UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Identificazione di Vulnerabilità SQL Injection tramite Taint Analysis: Analisi della Letteratura e Confronto Empirico

RELATORE

Prof. Andrea De Lucia

Dott. Emanuele Iannone

Università degli Studi di Salerno

CANDIDATO

Angelo Santangelo

Matricola: 0512112615

Anno Accademico 2022-2023

Questa tesi è stata realizzata nel

Abstract

Il web è diventato un elemento essenziale nella nostra vita quotidiana ma purtroppo è soggetto, spesso, ad attacchi di SQL Injection, la quale è tra le vulnerabilità più comuni e dannose che affliggono i siti online. In letteratura, sono presenti confronti empirici tra tool di detection di SQL Injection su un unico benchmark, tuttavia non è mai stato realizzato un confronto su molteplici benchmark. La tesi presenta un confronto empirico tra molteplici tool che rilevano SQL Injection facenti uso di Taint Analysis, tenendo conto dei principali linguaggi di programmazione usati per sviluppare app nell'ambito web e mobile. Tali tool verranno, poi, testati su molteplici benchmark presenti in letteratura. La ricerca dei tool e dei benchmark è stata condotta tramite il motore di ricerca GOOGLE SCHOLAR. Sono stati identificati dei criteri di inclusione/esclusione da applicare ai risultati al fine di ridurre il numero di articoli da consultare. Dopo l'applicazione di questi criteri, sono rimasti 19 articoli, rilevando 14 tool e 4 benchmark. Gli 8 tool disponibili online sono stati configurati e riprodotti per valutare la loro usabilità. Successivamente, i tool che si sono rivelati usabili, sono stati eseguiti sui benchmark risultanti per valutare le loro capacità di identificazione di vulnerabilità SQL Injection. I tool usati sono WAP, RIPS e SQL-SCAN. SQL-SCAN non è risultato eseguibile su nessun benchmark mentre WAP e RIPS sono risultati eseguibili sul benchmark "PHP-Vulnerability-Test-Suite". RIPS ha ottenuto le migliori prestazioni mostrando il 15% di Precision, 81% di Accuracy, 22% di Recall, 16% di F1-score e un picco massimo di 43.705 secondi di tempo impiegato per analizzare i 9,552 file php contenuti nel benchmark. In conclusione, sono da evidenziare alcuni limiti: i tool ne sono tanti ma solo la metà sono disponibili e usabili. Di conseguenza, è necessario effettuare molteplici studi che vadano ad analizzare il motivo per il quale tali tool falliscono.

Indice

Elenco delle figure

CAPITOLO 1

Introduzione

1.1 Contesto applicativo

Con lo sviluppo rapido della tecnologia, dei social network ed ecommerce, la dipendenza dalle applicazioni web e mobile sta incrementando in maniera esponenziale negli ultimi tempi perchè offrono la possibilità di svolgere una vasta gamma di attività quotidiane, tra cui shopping, lettura, operazioni bancarie e molto altro. La sicurezza e la privacy sono diventate una preoccupazione sempre maggiore per vari stakeholder, poichè gli utenti conservano, su queste applicazioni, alcuni dati sensibili, i quali, solitamente, sono memorizzati nella base di dati. Sfortunatamente, la presenza di vulnerabilità di sicurezza nelle applicazioni web e mobile consente agli utenti malintenzionati di sfruttarle per ottenere guadagni finanziari, dati sensibili (come password, pin di trasferimento bancario, ecc.) e, di conseguenza, portare l'applicazione in fallimento, ovvero non funzionare correttamente a causa di perdita di dati. La ricerca sulle tecniche di rilevamento delle vulnerabilità e di prevenzione degli attacchi è stata abbastanza intensa nell'ultimo decennio.
La OWASP (Open Web Application Security Project) è un'organizzazione che si occupa di sicurezza delle applicazioni web. [1] In particolare, ha inserito l'SQL Injection al

[1] https://owasp.org/

terzo posto nella sua lista delle dieci vulnerabilità di sicurezza più critiche nel mondo delle app sia web che mobile. [2] In particolare, uno studio del 2013 ha individuato vulnerabilità di SQL Injection in siti web di grandi aziende come eBay, Amazon, Google, Facebook, Twitter, LinkedIn, Adobe, e molte altre. Le vulnerabilità di SQL Injection sono dovute a una insufficiente validazione dell'input. Più precisamente, gli attacchi di iniezione SQL possono verificarsi quando un'applicazione web o mobile riceve un input dall'utente che contiene comandi SQL maliziosi, ad esempio da una form di login o direttamente dalla barra del browser contenente l'URL, e lo utilizza per costruire una query al database, eseguita, poi, senza un'adeguata convalida. L'impatto che si ha, in questi casi, è l'impossessamento, da parte del malintenzionato, delle informazioni sensibili degli utenti, come password o pin bancari.

Uno studio pubblicato nel 2020 [8] ha dimostrato che la *Taint Analysis* è una delle tecniche più efficaci per rilevare vulnerabilità di sicurezza nel software, affermando, inoltre, che ha superato molte altre tecniche esistenti in termini di precisione e di adattabilità, ovvero può essere utilizzata in una varietà di linguaggi di programmazione e framework a differenza di alcune che sono specifiche per determinati ambienti e linguaggi. Tale tecnica ci dà la possibilità di identificare come i dati sensibili, ad esempio una password, vengono propagati tra le varie componenti all'interno del sistema, tracciandone il flusso e identificando eventuali vulnerabilità di sicurezza. Esistono vari tool [9] che usano la tecnica della Taint Analysis, per ognuno dei quali ne sono state valutate le prestazioni sulla base di alcune metriche, come il tempo impiegato per il rilevamento, la precisione, la facilità d'uso, ecc. Sulla base di tali metriche, sono stati realizzati anche vari lavori di confronto tra tool con lo scopo di far comprendere agli utenti il contesto per il quale deve essere utilizzato un determinato tool piuttosto che un altro.

[2] https://www.zscaler.com/blogs/product-insights/what-owasp-top-10?

1.2 Motivazione ed obiettivi

La crescente diffusione delle applicazioni web e mobile ha reso la sicurezza un fattore cruciale per la protezione delle informazioni sensibili degli utenti. Al giorno d'oggi sono rarissimi gli studi che si concentrano sulla comparazione di tool di individuazione di SQL Injection facenti uso di Taint Analysis. Alla luce di quanto detto in precedenza, l'obiettivo della presente tesi è quello di analizzare e confrontare i principali tool di individuazione di SQL injection, tramite l'uso della Taint Analysis, e di sperimentarli su diversi benchmark. La ricerca di tali tool avverrà sul web poichè non si è in possesso di una lista di tool e benchmark noti. In questo modo, si vuole offrire un'analisi dettagliata delle loro prestazioni e si forniscono indicazioni utili per la scelta degli strumenti più adatti per la prevenzione degli attacchi di SQL Injection. In particolare, verranno usate le principali metriche di valutazione per valutare la loro efficacia e si concluderà identificando il tool che funziona meglio sulla base dei risultati ottenuti.

1.3 Struttura della tesi

Il secondo capitolo si concentra sullo studio della Taint Analysis e della SQL Injection. In particolare, il capitolo offre un background completo sui due concetti e uno stato dell'arte sull'utilizzo delle tecniche di Taint Analysis per prevenire e mitigare gli attacchi di SQL Injection, con un focus sulle innovazioni e le sfide ancora aperte. All'interno del terzo capitolo si discute degli strumenti e della metodologia utilizzata per la ricerca e la raccolta degli articoli/siti web al fine di arricchire il background e lo stato dell'arte per quanto riguarda la Taint Analysis e l'SQL Injection, cercando, al contempo, di trovare tool e benchmark da approfondire nello studio. Il quarto capitolo si focalizza sui risultati ottenuti attraverso il processo di literature review, rispondendo a quattro domande di ricerca relative alla disponibilità, eseguibilità e prestazioni dei tool. Il quinto capitolo, infine, riassume il lavoro svolto e si propongono diversi spunti di riflessione che potrebbero essere utilizzati in futuro per migliorare ed ampliare il lavoro.

CAPITOLO 2

Background

2.1 SQL Injection

Un attacco SQL Injection consiste nell'inserimento o "iniezione" di dati in input ad una query SQL dal client all'applicazione per alternarne il comportamento; facendo ciò l'attaccante potrebbe leggere, modificare, cancellare dati sensibili all'interno di un base di dati, il che porterebbe all'esecuzione di funzioni di amministrazione su di esso. L'SQL injection impatta su diversi aspetti legati alla sicurezza:

- **Riservatezza**: i database SQL generalmente contengono dati sensibili, come password o PIN della banca di riferimento e l'SQL Injection consente ad un'attaccante di leggere tali dati dal DB.
- **Autenticazione**: se vengono utilizzati comandi SQL per controllare nomi utente e password, potrebbe essere possibile connettersi a un sistema come un altro utente;
- **Autorizzazione**: se le informazioni di autorizzazione fossero conservate in un database SQL, ad esempio un sistema di Role Based Access Control, potrebbe essere possibile modificare queste informazioni.

- **Integrità**: proprio come potrebbe essere possibile leggere informazioni sensibili, è anche possibile apportare modifiche o addirittura eliminare queste informazioni con un attacco SQL Injection.

L'attacco viene realizzato inserendo un meta-carattere nell'input dei dati, che può essere, ad esempio, un modulo di login o una barra di ricerca, per poi inserire i comandi SQL come parametri della query: se l'applicazione web o l'app mobile non effettua una buona validazione dei dati, il database può eseguire la query dannosa senza alcuna restrizione; quindi è importante che gli sviluppatori conoscano escamotage per prevenire tali attacchi, come ad esempio l'uso di query parametrizzate oppure verificare che l'input non contenga del codice SQL che influisca sulla logica della query da eseguire.

Esempio

- query di partenza = **SELECT * FROM items WHERE owner = 'hacker' AND itemname =** <inputAttaccante>
- l'attaccante inserisce l'input: **name'; DELETE FROM items; – –**
- Di conseguenza, la query diventa:

```
SELECT *
FROM items
WHERE owner = 'hacker' AND itemname = 'name';
DELETE FROM items; - -
```

Molti DBMS consentono l'esecuzione in batch di più statement SQL separate da punto e virgola; in questo modo gli statement vengono inseriti in unico blocco di codice SQL e, quando la query viene eseguita, il motore di database eseguirà tutte gli statement SQL nell'ordine in cui sono stati elencati. Sebbene questa stringa di attacco provochi un errore in Oracle DB e in altri DBMS perchè non consentono questa esecuzione in batch, nei database che, invece, lo permettono, questo tipo di attacco consente all'attaccante di eseguire comandi arbitrari contro il database. Inoltre, i due trattini inseriti alla fine della query, stanno a significare che, nel caso in

cui sia presente altro codice dopo il WHERE, tale porzione verrà messa sottoforma di commento e, quindi, la query terminerà dopo il comando di DELETE, il quale pone il rischio che, se non viene realizzato un buon controllo preliminare sull'input, la tabella ITEMS del database usato dall'applicazione verrà definitivamente eliminata.

2.1.1 Cause dell'SQL Injection

Le maggiori **cause** dell'SQL Injection sono:

- **Input non validati**: questa è quasi sempre la causa più comune. Ci sono alcuni parametri di input nelle applicazioni web che vengono utilizzati nelle query SQL e se non vi è alcun controllo per questi parametri, possono essere utilizzati negli attacchi di SQL Injection. Possono contenere parole chiave SQL, ad esempio INSERT, UPDATE, DELETE, ecc.
- **Mancanza di autorizzazione adeguata**: se un'applicazione web non implementa un sistema di autorizzazione corretto, un attaccante potrebbe utilizzare l'SQL Injection per accedere, modificare o eliminare dati sensibili all'interno del database.
- **Controllo solo lato client**: se la convalida dell'input viene effettuata solo lato client, allora gli attaccanti possono bypassare tale convalida mediante la disattivazione del codice JavaScript nel browser dell'utente o mediante la modifica delle richieste HTTP. Inoltre, gli attaccanti, possono utilizzare tool automatizzati (e.g. bot) per inviare dati malevoli al server evitando il controllo lato client.
- **Mancanza di filtri di caratteri speciali**: se un'applicazione web non filtra correttamente i caratteri speciali o di escape nei dati di input, un attaccante potrebbe inserire caratteri che alterano la struttura della query SQL. Possono essere, ad esempio, le virgolette o i punti e virgola.

2.1.2 Forme di SQL Injection

- **Messaggi di errore**: i messaggi di errore che sono generati dal database di back-end possono essere restituiti al client e presentati nel browser web o

nell'app mobile. Se gli sviluppatori dell'applicazione non gestiscono bene gli errori, magari attraverso delle error page appositamente create, si aumentano i rischi per l'applicazione perchè gli attaccanti possono analizzare tali messaggi per riuscire a comprendere la struttura del database, al fine di costruire un loro attacco. Questo capita quando vengono visualizzati gli stessi errori che restituiscono i DBMS, ovvero messaggi che contengono nomi di tabelle o di colonne che possono essere sfruttate dall'attaccante per capire la struttura delle tabelle; oppure messaggi di errore di sintassi SQL, ad esempio, se l'errore indica: `Errore di sintassi vicino a ' OR 1=1`, l'attaccante potrebbe dedurre che l'input viene semplicemente concatenato nella query e potrebbe tentare di inserire codice SQL dannoso tra gli apici.

