il modulo *subprocess* viene utilizzato per eseguire una nuova applicazione creando un nuovo processo. Quando il parametro shell è uguale a true, il codice viene eseguito attraverso la shell del sistema (ad esempio con /bin/sh). Ciò viene considerato una bad practice in quanto può aumentare la frequenza di attacchi di tipo shell injection. Questa debolezza è collegata al security smell *Sottoprocesso senza shell Uguale Vero*.

I restanti security smell per cui non si è riuscito ad effettuare una mappatura con le debolezze riscontrate dal CWE [6, 7] sono:

1. **Canale di trasporto non sicuro**: La comunicazione web si basa su HTTP o HTTPS: entrambe le varianti esistono in configurazioni di server app. Il problema riscontrato è che HTTP non fornisce alcuna sicurezza: nè l'indirizzo, nè le informazioni di intestazione o il payload sono crittografati, cosa che invece è insita nel protocollo HTTPS. Ce ne si può accorgere quando l'URL inizia con http://.

2. **Divulgazione del codice sorgente**: I messaggi di errore lasciano trapelare informazioni preziose sull'implementazione del sistema in esecuzione. Il problema riscontrato è che i messaggi di errore vengono trasmessi come testo in chiaro nel corpo di risposta del messaggio del server. Tale messaggio rende note informazioni come i nomi dei metodi utilizzati, i numeri di riga e i percorsi dei file, rivelando quindi la struttura del file system interno e la configurazione del server. Ce ne si può accorgere quando il corpo HTTP restituito contiene una

traccia di stack o un frammento di codice che mostra lo snippet problematico. I termini relativi ai crash delle applicazioni sono comuni, ad esempio "stack", "trace" e "error".

3. **Divulgazione di informazioni sulla versione**: Oltre ai parametri utili per la connessione, le intestazioni HTTP forniscono informazioni sull'architettura e sulla configurazione del software di un sistema in esecuzione. Il problema riscontrato è che un software obsoleto soffre di gravi vulnerabilità di sicurezza. Ad esempio, un server che restituisce X-PoweredBy: PHP/5.5.23 nell'intestazione, divulga informazioni importanti circa la sua configurazione ed è esposto più facilmente ad attacchi. Ce ne si può accorgere quando nell'intestazione della risposta è presente una delle seguenti chiavi: engine,server,x-aspnet-version o x-powered-by.

4. **Mancanza di controllo degli accessi**: L'autenticazione tramite nome utente e password offre agli utenti finali esperienze personalizzate, ad esempio registri di chat individuali o liste di amici, e allo stesso tempo consente il controllo degli accessi per separare e proteggere i dati sensibili degli utenti. Il problema riscontrato è che l'accesso a dati o azioni sensibili non è limitato da un meccanismo di autenticazione forte, come una coppia di nome utente e password, ma vengono utilizzati identificatori facili da creare o nessun dato di identificazione per proteggere l'accesso. Ce ne si può accorgere quando un server non risponde con il codice di stato 401 Unauthorized o 403 Forbidden. In altre parole, il server risponde sempre con successo senza richiedere alcuna credenziale.

5. **Reindirizzamenti HTTPS mancanti**: Ci sono server che non reindirizzano i client a connessioni crittografate, anche se sono supportate. Il problema riscontrato è quindi che i server delle app non reindirizzano le connessioni HTTP in entrata verso HTTPS quando queste tentano di connettersi. Ce ne si può accorgere quando, per una richiesta HTTP, un server non fornisce un messaggio di reindirizzamento HTTP 3xx che punta all'implementazione HTTPS. Quindi, per una richiesta HTTPS, il server fornisce un messaggio di reindirizzamento HTTP.

6. **HSTS mancante**: Le informazioni dell'intestazione HTTP vengono utilizzate per impostare correttamente la connessione, specificando vari parametri di comunicazione, come ad esempio le lingue accettabili, la compressione utilizzata o l'imposizione di HTTPS per i futuri tentativi di connessione, una caratteristica chiamata HTTP Strict Transport Security (HSTS). L'HSTS fornisce una protezione contro gli attacchi HTTPS to HTTP downgrading, ovvero quando un utente accede ad una risorsa web in un ambiente sicuro (come casa o a lavoro), il client sa che la risorsa deve essere accessibile solo tramite HTTPS. Se ciò non è possibile, ad esempio in un aeroporto in cui un aggressore tenta di eseguire attacchi man-in-the-middle, il client visualizza un errore di connessione. Pertanto, l'HSTS dovrebbe essere utilizzato in combinazione con i reindirizzamenti da HTTP a HTTPS, poichè l'intestazione HSTS è considerata valida solo se inviata tramite connessioni HTTPS. Il problema riscontrato è che i server non sfruttano la funzione HSTS o non utilizzano i parametri consigliati. Ce ne si può accorgere quando un server non fornisce l'intestazione HTTP HSTS Strict-Transport-Security: max-age=31536000 o includeSubDomains per una richiesta HTTPS.