- **Dynamic SQL**: le query SQL vengono costruite dinamicamente da script o programmi in una stringa di query. Tipicamente, la parte dinamica degli statement SQL sono le clausole WHERE. Il problema è che le query possono ricevere anche parole chiave SQL e caratteri di controllo. Questo significa che gli attaccanti possono creare una query completamente diversa da quella prevista.

- **Supporto INTO OUTFILE**: molti RDBMS beneficiano della clausola INTO OUTFILE. In questo caso, un attaccante può manipolare le query SQL in modo che producano un file di testo contenente i loro risultati. Se gli attaccanti possono accedere, poi, a questo file, possono abusare delle stesse informazioni, ad esempio, superare l'autenticazione.

- **Istruzioni multiple**: se il database supporta la clausola UNION, l'attaccante ha molte chance perchè ci sono più metodi di attacco per l'SQL Injection. Ad esempio, ad una query (già fissata) di SELECT può essere aggiunto uno statement sql di INSERT, tramite UNION, causando l'esecuzione di due query diverse. Se questo viene eseguito in una form di accesso, l'attaccante potrebbe aggiungere se stesso alla tabella degli utenti.

- **Sub-selects (sottoquery)**: il supporto alle sottoquery è una debolezza per i RDBMS ad esempio, clausole SELECT aggiuntive possono essere inserite nelle clausole WHERE della query originale. Questa debolezza rende l'applicazione

web o l'app mobile più vulnerabile perchè l'attaccante potrebbe inserire degli statement che permettono di raccogliere informazioni sensibili.

2.1.3 Metodi di attacco di tipo SQL Injection

Ci sono diversi metodi di **attacco** che vengono eseguiti insieme o in sequenza a seconda dell'obiettivo dell'attaccante e sono:

Tautologie. Questo tipo di attacco inietta token SQL nella dichiarazione della query condizionale in modo che verrà valutata sempre come vera. Questo tipo di attacco viene utilizzato per aggirare il controllo di autenticazione e l'accesso ai dati sfruttando un campo di input vulnerabile che utilizza la clausola WHERE. Prendiamo, come esempio, la seguente query:

```
SELECT *
FROM users
WHERE id='' and password='';
```

Tale query stamperà tutte le informazioni dell'utente con un certo `id` e con una certa `password`. Supponiamo che l'attaccante modifichi la query in questo modo:

```
SELECT *
FROM users
WHERE id='112' and password='aaa' OR 1=1;
```

La tautologia, aggiunta dall'attaccante, è l'espressione **1=1** perchè è sempre vera e, di conseguenza, la query stamperà tutte le informazioni della tabella `users`, anche se l'attaccante non inserisce un valore corretto per i campi `id` e `password`, in quanto la tautologia rende comunque vera la clausola `WHERE`.

Query illegali/logicamente errate. Quando una query viene rifiutata dal database, viene restituito un messaggio di errore contenente informazioni utili per il debug. Infatti l'attaccante, intelligentemente, potrebbe aggiungere token SQL per produrre volontariamente errori di sintassi o errori logici. Questi messaggi d'errore danno informazioni utili all'attaccante per identificare i parametri vulnerabili del database dell'applicazione. Esempio:

1. URL originale:
   `http://www.zoomClickEventi.it/eventi/?id_nav=8864`
2. SQL Injection (aggiunta di un'apice alla fine):
   `http://www.zoomClickEventi.it/eventi/?id_nav=8864'`
3. Messaggio di errore mostrato:
   `SELECT name FROM users WHERE id =8864'`

Dal messaggio di errore capiamo il nome della tabella e i campi name e id. Di conseguenza, l'attaccante potrebbe organizzare attacchi più mirati.

Union Query. Gli attaccanti uniscono più query alla query originale. Esempio: supponiamo di avere la seguente query:

```
SELECT name, phone
FROM users
WHERE id=$id
```

Iniettando il valore id seguente:

```
1 UNION ALL SELECT creditCardNumber, 1
FROM CreditCardTable
```

Avremo la seguente query:

```
SELECT name, phone
FROM users
WHERE id=1 UNION ALL SELECT creditCardNumber, 1
FROM CreditCardTable
```

Unisce il risultato della query originale con tutti gli utenti delle carte di credito.

Piggy-backed Queries. Si sfrutta il delimitatore ';' per aggiungere una query extra alla query originale e il database le esegue distintamente. Nel seguente esempio, supponiamo che l'attaccante inietti: `0; drop table users` come valore dell'attributo *pin*, allora l'applicazione produrrà la seguente query:

```
SELECT info
FROM users
WHERE login='doe' AND pin=0; drop table users
```

Grazie al carattere ';' il database accetta ed esegue entrambe le query. La seconda query è illegittima e può eliminare la tabella *users* dal database. Da notare che alcuni DBMS non necessitano del delimitatore per separare query distinte quindi, per rilevare questo tipo di attacco, la scansione di un carattere speciale non è una soluzione sufficiente.

Stored Procedure. È una parte del database che il programmatore può creare come un livello di astrazione aggiuntivo al database. A seconda della specifica Stored Procedure nel database, esistono diversi modi per attaccare. Nell'esempio seguente, l'attaccante sfrutta una Stored Procedure parametrizzata.

```
CREATE PROCEDURE DBO.isAuthenticated
@userName varchar2, @pass varchar2, @pin int
AS
EXEC(``SELECT accounts FROM users
WHERE login='" +@userName+ ``' and pass='" +@password+ ``'
and pin=" +@pin);
GO
```

L'attaccante inserisce **'; SHUTDOWN; – –** per username o password. Quindi la Stored Procedure genera la seguente query:

```
SELECT accounts
FROM users
WHERE login='doe' AND pass='';
SHUTDOWN; -- -- AND pin=
```

Dopo di che, questo tipo di attacco funziona come attacco piggy-back. In particolare, la seconda query (illecita) causa lo spegnimento del database.

Blind Injection. Anche conosciuta come 'blind SQL injection' o 'time-based blind SQL injection', è una variante dell'SQL injection. Nel caso in cui ci sia un errore

in corrispondenza di qualche parametro di input, gli sviluppatori, a differenza del messaggio d'errore, creano e visualizzano a video una pagina generica che indica un errore altrettanto generico. In questo caso, l'attaccante non ha a disposizione possibili informazioni sulla struttura del database, come accadeva per il messaggio d'errore, e, di conseguenza, diventa più difficile eseguire un attacco. Nonostante ciò, l'attaccante potrebbe comunque eseguire una serie di domande booleane mediante statement SQL. Ad esempio, consideriamo la seguente query:

```
SELECT accounts
FROM users
WHERE login='' AND pass='' AND pin=0
```

Ora l'attaccante potrebbe iniettare una porzione di codice malevolo in questo modo:

```
SELECT accounts
FROM users
WHERE login='doe' AND 1=0 -- AND pass='' AND pin=0
```

Con l'aggiunta di `AND 1=0 --`, tutto quello che segue diventerà un commento, ma la query non andrà a buon fine e restituirà un errore perchè è presente, nella clausola WHERE, l'equazione 1=0, il che è sempre falsa. L'attaccante, quindi, siccome verrà visualizzata una pagina con un errore generico, non capisce se l'errore è dovuto alla validazione dell'input o a un errore logico nella query. L'attaccante, a questo punto, potrebbe lanciare una nuova query però inietta l'equazione 1=1, piuttosto che 1=0, in questo modo:

```
SELECT accounts
FROM users
WHERE login='doe' AND 1=1 -- AND pass='' AND pin=0
```

Se non ci sono messaggi di errore di login, l'attaccante scopre che il campo login è vulnerabile all'iniezione. Questa tecnica è chiamata *metodo di inferenza* e consiste, quindi, nell'effettuare una serie di interrogazioni booleane osservando le risposte e deducendone il significato.

Codifiche alternative. Gli attaccanti modificano la query utilizzando codifiche alternative, come ad esempio esadecimali, ASCII e Unicode. In questo modo possono

sfuggire ai filtri del programmatore che esamina query di input con "caratteri cattivi" noti. Ad esempio, l'attaccante può utilizzare "char(44)" invece dell'apostrofo che è un carattere cattivo noto. Esempio: supponiamo che nel campo *pin* viene iniettato la seguente stringa:

```
0; exec (0x73587574 64 5f77 6e)
```

Allora il risultato della query è:

```
SELECT accounts
FROM users
WHERE login=" AND
    pin=0; exec (char(0x73687574646f776e))
```

Questo esempio utilizza la funzione char () e la codifica ASCII esadecimale. La funzione char () prende la codifica esadecimale dei caratteri e restituisce il/i carattere/i effettivo/i. La sequenza di numeri nella seconda parte dell'iniezione è la codifica ASCII esadecimale della stringa di attacco, la quale, in questo caso, viene tradotta nel comando *shutdown* (arresto del database) quando viene eseguita.

2.1.4 Tecniche di difesa dall'SQL Injection

Le principali tecniche di difesa dall'SQL Injection sono:

2.1.4.1 Prepared Statement

La prepared statement permette di costruire una query pre-compilata e può essere riutilizzata più volte con parametri diversi. Il vantaggio è che i parametri vengono trattati come dati separati dalla query stessa in modo che non possano essere interpretati come codice SQL e non possano quindi influenzare la struttura della query. Ciò rende molto difficile per un attaccante eseguire un attacco di SQL Injection, poiché l'input fornito dall'utente viene sempre considerato come un valore, e non come parte integrante della query stessa. Inoltre le Prepared Statement migliorano anche le prestazioni dell'applicazione, poiché il server del database compila la query una sola volta e la memorizza in memoria; ciò significa che quando la query dovrà essere eseguita una seconda volta, anche potenzialmente con parametri diversi, il

server può riutilizzare la query già compilata, evitando di doverla ricompilare ogni volta. Facciamo un esempio:

```
import java.sql.*;

public class PreparedStatementExample {
    public static void main(String[] args) {

        String query = "SELECT * FROM users WHERE age > ?";

        //Ci facciamo restituire l'oggetto Connection dal DriverManager
        try (Connection conn = DriverManager.getConnection());
             PreparedStatement stmt = conn.prepareStatement(query)) {

            int minAge = 18;
            stmt.setInt(1, minAge);

            ResultSet rs = stmt.executeQuery();

            while (rs.next()) {
                String name = rs.getString("name");
                int age = rs.getInt("age");
                System.out.println("Name: " + name + ", Age:" + + age);
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

In questo esempio, stiamo eseguendo una query parametrizzata utilizzando PreparedStatement per selezionare tutte le righe dalla tabella “users" in cui l’età è maggiore di 18. Abbiamo creato un oggetto PreparedStatement utilizzando il metodo prepareStatement della connessione al database. La query contiene un segnaposto ’?’ che rappresenta il parametro da sostituire. Successivamente, abbiamo impostato il valore del parametro utilizzando il metodo `setInt(1, minAge)`, dove `1` indica il primo segnaposto nella query e `minAge` è il valore che vogliamo concatenare al segno > nella query, ossia la prima occorrenza del carattere `?`. Tale metodo, in particolare, fa in modo che se l’attaccante inietti parametri che non siano interi, allora si impedisce l’iniezione bloccando la query; di conseguenza, si impedisce un possibile inserimento di codice malevolo. Inoltre, analogamente a come spiegato in precedenza, il motore del database va a controllare semplicemente se l’età di ogni utente è maggiore di 18 e non considera 18 come parte integrante della query bensì solo come stringa per i confronti. Infine, abbiamo eseguito la query utilizzando il metodo `executeQuery()` dell’oggetto PreparedStatement. Abbiamo iterato i risultati ottenuti dal ResultSet e stampato il nome e l’età corrispondenti.

2.1.4.2 Convalida dell'input della lista consentita

Questa tecnica prevede di definire una lista di valori consentiti per un determinato input, e di verificare che il valore fornito dall'utente sia presente in questa lista. Per ogni campo di input, l'applicazione definisce una lista di valori accettabile: se tale valore non è presente nella lista, verrà restituito un errore e non esegue l'operazione richiesta. E' un metodo efficace per prevenire gli attacchi poiché limita l'input all'utente solo a determinati valori, tuttavia, non è sicura al 100% perché l'elenco dei valori consentiti potrebbe non essere definito correttamente e quindi un attacco potrebbe avere comunque successo. Inoltre questa tecnica non è adatta per i campi di input che richiedono una vasta gamma di valori, poiché sarebbe quasi impossibile definire un elenco esaustivo di valori accettati. La convalida dell'input consente anche l'uso di espressioni regolari, le quali servono per verificare il formato di un input, ma bisogna prestare attenzione ad attacchi RegEx Denial of Service (ReDoS) perché questi attacchi fanno sì che un programma che utilizza un'espressione regolare mal progettata, funzioni molto lentamente e utilizzi le risorse della CPU per molto tempo. Facciamo un esempio:

```
import java.sql.*;

public class Example {
    public static void main(String[] args) {
        //Lista di valori consentita
        List<String> valoriConsentiti = Arrays.asList("valore1",
        "valore2", "valore3");

        // Otteniamo l'input dall'utente
        Scanner scanner = new Scanner(System.in);
        System.out.print("Inserisci un valore: ");
        String input = scanner.nextLine();

        // Convalida dell'input
        if (valoriConsentiti.contains(input)) {
            System.out.println("Input valido!");
        } else {
            System.out.println("Input non valido. Riprova!");
            return;
        }

        //il resto è tutto uguale all'esecuzione della query dell'esempio
        precedente
    }
```

In questo esempio, abbiamo una lista di valori consentiti rappresentata come un oggetto List. L'utente viene invitato a inserire un valore tramite la classe Scanner.

Successivamente, verifichiamo se l'input dell'utente è presente nella lista dei valori consentiti utilizzando il metodo `contains()` della lista. Se l'input è presente nella lista, viene stampato "Input valido!", altrimenti viene stampato "Input non valido. Riprova!". Ovviamente si può personalizzare la lista dei valori consentiti con i valori che si desiderano convalidare rispetto all'input dell'utente.

2.1.4.3 Sanitizzazione dell'input

Nonostante sia una tecnica non usabile singolarmente ma da affiancare ad altre tecniche, risulta essere comunque una tecnica abbastanza efficace per prevenire possibili attacchi SQL Injection. Siccome l'uso di caratteri speciali, come "–" (commento in SQL) o "UNION", è spesso sfruttato dagli attaccanti per manipolare le query SQL e ottenere un accesso non autorizzato ai dati, la sanitizzazione dell'input implica la rimozione o l'escape di tali caratteri pericolosi prima di utilizzarlo nelle query SQL. Facciamo un esempio: supponiamo di avere un modulo di accesso che richieda di inserire all'utente un nome utente e una password per eseguire l'autenticazione al sito. Prima di passare questi due valori alla query SQL, sanifichiamo in questo modo:

```
<?php
// Ottenere l'input dell'utente
$username = $_POST['username'];
$password = $_POST['password'];

// Sanitizzazione dell'input
$username = mysqli_real_escape_string($conn, $username); // Utilizzando
la funzione mysqli_real_escape_string per evitare l'iniezione di SQL
$password = mysqli_real_escape_string($conn, $password);

// Creazione della query SQL parametrizzata
$query = "SELECT * FROM users WHERE username = '$username' AND password =
'$password'";

// Esecuzione della query SQL...
?>
```

Nell'esempio, la funzione `mysqli_real_escape_string` viene usata per eseguire l'escape dei caratteri speciali presenti nell'input, evitando così che possano alterare la sintassi della query SQL. In questo modo, anche se l'utente inserisse caratteri come "–" o "UNION", verranno trattati come dati letterali e non come comandi SQL. Tuttavia, è sempre consigliabile utilizzare prepared statements, ad esempio, per separare in modo sicuro i dati dalle istruzioni SQL.

2.1.4.4 Minimo privilegio

Tale tecnica di difesa mira a limitare l'accesso degli utenti solo alle risorse di cui hanno effettivamente bisogno per svolgere il loro lavoro e quindi mira nell'assegnare privilegi minimi piuttosto che concedere accesso illimitato o privilegi più elevati ad un utente. Ad esempio, se un utente ha solo bisogno di leggere i dati di un database, non dovrebbe avere accesso a privilegi di modifica e cancellazione. Ad esempio, nella colonna che riguarda il tipo di utente (e.g. amministratore, cliente, gestore, ecc.) bisogna fare in modo che ogni utente abbia un solo ruolo e che abbia accesso solo agli oggetti dell'applicazione di cui necessita; ovviamente si deve usare anche una buona pratica di programmazione per effettuare tutti i controlli ad hoc.

2.2 Taint Analysis

La Taint Analysis è una tecnica che consiste nel "contaminare" certi dati in input, ad esempio una password, per tracciare come essi vengono propagati tra le varie componenti all'interno del sistema. La contaminazione dei dati può avvenire in molti modi, ad esempio, attraverso l'inserimento di codice malevolo (e.g. parti di query SQL) all'interno di una componente di input (e.g. un modulo di inserimento dati) inviata a un'applicazione web o ad un'app mobile nella speranza di scatenare un'iniezione. In particolare, le variabili che provengono dalle componenti di input dall'utente vengono identificate come "contaminate" (o "tainted") in quanto sono soggette a contenere del codice malevolo. Generalmente la Taint Analysis consiste in tre componenti importanti:

- Source (in italiano "fonti"): sono tutti gli strumenti che possono essere utilizzate dagli attaccanti per inserire codice malevolo pericoloso, ad esempio in un'applicazione web-based, le fonti sono i parametri della richiesta HTTP, le sessioni, i form, ecc.

- Sink: sono le funzioni finali che eseguono azioni delicate agli attacchi, ad esempio in un'applicazione web-based alcuni dei sink sono le funzioni di query SQL, le system call, ecc. In pratica, determinano se è presente una possibile vulnerabilità nel programma.

- Sanitizer (in italiano "sanificatori"): sono tutte le funzioni che possono proteggere o filtrare i sources forniti, ad esempio in un'applicazione web-based, alcuni dei sanificatori, sono le funzioni di escape HTML (rimuovere caratteri speciali e.g. '<' e '>'), le funzioni di escape della query SQL (rimuovere caratteri speciali e.g. ' e "), e molti altri.

In generale si tracciano le variabili sospette e se esse vengono eseguite da un sink prima della sanificazione, viene segnalata una vulnerabilità. Solitamente, la Taint Analysis viene impiegata prima del rilascio del software per correggere ed identificare eventuali vulnerabilità ma può anche essere usata proprio durante lo sviluppo del software. Di seguito riassumiamo gli step generali:

- Sorgente di input: l'input fornito all'applicazione viene considerato come la sorgente dell'analisi. L'input può essere fornito tramite moduli di accesso o tramite linea di comando o anche tramite parametri URL.
- Propagazione dell'informazione: in questa fase, si analizza la sorgente in maniera tale che si tracci, partendo da esso, il flusso di dati fino alle variabili e alle funzioni che modificano o utilizzano tali dati forniti all'applicazione.
- Identificazione delle vulnerabilità: in questa fase, l'analisi identifica le vulnerabilità nell'applicazione, come ad esempio quella di SQL Injection.
- Output: l'output dell'analisi è un report che identifica le falle di sicurezza nell'applicazione e fornisce informazioni sui dati che sono stati trasmessi attraverso l'applicazione.

Possiamo spiegarlo ancora meglio con un semplice esempio minimale in PHP: supponiamo di avere un'applicazione web che utilizza un input fornito dall'utente (supponiamo da un campo di testo) per effettuare una stampa sul terminale di output:

```php
<?php
$name = $_GET['name'];
echo "Hello, ". $name;
?>
```

Questo è un esempio di codice che può causare delle vulnerabilità perché la variabile "name" è assegnata ad un input dell'utente, quindi verrà marcata come "tainted". Tale variabile verrà poi eseguita dalla funzione "echo", la quale è una delle funzioni sink: in questo caso si può verificare una potenziale vulnerabilità alla sicurezza in quanto un dato non attendibile (o non controllato) viene incorporato nel flusso di esecuzione di un'applicazione. L'attaccante, ad esempio, potrebbe inserire: `'); phpInfo(); //`, allora il codice sarà eseguito insieme alla visualizzazione del messaggio "Hello,". La funzione `phpInfo()` restituisce informazioni sulle impostazioni e la configurazione del server PHP in cui viene eseguito il codice, infatti è consigliato usarla solo in fase di debug dagli sviluppatori per risolvere problemi legati alla configurazione del server, evitando di esporre informazioni generiche su di esso ed evitando, anche, di consentire all'attaccante di eseguire comandi arbitrari sul server. Alcuni impatti di una variabile tainted in una sink sono: compromissione dei dati, ovvero quando una variabile tainted può portare alla divulgazione non autorizzata di informazioni riservate; impatto sulla reputazione, ovvero la divulgazione di dati sensibili possono minare la fiducia degli utenti nell'applicazione o nell'organizzazione; responsabilità legale, ovvero quando gli sviluppatori sono ritenuti responsabili per i danni causati e ciò può includere azioni legali, come multe o altre conseguenze. In generale, è di fondamentale importanza assicurarsi che le variabili tainted vengano adeguatamente sanificate, prima di raggiungere una sink, attraverso un sanitizer. In questo caso, allora, il dato sarà considerato "safe" (sicuro) dal pericolo dell'injection:

```php
<?php
$name = $_GET['name'];
$name = htmlentities($name);
echo "Hello, ". $name;
?>
```

In questo caso, la funzione `htmlentities()` viene utilizzata per convertire eventuali caratteri speciali nella stringa di input in equivalenti HTML-safe, ovvero si mettono al sicuro i dati tramite l'utilizzo di funzioni o metodi che eseguono l'en-

coding dei caratteri speciali facendo in modo che essi vengano trattati come testo normale e non come codice HTML, evitando così la vulnerabilità. Ad esempio, il carattere '<' viene convertito in '`<`', ecc. Facciamo un esempio: supponiamo che l'utente inserisca come input:

```
<script>alert('Hello, world!'); </script>
```

Questo codice verrà inserito all'interno dei tag '`<script>`', che sono utilizzati per l'esecuzione di codice JavaScript nelle pagine HTML. Esso verrà eseguito nel browser degli utenti e quindi diventa pericoloso se questa stringa non viene prima sanitizzata opportunamente in quanto l'obiettivo di un attaccante potrebbe essere quello di visualizzare un popup con un messaggio `Hello, world!` per scopi dimostrativi, ma potrebbe anche essere un codice più pericoloso che può compromettere la sicurezza dei dati dell'utente o del sistema. Otterremo in output, dopo l'applicazione di `htmlentities()`:

```
&lt;script&gt;alert(&#39;Hello, world!&#39;);&lt;/script&gt;
```

Si può dire che un programma ha una potenziale vulnerabilità se c'è una variabile che è "tainted", cioè non sanificata e poi eseguita da una sink. Per rendere questo discorso più comprensibile, fare riferimento alla Figura 2.1, la quale è stata adattata da [10].

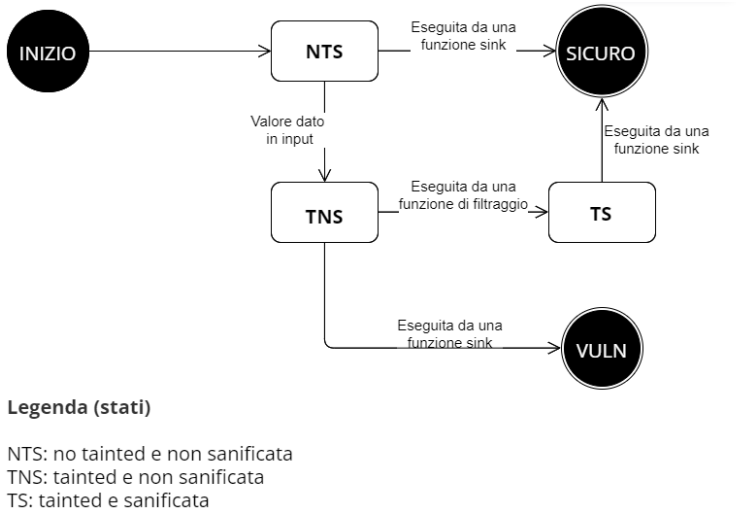

Figura 2.1: Diagramma di stato che mostra il comportamento della Taint Analysis con una determinata variabile.

La figura mostra che inizialmente una variabile risulta essere non pericolosa e non sanificata. Se essa viene eseguita da una funzione di sink, sarà sicura da qualsiasi pericolo. Al contrario, se viene assegnata ad un input dell'utente, sarà pericolosa ("tainted") e non sanificata. Se essa viene eseguita da una funzione di filtraggio, ovvero attraverso un sanitizer, allora sarà sanificata e, anche se venisse eseguita da una funzione di sink, sarà salva dal pericolo. Infine, se non venisse eseguita alcuna funzione di filtraggio bensì da una funzione di sink, allora la Taint Analysis avvisa che è presente una vulnerabilità.

La Taint Analysis può essere eseguita in due modi diversi: staticamente o dinamicamente e, inoltre, può avvenire con due modalità: Forward e Backward Analysis [11].

2.2.1 Static Taint Analysis

La Static Taint Analysis viene eseguita su codice sorgente o bytecode senza eseguire il programma. In pratica, analizza il flusso di dati e di controllo del programma e cerca di identificare come i dati fluiscono attraverso il programma in esame. Un esempio potrebbe essere quando ci sono problemi di sicurezza legati all'ambiente di esecuzione, come la configurazione del sistema operativo o delle librerie utilizzate oppure quando l'applicazione non gestisce correttamente l'input a runtime; in questi casi, la Static Taint Analysis non sarà in grado di rilevare tutte le possibili vulnerabilità che potrebbero manifestarsi durante l'esecuzione a runtime. Supponiamo di avere una password che viene trasmessa ad una funzione che effettua una query SQL: in questo caso la Static Taint Analysis, scansionando il codice, rileverà una potenziale SQL Injection se non viene effettuato prima un controllo preliminare sulla password. Al contrario, se è presente il controllo preliminare, ci assicurerà che la password non contenga codice malevolo magari perchè è stato passato ad un sanitizer e, di conseguenza, ci garantirà la sanitizzazione del dato stesso. Il flusso di controllo si riferisce al modo in cui il programma segue le istruzioni durante l'esecuzione del codice. La tecnica usata per analizzare il flusso di controllo in un programma è l'uso del Control Flow Graph (CFG). Un CFG rappresenta graficamente il flusso di

esecuzione del codice attraverso una serie di nodi e archi, dove i nodi rappresentano le istruzioni del programma e gli archi rappresentano il flusso di controllo tra le istruzioni. Ad esempio un'istruzione condizionale "if" creerà due archi uscenti dal nodo che rappresenta l'istruzione stessa, uno che rappresenta l'esecuzione del blocco di codice all'interno all'interno del "if" se la condizione è vera, e un altro che rappresenta l'esecuzione del blocco di codice all'interno dell'istruzione "else" se la condizione è falsa. La rappresentazione del flusso di controllo tramite CFG facilita l'identificazione di eventuali problemi nel flusso di controllo del programma, come loop infiniti, punti morti ("deadlocks") o la presenza di codice non raggiungibile. Esempio: supponiamo di avere il seguente codice:

```
1. x = Source(i);
2. y = x;
3. if(y == 0) {
4.     z = 0;
5. }
6. else {
7.     z = 1;
8. }
9. Sink(z);
```

Listing 1: Codice sorgente di esempio.

Il grafo del flusso di controllo è il seguente:

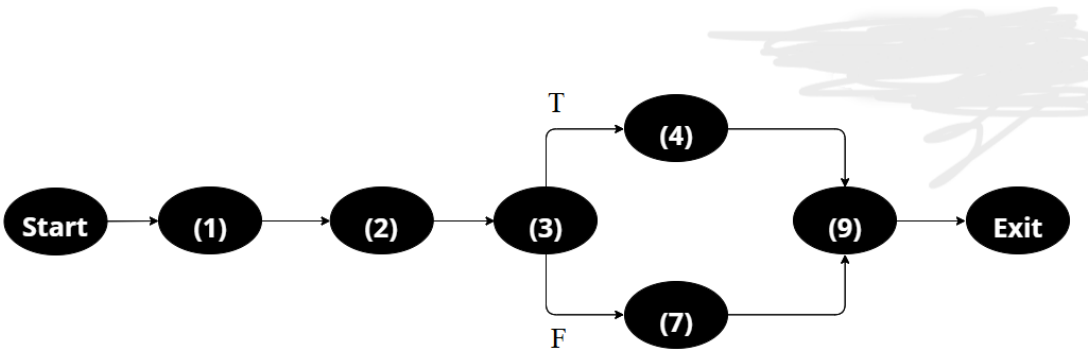

Figura 2.2: CFG generato a partire dal codice sorgente mostrato in Listing 1.

Dopodichè, la Static Taint Analysis, visitando il CFG nodo per nodo, assegna un'etichetta di "taint" solo a determinate variabili, ad esempio quelle che provengono in input da fonti esterne (e.g. moduli di accesso, terminale, parametri URL, ecc.), tramite una sorta di diagramma di stato, il quale inizialmente assegnerà a tutte le variabili etichette 'NT' (not tainted) in quanto non sono contaminati da valori non sicuri. Tale diagramma avrà, inizialmente, la seguente struttura:

Statement	X	Y	Z
1	NT	NT	NT
2	NT	NT	NT
3	NT	NT	NT
4	NT	NT	NT
7	NT	NT	NT
9	NT	NT	NT

Figura 2.3: Diagramma di stato iniziale.

Dopo aver analizzato il CFG, il diagramma risultante è il seguente:

Statement	X	Y	Z
1	T	NT	NT
2	T	T	NT
3	T	T	NT
4	T	T	NT
7	T	T	NT
9	T	T	T

Figura 2.4: Diagramma di stato finale.

N.B. Il simbolo 'T' sta per "tainted".
In questo caso la variabile 'x' verrà marcata nello statement (1) come "tainted" in quanto assumerà un valore dato in input da una fonte esterna. Analogamente, avviene per 'y' in quanto assumerà lo stesso valore di 'x', nello statement (2). Inoltre, la variabile 'z', anche se non sembrerebbe essere "tainted", lo sarà nello statement (9) quando verrà passata ad una funzione di sink perchè il suo valore dipende da una variabile "tainted", ovvero la 'y' e, inoltre, non è stata sanitizzata da nessun sanitizer. La Static Taint Analysis, quindi, ha il vantaggio di essere molto rapida e di non richiedere l'esecuzione del programma, ma il suo svantaggio è che non può rilevare alcune vulnerabilità che si manifestano solo durante l'esecuzione, e quindi

in fase di runtime. Inoltre, può essere eseguita anche durante la fase di sviluppo o di compilazione del software, può essere integrata nel processo di build o di controllo di qualità del software senza richiedere esecuzioni aggiuntive.

2.2.2 Dynamic Taint Analysis

La Dynamic Taint Analysis, invece, viene eseguita mentre il programma è in esecuzione (fase di runtime). In questo caso, la Taint Analysis analizza i dati che fluiscono attraverso il programma durante l'esecuzione, tracciando l'origine e il flusso dei dati attraverso il sistema. Quando si lavora con applicazioni che compiono computazioni onerose, si presta meglio la STA perchè la DTA richiede l'esecuzione del codice in un ambiente controllato, il che può rallentare il tempo di analisi e richiedere risorse aggiuntive. Inoltre, la DTA è particolarmente utile per individuare vulnerabilità legate ai flussi di dati sensibili; pertanto, quando si tratta di applicazioni con bassa presenza di dati sensibili, come le credenziali dell'utente, la DTA potrebbe essere inefficiente o addirittura inutile. Prendiamo lo stesso esempio che abbiamo usato per la Static Taint Analysis: la riga (2) si tratta di un flusso di informazioni esplicito, ovvero, esplicitamente, viene effettuata l'assegnazione di una variabile contaminata ad un'altra, mentre le righe (4) e (7) si trattano di flussi di informazioni impliciti, il che significa che il valore di taint dipende dalla decisione dello statement (3). Ricapitolando, quindi, il flusso di informazioni esplicito è legato alle dipendenze dei dati mentre il flusso di informazioni implicito è tipicamente dovuto alle dipendenze di controllo nel codice e, se quest'ultimo non viene eseguito, diventerà complicato da trovare alcuni flussi impliciti. La DTA, in questi casi, risulta essere più performante della STA. Mentre la Static Taint Analysis esamina il codice sorgente nel suo complesso e può rilevare potenziali problemi nel codice, anche se non vengono mai raggiunti durante l'esecuzione, la Dynamic Taint Analysis, invece, si concentra solo sulle parti del codice che vengono effettivamente eseguite durante un'istanza specifica di esecuzione del programma. Tuttavia, la STA rileva staticamente le vulnerabilità, il che potrebbe generare falsi positivi o falsi negativi con una probabilità più elevata, poiché non conosce il contesto esatto dell'esecuzione. La DTA, invece, ha una visione più precisa del flusso dei dati e, quindi, potrebbe ridurre il numero di falsi positivi ma anche

non rilevare tutti i potenziali problemi in tutte le situazioni. In conclusione, mentre la STA flagga una vulnerabilità in modo improprio (o sovrastimato), la DTA risulta essere più precisa ed avere meno falsi positivi ma con un tempo di esecuzione più alto. Tuttavia, è importante notare che entrambi gli approcci hanno vantaggi e limitazioni e possono essere utilizzati in modo complementare per garantire una migliore copertura dell'analisi delle informazioni sensibili dell'applicazione. La scelta dipende dalle esigenze specifiche dell'applicazione e dal contesto in cui viene utilizzata.

2.2.3 Backward Taint Analysis

La Backward Taint Analysis (analisi "all'indietro") parte dai punti di uscita del programma, ovvero identifica tutte le variabili e le funzioni che influenzano una determinata uscita, e risale al punto in cui i dati contaminati sono entrati nel programma. In questo modo, la Backward Taint Analysis può individuare tutti i punti in cui i dati contaminati sono stati inseriti nel programma e tutti i percorsi che quei dati hanno seguito all'interno del programma. Mostriamo un esempio artificiale: supponiamo che l'applicazione riceve un input dall'utente per effettuare una query sul database utilizzando la classe "executeQuery()" di JDBC; in questo caso, l'input dell'utente rappresenta una fonte di dati potenzialmente pericolosa che può influenzare il risultato dell'uscita del programma.

```
import java.sql.*;

public static void main(String[] args){
    try{
        //dopo aver fatto la connessione al DB

        String userInput = args[0];

        Statement stmt = conn.createStatement();
        String query = "SELECT * FROM mytable " +
                "WHERE id = " + userInput;
        ResultSet rs = stmt.executeQuery(query);

        while(rs.next()){ }

    }catch(SQLException e){}
}
```

Per applicare la Backward Taint Analysis a questo codice, si potrebbe utilizzare un framework di analisi statica o dinamica che supporta la Backward Taint Analysis,

come ad esempio FLOWDROID [11], e definire le regole di propagazione del taint a seconda delle operazioni eseguite dal programma, ovvero:

- Inizialmente l'output del programma (ottenuto tramite la variabile "rs") viene contrassegnato come "taint";
- Quando il risultato della query SQL viene assegnato alla variabile "result", il taint si propaga alla variabile stessa;
- Se il risultato della query viene utilizzato per elaborare i dati in qualche modo, il taint si propaga anche ai dati elaborati;
- Quando la query SQL viene costruita concatenando l'input dell'utente con la stringa fissa, il taint si propaga alla stringa "query";
- Infine, quando l'input dell'utente viene ottenuto tramite la variabile "userInput", il taint si propaga a questa variabile.

2.2.4 Forward Taint Analysis

La Forward Taint Analysis (analisi "in avanti") analizza il flusso di dati dal punto di ingresso del programma (solitamente si tratta della funzione principale "main()") fino alla sua uscita, ovvero, a partire da una fonte di dati contaminati (taint source), determina tutte le variabili e le funzioni che vengono influenzate da quella fonte. In questo modo, la Forward Taint Analysis può identificare tutte le operazioni che potrebbero essere eseguite sui dati contaminati e tutti i punti di uscita del programma in cui i dati contaminati potrebbero essere divulgati. Per mostrare un esempio, potremmo usare lo stesso esempio di prima cambiando l'ordine delle regole di propagazione del taint, ovvero:

- Il taint viene inizialmente propagato dal parametro "args[0]" che rappresenta l'input utente per la query;
- Quando il valore tainted viene concatenato alla stringa fissa, il taint viene propagato alla variabile "query";

- Quando la funzione “executeQuery()" viene chiamata passando la query tainted, il taint viene propagato al risultato della query, ovvero la variabile “rs”;
- Quando i dati tainted vengono processati tramite la funzione “rs.next()”, in questo caso, il taint non viene propagato a nessuna variabile successiva in quanto non c’è alcuna elaborazione di dati;

In pratica, quindi, la differenza sostanziale tra le due modalità è l’ordine di come avviene il flusso di dati. Presentiamo alcuni vantaggi e svantaggi di entrambe le modalità. La Backward Analysis ha una precisione più alta rispetto alla Forward perchè siccome quest’ultima segue il flusso di dati in avanti, a volte può perdere informazioni più importanti sulle origini dei dati. Di conseguenza, genera un numero minore di falsi positivi rispetto alla Forward. In generale, la Backward analysis richiede un maggiore overhead computazionale rispetto alla Forward perchè segue il flusso delle informazioni sensibile all’indietro nel codice, partendo dal punto in cui vengono utilizzate risalendo verso l’origine. Questo processo può richiedere l’analisi di più percorsi nel grafo di controllo del flusso del programma, il che può comportare un aumento dell’overhead computazionale. Al contrario, la Forward Analysis, invece, segue il flusso delle informazioni sensibili in avanti nel codice, partendo dall’origine fino al punto di utilizzo. Poichè il flusso di informazioni transita nella direzione naturale dell’esecuzione del programma, la Forward può richiedere meno risorse computazionali. Tuttavia, è importante sottolineare che l’overhead può variare a seconda delle caratteristiche del programma analizzato, quindi, è consigliabile considerare anche il contesto specifico e le prestazioni del sistema per valutare l’effettivo overhead computazionale. Alcuni fattori sono: il numero di istruzioni o percorsi da analizzare nell’analisi stessa. Infine, la Backward Analysis può avere una minore copertura rispetto alla Forward a causa dei percorsi di esecuzione limitati. Ad esempio, l’eventuale presenza di strutture di controllo condizionale, come i branch e i cicli, portano a molti possibili percorsi di esecuzione nel flusso di controllo nel programma. Di conseguenza, non solo richiederebbe notevoli risorse computazionali per analizzare tutti questi percorsi, ma potrebbe non essere in grado di coprire completamente tutti i percorsi del flusso di esecuzione.

2.3 Lavori Esistenti di Confronto tra Tool

In questa sezione esamineremo una serie di studi e ricerche che hanno affrontato la comparazione tra tool di rilevamento di SQL Injection e tool di Taint Analysis in contesti specifici. Questi studi hanno cercato di valutare l'efficacia di diverse soluzioni per identificare e mitigare le vulnerabilità di SQL Injection nelle applicazioni web e mobile. Altri studi, invece, si sono concentrati sull'efficacia e l'efficienza dei tool di Taint Analysis nell'identificare e tracciare il flusso dei dati sensibili all'intero delle applicazioni web e mobile. Due dei lavori effettuati che riguardano comparazione tra tool di Taint Analysis sono, ad esempio, [9] e [12]. Il primo effettua un confronto tra i tre tool di Taint Analysis più prominenti, ovvero FLOWDROID, AMANDROID e DROIDSAFE e vengono valutati sia su una suite famosa di benchmarking, come quella di DROIDBENCH e ICC-BENCH, e sia su reali applicazioni prese da Google Play Store. Il secondo, analogamente, effettua un confronto tra tre tool che rilevano flussi di informazioni nelle mobile application e sono: FLOWDROID, ICCTA e DROIDSAFE. Essi vengono valutati sia sulla suite di benchmarking DROIDBENCH e sia su un sottoinsieme di applicazioni prese dal benchmark F-DROID. In entrambi gli studi, i tool vengono confrontati e misurati in termini di efficienza ed efficacia.

Due dei lavori effettuati che riguardano comparazione tra tool di SQL Injection sono, ad esempio, [13] e [14]. Il primo presenta diversi tool in grado di rilevare o prevenire attacchi SQL Injection e viene valutato l'affrontare tutti i tipi di attacchi di SQL Injection tra gli strumenti attuali. I tool sono molteplici: WAVES, JDBC-CHECKER, CANDID, SQL-GUARD, SQL-CHECK, AMNESIA, WEBSSARI, SECURIFLY, SQL-IDS, SQL-PREVENT e SWADDLER. Il secondo, analogamente, presenta una comparazione tra tool di individuazione di SQL Injection e sono: BLIND SQL INJECTION, MARATHON, HAVIJ TOOLS, SQL BRUTE, SQL NINJA, ABSINTHE, PANGOLIN, SQLIER, SQ LSUS e SQLMAP. Essi non vengono valutati su benchmark come successo per i lavori di comparazione tra tool di Taint Analysis bensì su varie informazioni, come ad esempio: "eseguito automaticamente", "supporta il File System", "dipende dal Machine Learning", ecc.

CAPITOLO 3

Metodologia

3.1 Obiettivo di Ricerca

Per comprendere lo stato dell'arte della Taint Analysis e SQL injection, gli obiettivi sono molteplici:

- Cercare tool di identificazione di SQL Injection facenti uso di Taint Analysis;
- Cercare benchmark/dataset usati per valutare le capacità di tali tool;
- Valutare l'eseguibilità del maggior numero possibile di tool;
- Valutare le capacità dei tool eseguibili sui benchmark/dataset identificati;

3.1.1 Metodo di Ricerca

Lo studio è stato strutturato attorno a quattro domande di ricerca (RQs).

RQ$_{1.1}$. Quali sono i tool di identificazione di SQL Injection facenti uso di taint analysis disponibili?

Questa domanda di ricerca si pone l'obiettivo di ricercare tool per l'identificazione

di SQL injection tramite Taint Analysis e comprenderne il comportamento nei minimi dettagli per riconoscere le varie opzioni a disposizione degli sviluppatori e dei ricercatori.

RQ$_{1.2}$. Quali sono i benchmark/dataset usati per valutare i tool di identificazione di SQL Injection facenti uso di taint analysis?

Per valutare l'efficacia dei tool, è cruciale disporre di dataset o benchmark accurati e, soprattutto, dotati di oracolo per garantire la riproducibilità dei risultati. Questa domanda di ricerca nasce dalla necessità di individuare e valutare le risorse disponibili in modo da condurre test affidabili e rappresentativi delle capacità dei tool considerati.

RQ$_{2.1}$. Qual è il grado di eseguibilità dei tool disponibili?

L'implementazione pratica di qualsiasi tool è spesso influenzata dalla sua usabilità. Questa domanda di ricerca sorge dall'interesse di esaminare se i tool selezionati possono essere installati, configurati ed eseguiti senza difficoltà significative.

RQ$_{2.2}$. Quali sono le prestazioni dei tool eseguibili sui benchmark disponibili?

La misurazione delle prestazioni dei tool di detection di SQL Injection tramite Taint Analysis è un passo critico per valutarne l'efficacia. Questa domanda di ricerca è posta per valutare l'accuratezza, la recall e la precisione dei tool considerati, in modo da stabilire quali strumenti sono più adatti per l'individuazione e la prevenzione delle vulnerabilità SQL Injection.

3.1.2 Raccolta e Selezione dei Dati

Il motore di ricerca utilizzato per effettuare ricerche di informazioni riguardanti l'ambito della Taint Analysis è GOOGLE SCHOLAR, il quale dà la possibilità ai ricercatori di trovare articoli accademici, tesi di laurea e altro materiale tecnico riguardanti vari argomenti. Esso utilizza gli stessi algoritmi di ricerca utilizzati Google Search ma limita i risultati a riviste scientifiche, libri accademici, atti di convegno e tanto altro. [1] Il meccanismo di ricerca è molto simile a quello di Google ma con alcune differenze, ad esempio la possibilità di fare una ricerca per autore, per data di pubblicazione

[1] https://scholar.google.com/

e altri criteri. Questo lo si fa per raffinare i risultati e, quindi, ridurre il numero di articoli.

In relazione ai vari obiettivi di ricerca da rispettare bisognava creare una query di ricerca con lo scopo di identificare articoli e riferimenti che servissero per il lavoro di tesi da realizzare. Tale query, però, doveva essere scritta in modo da diminuire, il più possibile, il numero di risultati, e, quindi, l'unica modalità da seguire era l'uso di keyword, filtri, ecc. Sono state scelte le seguenti: `taint`, `taint analysis`, `SQL Injection`, `Android`, `Java`, `PHP`, `Tools`, `Benchmark`, `Suite`. In particolare, le prime 3 keywords sono state scelte perchè rappresentano gli argomenti principali del lavoro di tesi, invece, le altre 6 keywords, sono state scelte perchè l'obiettivo è capire come i tool rilevano SQL Injection nelle web application, le quali sono sviluppate, principalmente, tramite due linguaggi di programmazione: Java e PHP.[2] Inoltre, è stato scelto anche Android perchè anche le app mobile sono soggette a vulnerabilità di tipo SQL injection. Infine, l'obiettivo è testarli su varie suite di benchmarking che conterranno esempi minimali di web application o app mobile o altro. Queste parole chiave possono essere anche combinate insieme, attraverso dei filtri, ad esempio operatori AND, OR, ecc, che servono, appunto, per raffinare i risultati ottenuti.
Dopo una serie di prove e cambiamenti della query, il risultato è il seguente:

QUERY

("taint analysis") AND ("sql injection") AND (tool OR benchmark OR suite) AND (java OR android OR php)

Sono stati identificati vari criteri di inclusione/esclusione da applicare a ciascun risultato e sono state individuate varie informazioni che aiutassero a descrivere gli argomenti trattati in quel determinato articolo. Sono state identificate le seguenti informazioni:

- **URL**: collegamento diretto all'articolo;
- **Breve descrizione**: l'obiettivo del lavoro del documento;

[2] https://w3techs.com/

- **Autori**;
- **Anno di pubblicazione**;
- **Venue**: conferenza o journal dove è stato pubblicato l'articolo;