7. **Richieste illimitate durante invocazione API**: debolezza riscontrata quando non si offre alcuna politica di limitazione per ridurre il numero di richieste da parte degli utenti durante l'invocazione dell'API. Questa insufficienza potrebbe portare ad attacchi Denial of Service (DoS).

3.3 Analisi dei risultati ottenuti - RQ2

In questo paragrafo si fornirà la risposta alla seconda domanda di ricerca.

RQ2. In quali infrastrutture, metodologie e tecniche di sviluppo sono più frequenti i security smell, e come cambiano in essi? Per rispondere alla domanda di ricerca (RQ2) sono stati analizzati i dati che si riferiscono alle infrastrutture, metodologie e teniche di sviluppo in cui sono maggiormente frequenti i security smell, e con che frequenza essi si presentano. Negli ultimi anni sono stati condotti degli studi sui security smell, maggiormente revisioni sistematiche della letteratura. Alcuni di essi ([1], [2], [3], [4]) hanno affrontato il problema dei security smell in ambito IaC (Infrastructure as Code), identificando nove security smell. Poichè gli script IaC (noti anche come script di configurazione o script di configurazione di codice) - i cui fornitori più commerciali sono Ansible, Chef, Puppy - sono utilizzati per fornire e configurare server e sistemi basati su cloud, i security smell presenti in questi script potrebbero essere utilizzati per consentire ad utenti malintenzionati di sfruttare le vulnerabilità dei sistemi forniti. Altri invece ([6], [7], [8]), hanno affrontato il problema dei security smell in ambito Mobile, indentificando sei security smell. La comunicazione web ha già ricevuto molta attenzione da parte della comunità della sicurezza, con conseguente miglioramento del supporto degli strumenti. Ciò nonostante, le configurazioni dei server delle applicazioni e i server di navigazione hanno ricevuto molta meno attenzione, ragion per cui i security smell in ambito Mobile sono sempre più frequenti. Inoltre, Mazharul Islamÿ, Sazzadur Rahamanÿ et al. [9] hanno affrontato il problema dei security smell in ambito Spring Security conducendo un'analisi manuale di 28 Applicazioni Spring. La sicurezza di Spring è altamente personalizzabile in base alla progettazione per consentire un'integrazione perfetta con molteplici casi d'uso. Sfortunatamente, l'abuso di tali capacità di personalizzazione può essere una grande fonte di insicurezza delle applicazioni. Senza un'attenta considerazione, tale personalizzazione può rendere un'applicazione Web vulnerabile. Infine, l'ultimo studio rimanente ([5]) ha affrontato il problema dei security smell in ambito della programmazione generale per la generazione di codice (Python), identificando tre security smell. Le tecniche di generazione del codice mirano a generare automaticamente codice funzionale sulla base di richieste;

recentemente, si è assistito a un aumento delle tecniche basate sull'apprendimento automatico. Sebbene un buon strumento di generazione del codice possa aiutare i programmatori a ridurre gli sforzi di sviluppo, gli snippet di codice utilizzati per addestrare queste tecniche sono in genere presi da repository open-source.

Quindi, le infrastrutture, le metodologie e le tecniche di sviluppo che sono state identificate sono:

- **IaC: Infrastructure as Code** [1, 2, 3, 4] con una frequenza di security smell pari al 26,5% (vedi Figura 3.2). Questo dato è stato ottenuto analizzando gli script Ansible e Chef. Infatti, i security smell identificati sono presenti in tutti i set di dati e, per Ansible, nei dataset GitHub e Openstack analizzati si è osservato rispettivamente che il 25,3% e il 29,6% degli script totali contiene almeno uno dei security smell sopracitati. Per Chef, invece, nei dataset GitHub e Openstack analizzati si è osservato rispettivamente che il 20,5% e il 30,4% degli script totali contiene almeno uno dei security smell sopracitati. Più precisamente, nel caso degli script Ansible, sono state osservate 18353 occorrenze di security smell, mentre per Chef 28247 occorrenze. Un altro dato considerevole è che 15131 dei security smell riscontrati in Ansible sono occorrenze di Segreti-Hard-Coded (quindi l'82,4% dei security smell totali identificati in Ansible), di cui il 55,9%, il 37,0% e il 7,1% sono rispettivamente: chiavi hard-coded, nomi utente e password. Per Chef invece sono state riscontrate 15363 occorrenze di Segreti-Hard-Coded (quindi il 54,4% dei security smell totali identificati in Chef), di cui il 47,0%, l'8,9% e il 44,1% sono rispettivamente: chiavi hard-coded, nomi utente e password. Un importante risultato di quest'accurata analisi circa le informazioni riscontrate è che il security smell più frequente nell'ambiente di sviluppo IaC (Infrastructure as Code) è: *Segreto Hard-Coded (nome utente - password)*. Infine, si è stimato che la percentuale di script che presenta almeno un security smell rispettivamente in GitHub, Mozzilla, Openstack e Wikimedia è: 29,3%, 17,9%, 32,9%, 26,7% e che la persistenza media di un security smell in GitHub, Mozzilla, Openstack e Wikimedia è rispettivamente: 92, 77, 89 e 98 mesi.

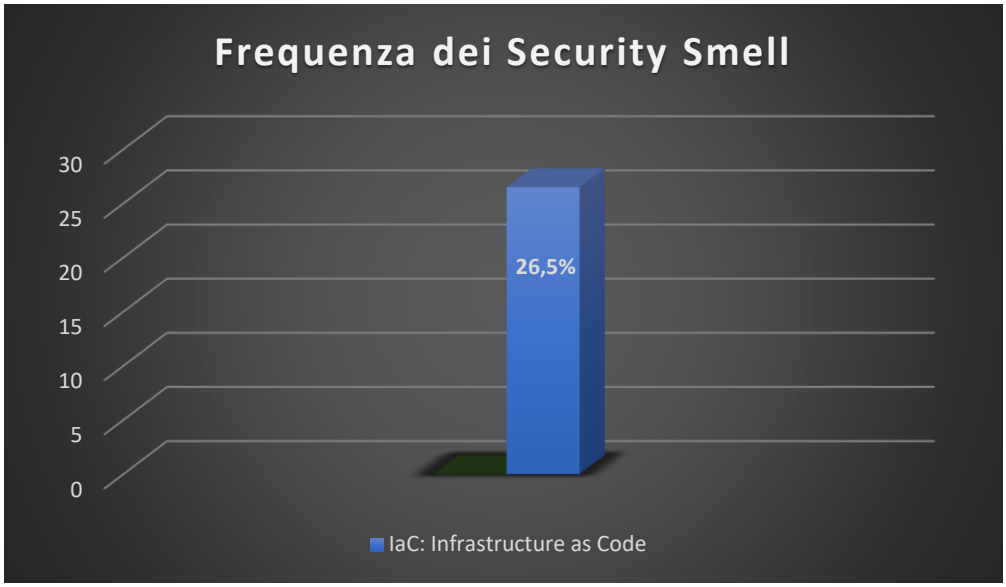

Figura 3.2: Percentuale delle occorrenze dei security smell presente negli script analizzati in ambito Infrastructure as Code (IaC).

- **Mobile** [6, 7, 8] con una frequenza di security smell pari al 25% (vedi Figura 3.3). Questo dato è stato ottenuto analizzando applicazioni sia closed-source gratuite scaricate da Google Play Store (3073) e sia applicazioni open-source ottenute tramite repository software F-Droid (303). Gli errori di sicurezza dei server delle applicazioni rappresentano una grave minaccia. La maggior parte degli errori di sicurezza è presente in oltre il 25% di tutte le applicazioni indipendentemente dal fatto che l'applicazione sia open-source o closed-source. Particolarmente allarmante è la scoperta che le applicazioni closed-source soffrono di problemi di sicurezza 1,6 volte in più rispetto a quelle open-source. Inoltre, secondo una ricerca, i server delle app vengono solitamente configurati una volta e poi non vengono più toccati. Questo paradigma introduce gravi rischi per la sicurezza e causa del software obsoleto che gira su interfacce pubblicamente accessibili. Di conseguenza, i dati sensibili degli utenti potrebbero essere estorti quando gli

avversari applicano gli exploit adatti a questi sistemi.
Infine, un dato considerevole è che il Security Smell più frequente nelle applicazioni Mobile è *Canale di Trasporto non sicuro* con una frequenza pari al 50%.