Nel processo di ricerca, sono stati individuati e selezionati attentamente articoli che si inquadrassero in modo preciso nel contesto del nostro studio. Questi articoli dovevano presentare lavori simili al nostro attuale, condurre revisioni della letteratura sulla Taint Analysis e sugli attacchi SQL Injection in generale, e infine, proporre nuovi strumenti per l'analisi e l'identificazione di potenziali vulnerabilità.
I criteri di inclusione/esclusione sono i seguenti:

- Un risultato viene scartato se:
  - E1. Non riguarda vulnerabilità di tipo SQL Injection.
  - E2. Non riguarda l'identificazione di vulnerabilità tramite Taint Analysis.
  - E3. Coinvolge un tool che scansioni applicazioni non sviluppate in PHP, Java o Android.
  - E4. L'articolo di riferimento non supera almeno cinque pagine.
  - E5. Si tratta di una tesi o di un libro di testo.
- Un risultato viene accettato se:
  - I1. Presenta un tool per l'identificazione di SQL Injection tramite taint analysis.
  - I2. Presenta un benchmark o dataset per validare tool per l'identificazione di SQL Injection.
  - I3. Confronta tool di identificazione di SQL Injection tramite taint analysis.

I criteri di esclusione sono in OR, ovvero se almeno uno dei cinque risulta verificato allora si scarta. I criteri di inclusione sono anch'essi in OR, ovvero se almeno uno dei tre risulta verificato allora si accetta.
In particolare, il primo step per verificare l'aderenza dei criteri scelti sono la lettura del titolo, dell'abstract e una rapida ricerca all'interno dell'articolo per verificare

la presenza delle keyword considerate; ad esempio, se all'interno dell'articolo non era presente l'espressione "taint analysis", allora il risultato veniva scartato poiché rispettava almeno uno dei criteri di "esclusione". Dopodiché, sul set di risultati residui, ogni articolo veniva letto e venivano scartati quelli non rilevanti a valle della lettura completa.

3.2 Metodologia di identificazione dei Tool/Benchmark

Innanzitutto, spieghiamo, brevemente, quanti e come sono stati scelti i tool. Inizialmente, si andava ad effettuare una ricerca esaustiva all'interno dell'articolo di riferimento, anche tramite parole chiave che potessero aiutare a ritrovare un link ad un web site o ad un loro profilo come GITHUB, GITLAB, ecc. Se era presente tale link, allora il tool veniva considerato, altrimenti si passava allo step successivo, ovvero si andava ad effettuare una ricerca su Google Search al fine di individuare sempre un suo web site o un suo profilo di riferimento; il tutto, sempre rimanendo entro le due pagine di risultati restituiti dal motore di ricerca. Se si trovava tale profilo, il tool veniva considerato, altrimenti si scartava in quanto si sarebbe trattato di un tool sicuramente implementato e sviluppato da ricercatori, ma non reso online dagli stessi.

3.2.1 Metriche di valutazione per i Tool

Nella valutazione delle prestazioni dei tool di detection di SQL injection tramite Taint Analysis, è essenziale stabilire metriche oggettive e accurate per misurare l'efficacia di ciascuno strumento. Le seguenti metriche sono state selezionate al fine di fornire una valutazione completa e comparabile dei tool esaminati:

- Matrice di confusione (Confusion Matrix): La matrice di confusione è uno strumento di valutazione che rappresenta le prestazioni di un sistema di classificazione, come un tool di detection, suddividendo le previsioni in quattro categorie in base alla loro correttezza:

  - Veri positivi (VP): sono i casi in cui sia il sistema che la realtà concordano sul fatto che l'istanza sia positiva (ad esempio, il tool ha rilevato correttamente una vulnerabilità SQL Injection presente nell'applicazione).
  - Veri negativi (VN): indicano i casi in cui sia il sistema che la realtà concordano sul fatto che l'istanza sia negativa (ad esempio, il tool non rileva una vulnerabilità non presente nell'applicazione).
  - Falsi positivi (FP): si verificano quando il sistema etichetta erroneamente un'istanza come positiva, ma in realtà non è positiva (ad esempio, il tool segnala una vulnerabilità non presente nell'applicazione).
  - Falsi negativi (FN): si verificano quando il sistema non rileva correttamente un'istanza positiva (ad esempio, il tool non segnala una vulnerabilità che invece è presente nell'applicazione).

- Precisione (Precision): misura la capacità di un modello o di un sistema di rilevamento (come un tool di identificazione di SQL Injection) di identificare correttamente le istanze positive tra tutte le istanze che ha etichettato come positive. In altre parole, la precisione indica quanto siano affidabili le segnalazioni positive del tool.

$$\text{Precisione} = \frac{\text{VP}}{\text{VP} + \text{FP}}$$

- Accuratezza (Accuracy): misura la proporzione di risultati corretti (veri positivi e veri negativi) rispetto al totale dei risultati. L'accuratezza è un indicatore generale delle capacità di un tool nel rilevare correttamente sia le istanze di SQL Injection che i risultati negativi.

$$\text{Accuracy} = \frac{\text{VP} + \text{VN}}{\text{VP+VN+FP+FN}}$$

- Recall: rappresenta la proporzione di veri positivi (istanze di SQL Injection rilevate correttamente) rispetto a tutte le istanze di SQL Injection presenti nel benchmark/dataset. Questa metrica è importante per valutare la capacità di un tool di individuare in modo efficace le vulnerabilità reali.

$$\text{Recall} = \frac{\text{VP}}{\text{VP} + \text{FN}}$$

- F1-Score: è la media armonica tra precision e recall. Questa metrica tiene conto sia dei falsi positivi che dei falsi negativi ed è particolarmente utile quando si desidera bilanciare la precisione e la capacità di rilevamento.

$$\text{F1-Score} = 2 \times \frac{\text{Precisione} \times \text{Recall}}{\text{Precisione} + \text{Recall}}$$

- Tempo di esecuzione (in secondi): rappresenta il tempo richiesto dal tool per analizzare un'applicazione (o un set di applicazioni); esso è un aspetto critico, specialmente in scenari in cui l'analisi deve essere eseguita in tempo reale o su grandi applicazioni. Questa metrica misura l'efficienza temporale del tool.

CAPITOLO 4

Risultati

4.1 Risultati del processo di Literature Review

Dal lancio della query su GOOGLE SCHOLAR, descritto in sezione 3.1.2, sono stati ottenuti 1,300 risultati. Di questi, sono stati selezionati i primi 100 corrispondenti alle prime 10 pagine restituite da GOOGLE SCHOLAR poichè ci si era accorto che i risultati successivi diminuivano di rilevanza per l'obiettivo di tesi. Dopo l'applicazione dei criteri di inclusione/esclusione, sono stati scartati 64 articoli per un totale rimasto di 36 articoli. In seguito a questa fase, sono stati scartati altri 6 risultati poiché si trattavano di lavori che non davano la possibilità di accedere all'articolo, in quanto richiedevano l'acquisto dello stesso. Infine, di quelli rimanenti, si è andati ad effettuare una lettura completa per verificare, più approfonditamente, la loro rilevanza o meno per l'obiettivo di tesi e ne sono stati scartati altri 11. Di conseguenza, il totale è di 19 articoli letti, rappresentati nella Tabella 4.1:

Tabella 4.1: Tabella dei risultati ritenuti rilevanti a valle della lettura completa e dell'applicazione dei criteri di inclusione/esclusione.

Nome	Anno	Venue
ConsiDroid: A concolic-based tool for detecting SQL injection vulnerability in android apps [15]	2019	IET Research Journals
Secucheck: Engineering configurable taint analysis for software developers [5]	2021	International Working Conference on Source Code Analysis and Manipulation (SCAM)
JOZA: Hybrid taint inference for defeating web application SQL injection attacks [2]	2015	IEEE/IFIP International Conference on Dependable Systems and Networks
Effective Dynamic Taint Analysis of Java Web Applications [3]	2022	International Conference on Bigdata Blockchain and Economy Management (ICBBEM)
Fluently specifying taint-flow queries with fluentTQL [5]	2021	International Working Conference on Source Code Analysis and Manipulation (SCAM)
Using positive tainting and syntax-aware evaluation to counter SQL injection attacks [4]	2006	Conference FSE: Foundations of Software Engineering
Dynamic taint propagation for Java [16]	2005	Annual Computer Security Applications Conference (ACSAC)
Simulation of Built-in PHP Features for Precise Static Code Analysis. [17]	2014	Internet Society, ISBN
SIDP-SQL Injection Detector and Preventer [18]	2012	International Journal of Computer Application (IJCA)
Security analysis of the OWASP benchmark with Julia [19]	2017	In Proceedings of the First Italian Conference on Cybersecurity (ITASEC17)
Static Identification of Injection Attacks in Java [20]	2019	ACM Transactions on Programming Languages and Systems
Web application recognition and statistical analysis to reduce vulnerabilities in data mining [21]	2021	Industrial Engineering Journal
SQLSCAN: A Framework to Check Web Application Vulnerability [22]	2016	International Institute for Science, Technology and Education (IISTE)
Large scale generation of complex and faulty PHP test cases [23]	2016	IEEE International Conference on Software Testing, Verification and Validation (ICST)
TChecker: Precise Static Inter-Procedural Analysis for Detecting Taint-Style Vulnerabilities in PHP Applications [7]	2022	CCS: ACM SIGSAC Conference on Computer and Communications Security
SQL Injection Detection and Prevention Tools Assessment [13]	2010	International Conference on Computer Science and Information Technology
Towards Web Application Security by Automated Code Correction [6]	2020	International Conference on Evaluation of Novel Approaches to Software Engineering
Static taint analysis tools to detect information flows [24]	2018	Conference Software Engineering Research and Practice
Finding application errors and security flaws using pql: A program query language [25]	2005	ACM SIGPLAN Notices

4.1.1 RQ1.1 - Analisi dei Tool Esistenti

In particolare, per ciascuno dei tool, trovati in letteratura, verrà presentata una breve descrizione evidenziandone le caratteristiche principali, le peculiarità e limitazioni. Inoltre, se l'articolo di riferimento lo rappresenta, verrà presentata anche l'architettura di base sottoforma di Figura. La Tabella 4.2 riassume le differenze sostanziali tra tali tool.

Tabella 4.2: Tabella di informazioni riassuntive per i tool identificati mediante il processo di Literature Review.

Nome	Contesto	Tipo Taint Analysis	Vulnerabilità	Disponibile
SECURIFLY [25]	Java web app	Dinamica	SQL Injection	NO
AUSERA[1] [1]	Mobile app	Statica	Command Injection	SI
SIDP [18]	Web/Mobile app	Ibrida	SQL Injection	NO
JOZA [2]	Web/Mobile app	Ibrida	SQL Injection	NO
SQL-SCAN[2] [22]	PHP/Java web app	Statica	SQL Injection	SI
WAP[3] [21]	PHP web app	Ibrida	Command Injection	SI
FAST-TAINT [3]	Java web app	Dinamica	Command Injection	NO
WASP [4]	PHP web app	Dinamica	SQL Injection	NO
SECUCHECK[4] [5]	Java/Mobile web app	Statica	SQL Injection	SI
CONSIDROID [15]	Mobile app	Ibrida	SQL Injection	NO
WIRECAML[5] [26]	PHP web app	Statica	Command Injection	SI
PHPCORRECTOR [6]	PHP web app	Statica	SQL Injection e XSS	NO
TCHECKER[6] [7]	PHP web app	Statica	SQL Injection	SI
RIPS[7] [17]	PHP web app	Statica	Command Injection	SI