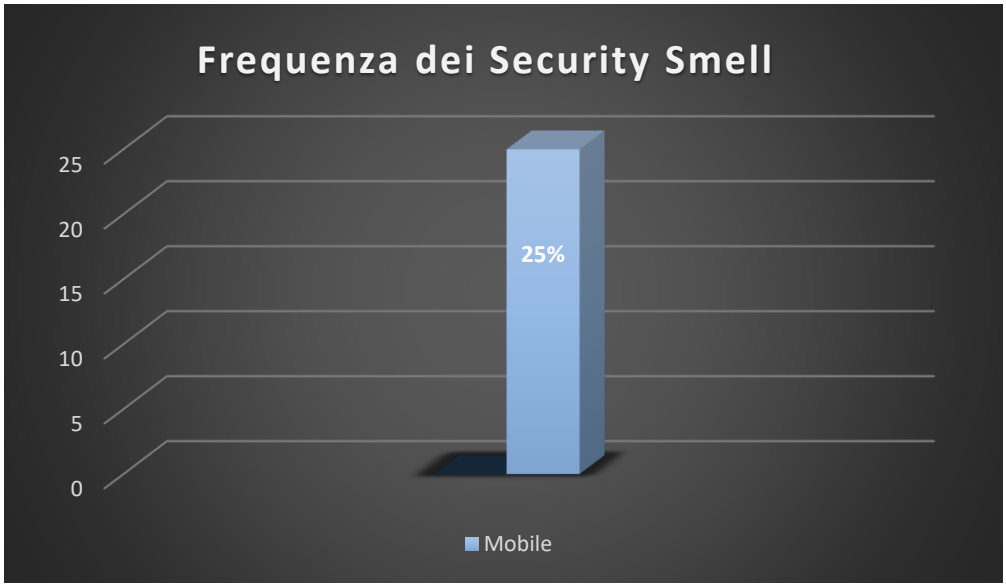

Figura 3.3: Percentuale delle occorrenze dei security smell presente nelle applicazioni analizzate in ambito Mobile.

- **Spring Security** [9] con una frequenza di security smell pari al 35% (vedi Figura 3.4). Questo dato è stato ottenuto analizzando 28 applicazioni Spring Security: sono stati identificati 10 occorrenze di security smell o comportamenti definiti non sicuri che possono causare attacchi CSRF (Cross-site Request Forgery) e man-in-the-middle. Infine, un dato considerevole è che il Security Smell più frequente nelle applicazioni Spring è *Uso di algoritmi di crittografia deboli*. Infatti un'analisi ha rilevato che la sicurezza di Spring utilizza "10" come forza predefinita in BCrypt durante la codifica della password, mentre si consiglia di utilizzare almeno sempre 16 per essere protetti. Si è anche scoperto che la sicurezza di Spring utilizza l'hashing MD5 considerato non sicuro. Ancora più

importante, si è identificato che la sicurezza di Spring non offre alcuna politica di limitazione per limitare il numero di richieste da parte degli utenti durante l'invocazione dell'API.

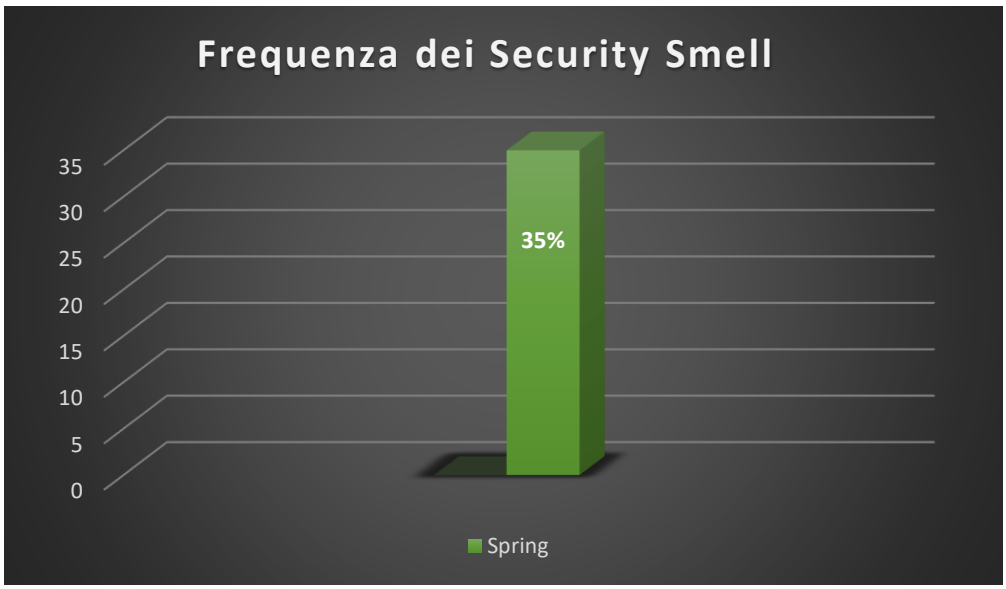

Figura 3.4: Percentuale delle occorrenze dei security smell presente nelle applicazioni analizzate in ambito Spring Security.