SECURIFLY [25]. È un tool implementato per Java. Può essere integrato con un web server in modo che ogni volta che viene aggiunta una nuova applicazione web, questa venga instrumentata automaticamente. Ciò elimina le preoccupazioni legate alla distribuzione di applicazioni web sconosciute potenzialmente non sicure. Inoltre, SECURIFLY non richiede modifiche al programma originale e non ha bisogno di accedere a nulla oltre al bytecode finale. Ciò può essere particolarmente vantaggioso quando si tratta di applicazioni che dipendono da librerie di cui non si ha accesso al codice sorgente. A differenza di altri tool, segue le stringhe invece dei caratteri per le informazioni di contaminazione. In pratica, il controllore dinamico effettuerà una corrispondenza ogni volta che una stringa, controllata dall'utente, fluisce in qualche modo verso un sink sospetto, indipendentemente dal fatto che un input sia dannoso o meno. SECURIFLY reagirà sostituendo la stringa potenzialmente pericolosa con una sicura. Cerca, quindi, di sanificare le stringhe di query che sono state generate utilizzando input contaminati, ma purtroppo l'iniezione nei campi numerici non può essere bloccata con questo approccio. Infatti, la difficoltà nell'identificare tutte le fonti di input dell'utente è la principale limitazione di questo approccio.

AUSERA [1]. È un tool automatizzato per rilevare le vulnerabilità di sicurezza nelle app mobile. Una delle caratteristiche principali di AUSERA è la sua tassonomia delle vulnerabilità che fornisce una classificazione completa e sistematica dei tipi di vulnerabilità comuni nelle app mobile. Infatti, prende in input un'app, l'insieme di parole chiave dei dati sensibili con una rappresentazione formale tramite espressioni regolari, l'insieme di sorgenti/sink e, infine, restituisce l'insieme di vulnerabilità di sicurezza nell'app in fase di test, il livello di danni, gli attacchi potenziali e i metodi di correzione. Di conseguenza, AUSERA non è specializzato esclusivamente per la rilevazione delle vulnerabilità di SQL Injection bensì copre una vasta gamma di tipi di vulnerabilità. Tuttavia, ciò non esclude la possibilità che AUSERA rilevi potenziali vulnerabilità di SQL Injection, a condizione che tale vulnerabilità sia inclusa nella sua tassonomia delle vulnerabilità. Di seguito mostrata l'architettura di base:

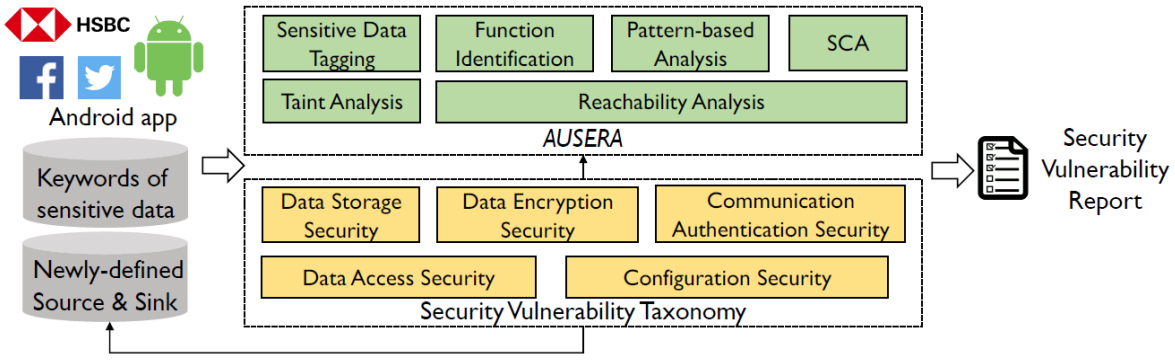

Figura 4.1: Architettura di AUSERA. Figura presa dall'articolo originale. [1]

SIDP (SQL Injection Detection and Prevention) [18]. È un tool progettato per rilevare e prevenire gli attacchi di SQL Injection nelle applicazioni web. Utilizza diverse tecniche, una tra queste è il concetto di "positive tainting" che si limita ai dati fidati anzichè dei dati non fidati. Il motivo è che il "negative tainting" porta a fidarsi di dati che non dovrebbero essere fidati; di conseguenza, avremo abbastanza falsi negativi. Un'altra tecnica usata da SIDP è la *propagazione del taint*, effettuata in fase di esecuzione. Consiste nell'identificare i marcatori di taint associati ai dati mentre quest'ultimi vengono utilizzati e manipolati dagli utenti durante l'esecuzione. I dati sono costituiti da caratteri. Pertanto, per garantire l'accuratezza, il tainting viene effettuato a livello di carattere e le stringhe, invece, vengono costantemente suddivise in sottostringhe per la creazione di query SQL. Ecco come funziona SIDP: gli attori sono lo Sviluppatore e l'applicazione web; quest'ultima invoca un controllore di token, per verificare la validità dei token di accesso, un rilevatore di stringhe, per individuare le potenziali vulnerabilità di SQL Injection, e accede alla libreria di stringhe meta per l'accesso e la gestione delle stringhe fidate. Allo stesso modo, lo sviluppatore imposta le politiche di trust e contrassegna le stringhe fidate. Dal punto di vista dell'utente, invece, comunica con il database passando prima per (in sequenza) l'applicazione web, il controllore e la Tabella della libreria.

JOZA [2]. JOZA è composto da due componenti di analisi principali, l'analisi PTI (Positive Taint Inference) e l'analisi NTI (Negative Taint Inference). La componente di analisi PTI implementa l'algoritmo di inferenza di taint positivo il quale consiste

nel dedurre le parti di una stringa di query SQL che dovrebbero essere considerate affidabili e funziona, quindi, ricostruendo comandi critici per la sicurezza utilizzando frammenti di stringa estratti dal programma. La componente di analisi NTI implementa l'algoritmo di inferenza di taint negativo il quale consiste nel dedurre le marcature di contaminazione tracciando le correlazioni tra gli input dell'applicazione e le stringhe di query. Utilizza un algoritmo di corrispondenza approssimativa delle stringhe per tener conto delle piccole trasformazioni eseguite dall'applicazione sulla stringa, come la rimozione degli spazi vuoti, conversioni maiuscole/minuscole, ecc. Infine, tutti i comandi destinati al sistema di gestione del database (DBMS) sul backend vengono intercettati e inviati prima alla componente di analisi PTI e poi alla componente di analisi NTI prima di avere il permesso di procedere verso il DBMS. Di seguito mostrata l'architettura di base:

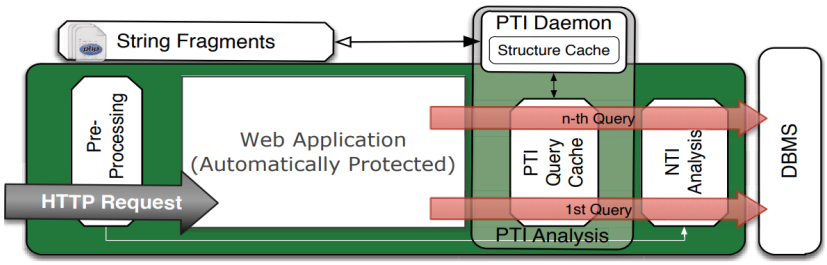

Figura 4.2: Architettura di JOZA. Figura presa dall'articolo orginale. [2]

SQL-SCAN [22]. È un tool sviluppato in Java e funziona in qualsiasi ambiente. Prende in input un'intera applicazione web, analizza ogni istruzione SQL e le verifica. Tutte le possibili minacce vengono visualizzate come output. Ecco una descrizione, ad alto livello, delle funzionalità di base:

1. Analizza singolarmente ogni file dell'applicazione per verificare l'esecuzione di SQL vulnerabili;
2. Mostra tutte le istruzioni SQL (minacce) in un'interfaccia utente intuitiva, fornendo il nome del file, il numero di riga e la causa della vulnerabilità;
3. Fornisce consigli agli utenti su cosa fare per ogni minaccia;

4. Una volta che l'utente risolve un problema, può contrassegnare la minaccia come risolta in modo che non venga visualizzata nella successiva esecuzione;

5. Può essere facilmente aggiornato per soddisfare le esigenze dei rapidi cambiamenti tecnologici.

WAP (Web Application Protection) [21]. Copre un considerevole numero di classi di vulnerabilità, tra cui SQL Injection e XSS (e non solo). WAP assume che il database di background sia MySQL, DB2 o PostgreSQL. L'approccio che usa è composto da tre moduli:

1. Analizzatore di codice;

2. Predittore di falsi positivi;

3. Correttore di codice;

4. Testing;

L'analizzatore di codice analizza inizialmente il codice sorgente PHP, generando un AST (Abstract Syntax Tree) per eseguire la taint analysis, ossia per tracciare se i dati forniti dagli utenti attraverso i punti di ingresso (input) raggiungono sink sensibili senza essere sanificati. Durante questa analisi, l'analizzatore d codice genera tabelle di simboli tainted e alberi di percorso di esecuzione contaminati per quei percorsi che collegano i punti di ingresso a sink sensibili senza una corretta sanificazione. Il predittore di falsi positivi prosegue da dove si ferma l'analizzatore di codice. Per ogni sink sensibile che è stato raggiunto da input contaminati, viene tracciato il percorso da quel sink al punto di ingresso utilizzando le tabelle e gli alberi che genera l'analizzatore. Lungo i percorsi tracciati vengono raccolti vettori di attributi (istanze) e classificati dall'algoritmo di data mining come veri positivi (vera vulnerabilità) o falsi positivi (non una vera vulnerabilità). Il correttore di codice seleziona i percorsi classificati come veri positivi per segnalare gli input contaminati che devono essere sanificati utilizzando tabelle e gli alberi sopra menzionati. Di conseguenza, il codice sorgente viene corretto inserendo le correzioni, ad esempio, chiamate a funzioni di sanificazione.

FAST-TAINT [3]. In primo luogo, vengono raccolte varie funzioni di input per diversi server web Java; i dati trasmessi dalle funzioni di input vengono contrassegnati come **tainted**, viene calcolato il codice hash e gli oggetti vengono aggiunti all'heap di taint. Poichè i tipi di dati in Java sono numerosi e complessi, FastTaint scompone gli oggetti complessi, li trasforma in un singolo oggetto per calcolare il valore dell'inidirizzo dell'oggetto e quindi aggiunge il valore dell'indirizzo all'heap di taint. Nel linguaggio java, esistono strutture dati come integer, long, double, float, ecc. Pertanto, per rappresentare accuratamente il taint, ad esempio, in un array, solo alcuni elementi sono contaminati, quindi non è possibile contrassegnare l'intero array come contaminato altrimenti si avrebbe un eccesso di marcatura di taint. Il tool FastTaint scompone l'oggetto array in un singolo oggetto e calcola il codice hash del singolo oggetto. In secondo luogo, in base alla strategia di propagazione del taint, tramite un algoritmo di marcatura del taint, in seguito all'aggiunta degli oggetti all'heap di taint, viene registrata la traiettoria di propagazione del taint; in base alla strategia di controllo del taint definita dall'utente, vengono rimossi dalla memoria i marcatori di taint corrispondenti all'heap di taint, al fine di ridurre il consumo di memoria e migliorare l'efficienza dell'analisi di taint. Infine, viene eseguita la strategia di controllo del taint nel punto di fuga del taint; in questo modo, si determina la presenza di vulnerabilità ed è possibile emettere un rapporto dettagliato sulla vulnerabilità riscontrata. Di seguito mostrata l'architettura di base:

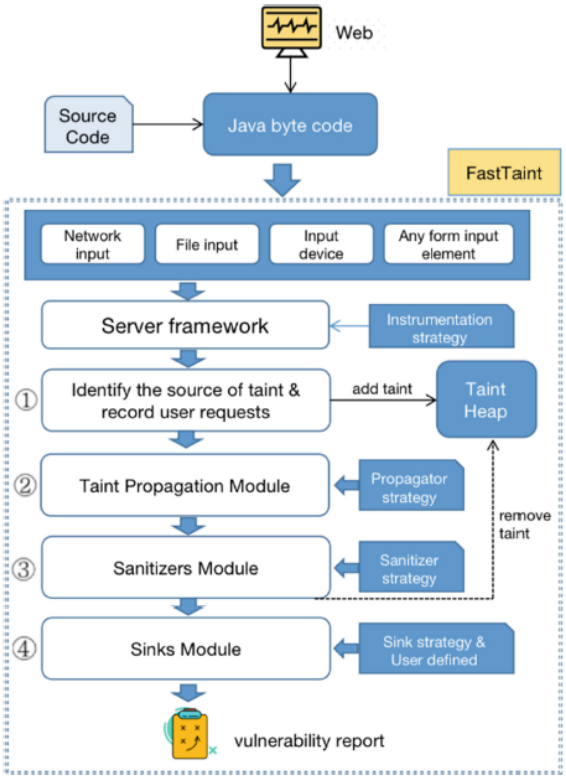

Figura 4.3: Architettura di FAST-TAINT. Figura presa dall'articolo originale. [3]

WASP [4]. È un tool che implementa una tecnica di prevenzione dell' SQLi per le applicazioni web basate su Java. In maniera intuitiva, l'approccio usato dal tool funziona identificando le stringhe "affidabili" in un'applicazione e consentendo solo a queste stringhe di essere utilizzate per creare determinate parti di una query SQL, come parole chiave o operatori. Il meccanismo generale per implementare questo approccio si basa sul "dynamic tainting", che contrassegna e traccia determinati dati in un programma durante l'esecuzione. La tipologia di "dynamic tainting" utilizzata dal tool offre importanti vantaggi rispetto alle altre tipologie. Molte tecniche si basano su analisi statiche complesse al fine di individuare potenziali vulnerabilità nel codice, ma possono generare elevati falsi positivi o possono avere problemi di scalabilità quando applicate a grandi applicazioni complesse. L'approccio di WASP, invece, è altamente automatizzato e richiede, a volte, un intervento minimo o nullo dello sviluppatore. Inoltre, non richiede infrastrutture aggiuntive e può essere distribuito automaticamente. Il primo vantaggio è l'uso del "positive tainting", il quale identifica e traccia i dati affidabili, a differenza del tradizionale "negative tainting", il quale

si concentra sui dati non affidabili. Nel contesto delle applicazioni web, le fonti di dati affidabili possono essere identificate in modo più semplice e accurato rispetto a quelli non affidabili, migliorando l'automazione. Il secondo vantaggio è l'uso di una valutazione flessibile che tiene conto della sintassi, la quale fornisce agli sviluppatori un meccanismo per regolare l'uso dei dati stringa in base anche al loro ruolo sintattico in una query. In questo modo, gli sviluppatori possono utilizzare una vasta gamma di fonti di input esterne per creare query, proteggendo allo stesso tempo l'applicazione da possibili attacchi introdotti tramite tali fonti. Tale approccio impone un basso overhead all'applicazione, richiede requisiti minimi di implementazione, non richiede alcun sistema di runtime personalizzato o infrastrutture aggiuntive. Di seguito mostrata l'architettura di base:

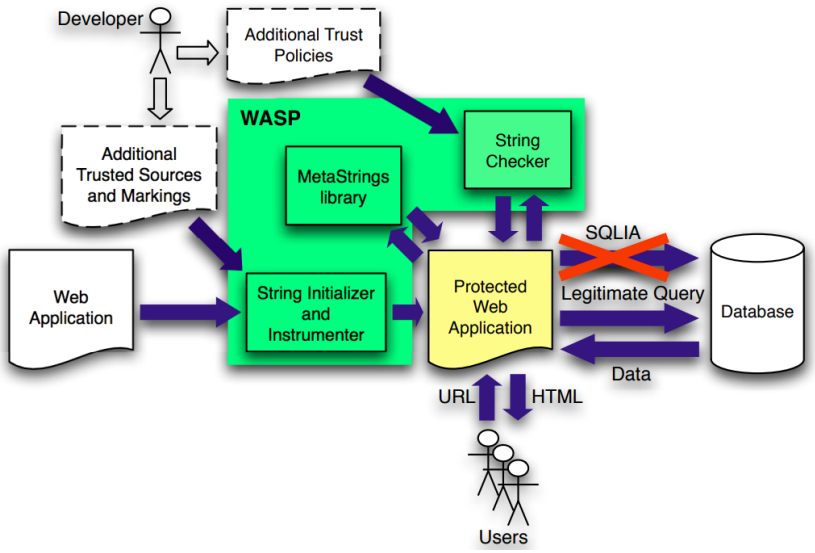

Figura 4.4: Architettura di WASP. Figura presa dall'articolo originale. [4]