- **Programmazione generale per la generazione di codice** [5] con una frequenza di security smell pari al 37%. Questo dato è stato ottenuto analizzando un campione statisticamente significativo selezionando 384 output di Pylint e 380 output di Bandit (con un margine di errore del 5%). Il numero di occorrenze di security smell rilevate da Pylint è del 68,3%. Invece, il numero di occorrenze di security smell rilevate da Bandit è del 5,73%. I security smell maggiormente riscontrati in questo campione significativo sono: *Uso di Assert* e *Funzioni in lista nera*. In conclusione, la generazione automatica di codice può sicuramente aiutare gli sviluppatori a ridurre il tempo dedicato alla scrittura di codice con pattern comuni. Con l'avvento degli strumenti di generazione del codice integrati negli IDE, infatti ricerche precedenti hanno studiato che solitamente

il codice generato è funzionalmente corretto (cioè implementa ciò che l'utente si aspettava). Tuttavia questi lavori non hanno verificato la qualità del codice generato. Eseguendo uno studio empirico su larga scala su un set di addestramento si è scoperto che questi codici contengono diverse occorrenze di smell, in cui l'uso di funzioni pericolose è il security smell più ricorrente. Quindi questo studio evidenzia l'importanza di non accettare il codice generato così com'è ma di verificarne sempre la correttezza.

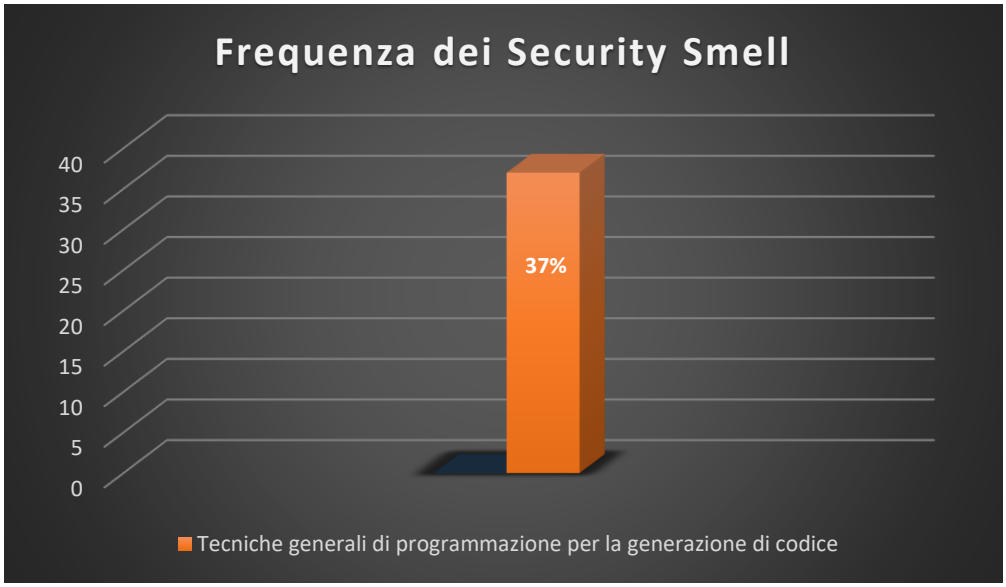

Figura 3.5: Percentuale delle occorrenze dei security smell presente nel codice analizzato in ambito Programmazione generale per la generazione di codice.

In conclusione, per rispondere alla domanda di ricerca **RQ2. In quali infrastrutture, metodologie e tecniche di sviluppo sono più frequenti i security smell e come cambiano in essi**, diremo che le infrastrutture, le metodologie e le tecniche di sviluppo analizzate sono:

- **IaC (Infrastructure as Code)** in cui sono stati individuati i seguenti security smell: Admin by Default, Password Vuota, Segreto Hard-Coded, Indirizzo IP

senza restrizioni, Commento Sospetto, Uso di HTTP senza SSL/TLS, Nessun controllo di integrità, Uso di algoritmi di crittografia deboli e default mancante nella dichiarazione di istruzioni switch-case. Il security smell più frequente negli script IaC risulta essere *Segreto Hard-Coded* (vedi Tabella 3.1).

- **Mobile** in cui sono stati individuati i seguenti security smell: Canale di trasporto non sicuro, Divulgazione del codice sorgente, Divulgazione di informazioni sulla versione, Mancanza di controllo degli accessi, Reindirizzamenti HTTPS mancanti, HSTS mancante. Il security smell più frequente nelle applicazioni Mobile risulta essere *Canale di trasporto non sicuro* (vedi Tabella 3.1).

- **Spring Security** in cui sono stati individuati i seguenti security smell: Segreto Hard-Coded, Uso di algoritmi di crittografia deboli, Richieste illimitate durante invocazione API e Utilizzo di BCrypt con forza predefinita 10. Il security smell più frequente nelle applicazioni Spring risulta essere *Uso di algoritmi di crittografia deboli* (vedi Tabella 3.1).

- **Programmazione generale per la generazione di codice** in cui sono stati individuati i seguenti security smell: Uso di Assert, Funzioni in lista nera, Sottoprocesso senza shell Uguale Vero. Il security smell più frequente nella programmazione generale per la generazione di codice (Python) risulta essere *Uso di assert* (vedi Tabella 3.1).

Categorie analizzate	Security Smell più frequente
IaC: Infrastructure as Code	Segreto Hard-Coded
Mobile	Canale di trasporto non sicuro
Spring	Uso di algoritmi di crittografia deboli
Programmazione Generale	Uso di Assert

Tabella 3.1: Security smell più diffusi.

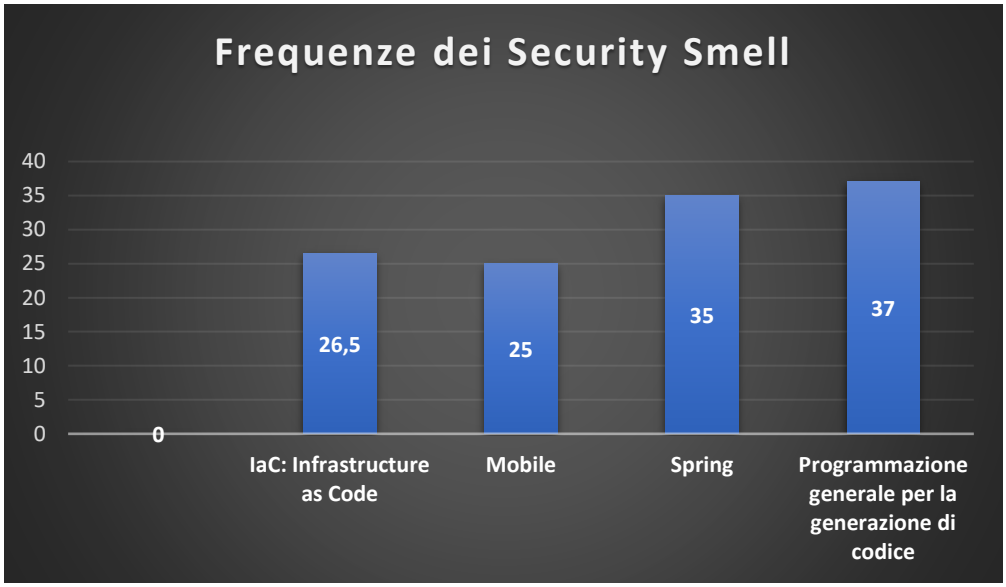

Figura 3.6: Percentuale delle occorrenze dei security smell comparate nelle varie categorie analizzate.

3.4 Analisi dei risultati ottenuti - RQ3

In questo paragrafo si fornirà la risposta alla terza domanda di ricerca. **Quali tecniche si possono adottare per correggere il codice affinché non si presentino security smell?** Per rispondere alla domanda di ricerca (RQ3) sono stati analizzati i dati che si riferiscono alle tecniche di programmazione che si possono adottare al fine di evitare le occorrenze di security smell nel codice. Doverosa è la premessa che, essendo i security smell tutti distinti tra loro, bisognerà analizzarli uno ad uno per fornire delle tecniche che ci permettano di ovviare al problema dei security smell.

Quindi, le tecniche che si possono adottare per correggere il codice affinché non si presentino security smell sono:

- **Admin by default (CWE-250) [1, 4]**: si raccomanda ai professionisti di progettare ed implementare un sistema che, come impostazione predefinita, fornisca a qualsiasi entità il minor numero di privilegi necessari.

- **Password vuota (CWE-258) [1, 4]**: si raccomanda ai professionisti di utilizzare password forti, eliminando quindi così la comparsa di password vuote.
- **Segreto hard-coded (CWE 259, CWE 798) [1, 4, 9]**: si raccomanda ai professionisti di utilizzare strumenti differenti per memorizzare i segreti hard-coded (come, ad esempio, Ansible/AWS35 o Vault).
- **Indirizzo IP senza restrizioni (CWE-284) [1, 4]**: si raccomanda ai professionisti di assegnare gli indirizzi IP in modo sistematico, in base ai servizi e alle risorse che devono essere forniti. Ad esempio, le connessioni in entrata ed in uscita per un database contenente informazioni sensibili possono essere limitate ad un determinato indirizzo IP e ad una determinata porta.
- **Commento sospetto (CWE-546) [1, 4]**: si raccomanda ai professionisti di creare delle linee guida in cui esplicitare quali informazioni memorizzare nei commenti e quali no, cercando di seguire e far seguire attentamente questi suggerimenti.
- **Uso di HTTP senza SSL/TLS (CWE-319) [1, 4]**: banalmente, si raccomanda ai professionisti di utiizzare sempre SSL e TLS.
- **Nessun controllo di integrità (CWE-353) [1, 4]**: si raccomanda ai professionisti di verificare il contenuto scaricato calcolando gli hash del contenuto o controllando le firme GPG.
- **Uso di algoritmi di crittografia deboli (CWE 326, CWE 327) [1, 4, 9]**: si raccomanda ai professionisti di utilizzare algoritmi di crittografia raccomandati dal National Istitute of Standards and Technology (ad es. SHA256 o SHA512). Si raccomanda inoltre di utilizzare forze di crittografia adeguate.
- **Default mancante nella dichiarazioni di istruzioni switch-case (CWE-478) [1, 4]**: si raccomanda ai professionisti di aggiungere sempre un blocco "else" predefinito, in modo che un input inaspettato non scateni eventi imprevisti.
- **Uso di assert (CWE-703) [5]**: si raccomanda ai professionisti di sostituire l'uso di assert (bad-practice) con una corretta gestione delle clausole try-except.

- **Funzioni in lista nera (CWE-78, CWE-330) [5]**: si raccomanda ai professionisti di utilizzare generatori di numeri pseudocasuali sicuri dal punto di vista crittografico e che non creino problemi di sincronizzazione.
- **Sottoprocesso senza shell Uguale Vero (CWE-78) [5]**: si raccomanda ai professionisti di invocare un sottoprocesso senza utilizzare la shell, evitando così possibili attacchi di tipo shell-injection.
- **Canale di trasporto non sicuro [6, 7]** : si raccomanda ai professionisti di utilizzare sempre un canale di trasporto sicuro (HTTPS).
- **Divulgazione del codice sorgente [6, 7]**: si raccomanda ai professionisti di non mettere in chiaro nei messaggi di errore informazioni rilevanti come i nomi dei metodi utilizzati, i numeri di riga, il percorso file o qualsiasi altra informazione che possa rivelare la struttura del file system interno.
- **Divulgazione di informazioni sulla versione [6, 7]**: si raccomanda ai professionisti di non mettere in chiaro nelle intestazioni HTTP informazioni sull'architettura e sulla configurazione del software del sistema in esecuzione.
- **Mancanza di controllo degli accessi [6, 7]**: si raccomanda ai professionisti di gestire gli accessi sempre mediante meccanismi di autenticazione forte (come una coppia nome utente - password).
- **Reindirizzamenti HTTPS mancanti [6, 7]**: si raccomanda ai professionisti, ove possibile, di gestire correttamente le richieste HTTP che devono essere reindirizzate ad HTTPS.
- **HSTS mancante [6, 7]**: si raccomanda ai professionisti di sfruttare la funzione HSTS e di utilizzare sempre i parametri consigliati (come, ad esempio, Strict-Transport-Security: max-age=63072000; includeSubDomains; preload).
- **Utilizzo di BCrypt con forza predefinita 10 [9]**: si raccomanda ai professionisti di utilizzare come forza predefinita in BCrypt durante la codifica delle password almeno 16 (come da linee guida), per essere sempre protetti.

- **Richieste illimitate durante invocazione API [8, 9]**: si raccomanda ai professionisti di offrire sempre politiche di limitazione per, appunto, limitare il numero di richieste da parte degli utenti durante le invocazioni API.

CAPITOLO 4

Conclusioni

Questo lavoro di tesi riporta una revisione sistematica della letteratura che mira ad approfondire il concetto dei security smell, la loro individuazione, le infrastrutture, le metodologie o le tecniche di sviluppo dove essi sono maggiormente frequenti e come poterli, laddove possibile, mitigare al fine di non generare debolezze nella sicurezza di un sistema. I risultati di questa analisi sistematica lasciano emergere che, ad oggi, i security smell individuati in letteratura compaiono maggiormente in ambiti come IaC (Infrastructure as Code) e Mobile, rispetto a Spring o nelle Tecniche generali di programmazione per la generazione di codice. Inoltre, i security smell più frequenti nelle categorie analizzate, sono rispettivamente: Segreto Hard-Coded, Canale di Trasporto non sicuro, Uso di algoritmi di crittografia deboli e Uso di assert. Ciò che differenzia i documenti citati in questo articolo con il nostro studio è la generalità dei security smell analizzati: il nostro studio non approfondisce i security smell in un solo ambito, ma prende in considerazione tutti quelli trattati dalla letteratura esistente, mettendoli a confronto tra loro, analizzandone la frequenza con cui essi si presentano e come possono essere eventualmente risolti o mitigati.

4.1 Sviluppi Futuri

Il nostro futuro programma di ricerca è guidato dalle considerazioni e dalle limitazioni che hanno caratterizzato questa revisione sistematica della letteratura. Infatti, questo studio si basa su informazioni già presenti in letteratura, le quali sono state analizzate approfonditamente. Un possibile sviluppo futuro dipende fortemente dall'interesse che i security smell susciteranno nella letteratura; ad oggi, infatti, non vi sono numerosi articoli che trattano i security smell.
Un altro possibile sviluppo futuro potrebbe mirare all'implementazione di un tool che riesca ad identificare i security smell in uno o più ambiti analizzati in questo documento. Infine, per poter migliorare ulteriormente i risultati, i futuri lavori dovranno incrementare la quantità di dati analizzati al fine di poter riconoscere nuovi security smell, mitigarli, e individuare quali sono gli ambiti dove essi sono maggiormente frequenti.

Bibliografia

[1] 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*, ser. ICSE '19. IEEE Press, 2019, p. 164–175. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00033 (Citato alle pagine ii, 2, 4, 5, 11, 13, 17, 18, 22, 23, 29 e 30)

[2] A. Rahman and L. Williams, "Different kind of smells: Security smells in infrastructure as code scripts," *IEEE Security Privacy*, vol. 19, no. 3, pp. 33–41, 2021. (Citato alle pagine ii, 2, 11, 13, 17, 18, 22 e 23)

[3] N. Saavedra and J. a. F. Ferreira, "Glitch: Automated polyglot security smell detection in infrastructure as code," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556945 (Citato alle pagine ii, 11, 13, 17, 18, 22 e 23)

[4] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, "Security smells in ansible and chef scripts: A replication study," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 1, jan 2021. [Online]. Available: https://doi.org/10.1145/3408897 (Citato alle pagine ii, 11, 13, 17, 18, 22, 23, 29 e 30)

[5] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. S. Santos, "An empirical study of code smells in transformer-based code generation techniques," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2022, pp. 71–82. (Citato alle pagine ii, 11, 14, 18, 19, 22, 26, 30 e 31)

[6] 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)*, 2017, pp. 121–130. (Citato alle pagine ii, 11, 15, 19, 22, 24 e 31)

[7] P. Gadient, M.-A. Tarnutzer, O. Nierstrasz, and M. Ghafari, "Security smells pervade mobile app servers," in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3475716.3475780 (Citato alle pagine ii, 11, 15, 19, 22, 24 e 31)

[8] C. Cheh and B. Chen, "Analyzing openapi specifications for security design issues," in *2021 IEEE Secure Development Conference (SecDev)*, 2021, pp. 15–22. (Citato alle pagine ii, 12, 15, 22, 24 e 32)

[9] M. Islam, S. Rahaman, N. Meng, B. Hassanshahi, P. Krishnan, and D. D. Yao, "Coding practices and recommendations of spring security for enterprise applications," in *2020 IEEE Secure Development (SecDev)*, 2020, pp. 49–57. (Citato alle pagine ii, 12, 15, 22, 25, 30, 31 e 32)

Ringraziamenti

Per concludere è doveroso ringraziare coloro che hanno contribuito alla realizzazione della mia tesi di laurea e che mi hanno sostenuto durante questi anni universitari.

Innanzitutto, vorrei ringraziare il mio relatore, il prof. Fabio Palomba, il quale mi ha guidato sin dalla scelta dell'argomento di questo lavoro di ricerca.

Ringrazio inoltre i tutor che mi hanno seguito durante la stesura di questo lavoro di tesi, la Dott.ssa Giulia Sellitto ed il Dott. Carmine Ferrara, per avermi fornito preziosi consigli e supporto costante.

Un ringraziamento speciale va ai miei genitori, a mio fratello e ad Alessia, per aver sempre creduto in me. Siete stati, in questo percorso e da sempre, un punto di riferimento e di forza. Grazie.

Grazie alla mia famiglia, che mi ha supportato durante questi anni di studio.

Infine, ringrazio i miei colleghi Daniele, Severino, Gennaro, Sabatino, Marco, Valentina, Martina e Pasquale per avermi supportato e aiutato durante questo percorso universitario.