SECUCHECK [5]. È un tool di taint analysis open-source, configurabile in più ambienti di sviluppo. Lavora in vari step:

1. Tramite l'uso di un framework di analisi statica, denominato Soot, trasforma il bytecode Java compilato in una rappresentazione intermedia a 3 indirizzi semplice su cui vengono eseguite le analisi del flusso di dati. Soot dispone di analisi semplici, ad esempio l'eliminazione del codice morto, e fornisce due strutture dati principali, ovvero un grafo del flusso di controllo per le analisi intra-procedurali e una grafo delle chiamate per le analisi inter-procedurali;

2. SECUCHECK integra due risolutori di analisi del flusso di dati, BOOMERANG e FLOWDROID, entrambi basati su Soot. Boomerang è un'analisi che permette al client di creare query per una determinata posizione (variabile o istruzione) per calcolare una struttura ad albero di percorsi raggiungibili da quella posizione. Di conseguenza, SECUCHECK utilizza questa struttura ad albero per rilevare flussi di dati potenzialmente pericolosi. Boomerang è il risolutore di default; tuttavia, l'utente può utilizzare FLOWDROID, il quale è stato progettato come un motore di tracciamento di dati potenzialmente pericolosi per le app mobile che, dato un elenco di sorgenti e sink, segnala i flussi di dati pericolosi esistenti tra coppie di sorgenti/sink;

3. Tramite l'uso di MAGPIEBRIDGE, che utilizza il protocollo LSP, si può comunicare con qualsiasi IDE compatibile con LSP; infatti, MAGPIEBRIDGE esegue l'analisi dell'utente tramite una pagina di configurazione HTML e restituisce i risultati in formato JSON all'IDE. Il protocollo LSP supporta molte funzionalità standard dell'interfaccia utente, come l'evidenziazione della sintassi, i marcatori degli errori, i messaggi per la visualizzazione degli errori e altre informazioni.

Di seguito mostrata l'architettura di base:

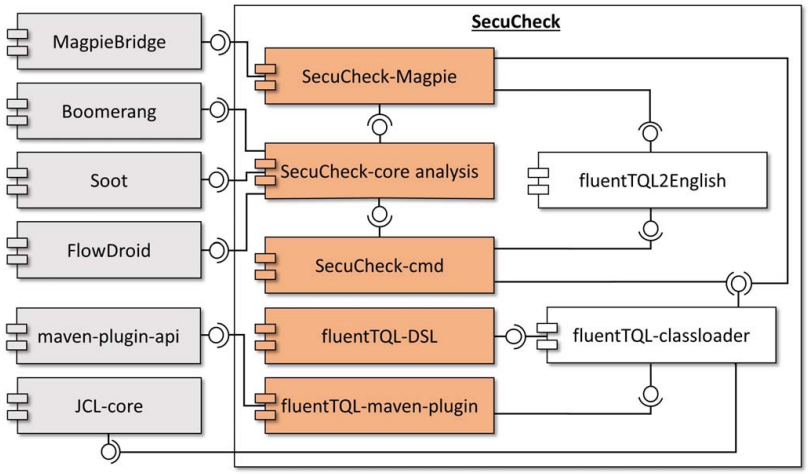

Figura 4.5: Architettura di SECUCHECK. Figura presa dall'articolo originale. [5]

PHPCORRECTOR [6]. PHPCORRECTOR è un tool sviluppato in Python e utilizza come parser una versione modificata di PHPly5, che supporta sia PHP 5 che PHP 7.

La Taint analysis, le simulazioni e le correzioni sono stati implementati da zero. L'approccio di base mira a correggere le applicazioni web PHP inserendo nuove righe di codice che sanificano o convalidano gli input che arrivano ai punti di ingresso, i quali vengono successivamente utilizzati in sink sensibili in modo insicuro. L'intenzione è quella di evitare possibili errori sintattici o interruzioni della logica dell'applicazione ma senza correggerne il comportamento funzionale. Il tool inizia ricevendo in input una porzione di codice, contenente un flusso dati che inizia da uno (o più) punti di ingresso e termina in sink sensibile, insieme alle informazioni sul tipo di vulnerabilità che il codice potrebbe subire. Successivamente, il codice viene analizzato mediante una soluzione ispirata al tracciamento del taint per scoprire quali variabili devono essere sanificate o convalidate e dove correggere la porzione. L'output del tool è un blocco di codice sicuro. Se il tool considera la porzione di codice non vulnerabile, non verranno apportate modifiche. Di seguito mostrata l'architettura di base:

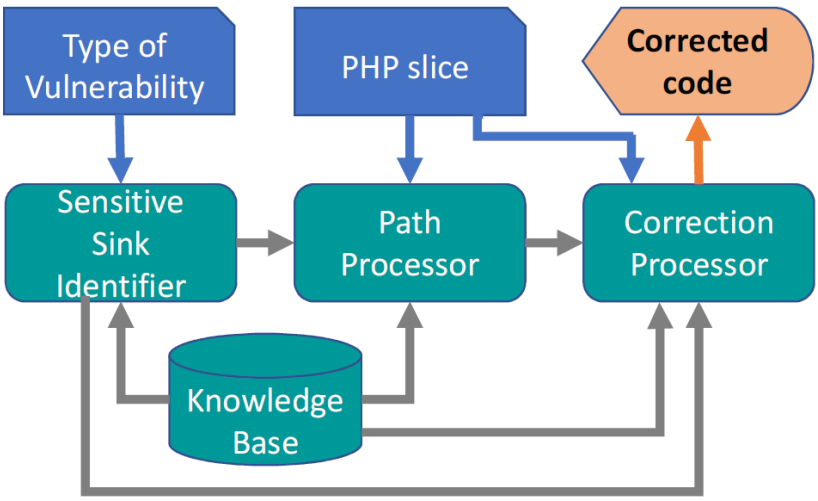

Figura 4.6: Architettura di PHPCORRECTOR. Figura presa dall'articolo originale. [6]

CONSIDROID [15]. È un framework composto da cinque parti principali:

1. Analisi Statica: in questa fase, il primo obiettivo è generare la funzione principale necessaria per compilare ed eseguire l'applicazione nella Java Virtual Machine (JVM). Questa funzione principale viene creata in una classe che prende il nome di **DummyMain**. Viene esteso il codice dell'applicazione con un insieme di classi **DummyMain**, ognuna delle quali rappresenta un possibile percorso di esecuzione nell'applicazione originale. Il secondo obiettivo, invece,

è ottimizzare l'analisi dinamica, rendendola ibrida e mirata; in altre parole, attraverso l'analisi statica, si limita l'esecuzione di un'applicazione ai percorsi desiderati, o anche detti "percorsi vulnerabili";

2. Classi Mock e Classi Mock Simboliche: vengono utilizzate per modellare l'SDK e inoltre, per la taint analysis, si deve tener traccia della propagazione delle variabili contaminate dalle fonti alle funzioni "sink"(dove i dati contaminati possono avere effetto);

3. Motore esteso di Esecuzione Concolica: il motore di esecuzione concolica è SPF, un tool di test java. Estendendo il codice dell'applicazione (senza modificarlo), è possibile testare le classi Mock con SPF. Si estende SPF in due aspetti: primo, viene sviluppato un componente per il rilevamento delle vulnerabilità di SQL injection; secondo, viene gestita l'esecuzione concolica per esaminare i percorsi vulnerabili estratti dall'analisi statica iniziale;

4. Rapporto di Rilevamento delle Vulnerabilità: è la fase in cui vengono presentati i risultati che aiuteranno gli sviluppatori a correggere le loro app e risolvere le vulnerabilità di SQL injection. Vengono presentati l'ID e il nome delle funzioni sorgenti e di "sink" (dove i dati contaminati possono avere effetto);

5. Robolectric: è uno strumento di test unitario per app mobile per convalidare i risultati.

WIRECAML [26]. WIRECAML è un tool di rilevazione di SQL Injection tramite l'uso della taint analysis e Machine Learning. Come primo passo, vengono estratte le features e vengono analizzati i file contenenti codice sorgente PHP convertendoli in alberi di sintassi astratta (AST) tramite l'uso del parser phply. Successivamente, vengono ricavati i CFG dagli AST, da cui vengono estratte le caratteristiche. Sebbene il focus è su PHP, vengono selezionate le caratteristiche più neutre possibile rispetto al linguaggio in modo che l'approccio di base può essere applicato ad altri linguaggi dinamici nelle ricerche successive. Un CFG rappresenta tutti i percorsi che potrebbero essere attraversati dal codice durante la sua esecuzione. Vengono applicate poi l'analisi delle definizioni raggiungibili, la taint analysis e l'analisi delle

costanti raggiungibili per estrarre le caratteristiche dai CFG. Non si entrerà troppo nel dettaglio in merito al suo comportamento funzionale.

TCHECKER [7]. È un tool di analisi statica inter-procedurale sensibile al contesto e con supporto per le caratteristiche orientate agli oggetti per identificare vulnerabilità di tipo "taint" nelle applicazioni PHP. La principale sfida nell'eseguire un'analisi di taint inter-procedurale su programmi PHP è analizzare le relazioni di chiamata. A tal fine, TCHECKER costruisce innanzitutto un grafo delle chiamate, il che gli consente, in seguito, di eseguire una propagazione di taint più accurata attraverso le chiamate di funzione. Il grafo delle chiamate è costruito per contenere già tutte le possibili funzioni di destinazione di un sito di chiamata in modo sensibile al contesto, altrimenti TCHECKER dovrebbe dedurre ripetutamente il target di chiamata di un sito di chiamata ogni volta che viene incontrato. Inoltre, il grafo delle chiamate consente a TCHECKER di eseguire un'analisi di taint selettiva. Ad esempio, TCHECKER potrebbe saltare l'analisi di taint per un sito di chiamata se tutte le sue funzioni di destinazione non utilizzano alcuna variabile contaminata. Questo potrebbe evitare un'analisi inutile su funzioni non rilevanti e migliorare quindi l'efficienza complessiva dell'analisi. In conclusione, dato il codice sorgente di un'applicazione PHP, TCHECKER costruisce incrementalmente il suo grafo delle chiamate deducendo i target delle chiamate presso ciascun sito di chiamata; esegue quindi un'analisi di taint inter-procedurale sensibile al contesto su tutto il programma per identificare vulnerabilità di tipo taint. Di seguito mostrata l'architettura di base:

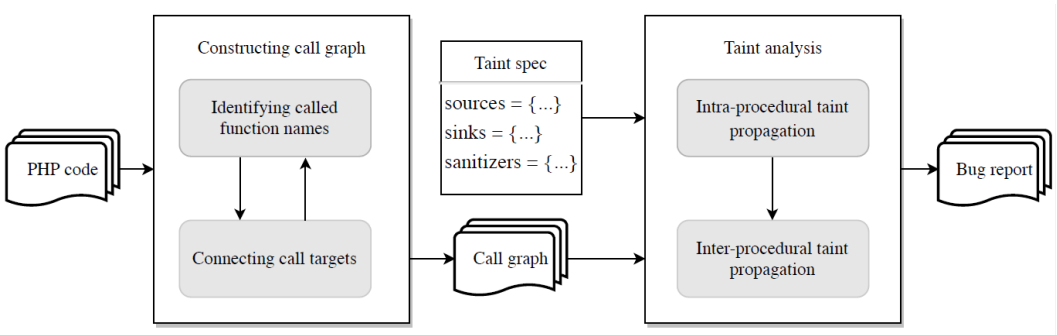

Figura 4.7: Architettura di TCHECKER. Figura presa dall'articolo originale. [7]

RIPS [17]. RIPS ha l'obiettivo di rilevare tutte le vulnerabilità di tipo "taint" e cercare di modellare il linguaggio PHP nel modo più preciso possibile. L'approccio di base utilizza riepiloghi di blocco, funzione e file per memorizzare i risultati dell'analisi del flusso di dati all'interno di ciascuna unità e per costruire un modello astratto di flusso di dati per un'analisi efficiente. Più precisamente vengono eseguiti i seguenti passaggi:

1. Per ciascun file PHP nel progetto, viene costruito un Albero di Sintassi Astratto (AST) basato sulle interne di PHP open source. Inoltre, tutte le funzioni definite dall'utente vengono estratte e informazioni rilevanti come il nome e i parametri vengono memorizzate nell'ambiente. Il corpo della funzione è salvato come un AST separato e rimosso dall'AST principale del file analizzato.

2. Si inizia a trasformare ciascun AST principale in un Grafo di Flusso di Controllo (CFG). Ogni volta che un nodo dell'AST esegue un salto condizionale, viene creato un nuovo blocco di base e collegato al blocco di base precedente con un arco di blocco. La condizione di salto viene aggiunta all'arco di blocco e i nodi AST successivi vengono aggiunti al nuovo blocco di base.

3. Si simula il flusso di dati di ciascun blocco di base non appena viene creato un nuovo blocco di base. Il vantaggio principale è che l'analisi di un blocco di base dipende solo dai blocchi di base precedenti quando si esegue un'analisi del flusso di dati orientata all'indietro. Inoltre, i risultati dell'analisi vengono integrati nel cosiddetto riepilogo di blocco creato durante la simulazione. Esso riassume il flusso di dati all'interno di un blocco di base.

4. Se durante la simulazione viene incontrata una chiamata a una funzione definita dall'utente precedentemente sconosciuta, viene costruito il CFG dall'AST della funzione e viene creato un riepilogo della funzione con un'analisi intra-procedurale. Successivamente, le pre e post-condizioni per questa funzione possono essere estratte dal riepilogo e viene eseguita un'analisi inter-procedurale. Infine, la costruzione del CFG principale continua.

5. Si conduce una taint analysis a partire dal blocco di base attualmente simulato per ciascun parametro vulnerabile di una funzione definita dall'utente o per un sink sensibile configurato.

4.1.2 RQ1.2 - Analisi dei Benchmark/Dataset Esistenti

Come per i tool, anche per i benchmark è stato applicato un procedimento molto simile per assicurarsi della loro disponibilità online e, soprattutto, che siano in possesso di un oracolo in modo da poter misurare le prestazioni del tool sulla base del suo comportamento. I dataset/benchmark rilevati per valutare i tool di identificazione di SQL injection facenti uso di Taint Analysis sono descritti nella Tabella 4.3.

Tabella 4.3: Tabella dei benchmark usati per valutare i tool di identificazione di SQL Injection, tramite Taint Analysis.

Nome	Tipo test-case	Numero test-case
SecuriBench-Micro [8] [24]	Java	123
FluentTQL [9] [5]	Java	5
DROIDBENCH [10] [15]	Mobile app	112
PHP-Vulnerability-Test-Suite [11] [23]	PHP	9,552

- SecuriBench-Micro (versione 1.08) [24]: progettata per le applicazioni web basate su Java, contiene una serie di piccoli casi di test, con oracoli disponibili, che semplificano il processo di confronto.

- FluentTQL [5]: progetto di micro-benchmark comprendente vari tipi di vulnerabilità. Siccome il focus principale della tesi è sull'SQL injection, si è decisi di tener conto solo delle due cartelle **NoSQLInjection**, contenente 2 file java e la cartella **SQLInjection** contenente 3 file java per un totale di 5 casi di test.

- DROIDBENCH [15]: uno dei benchmark più famosi di applicazioni mobile. Esso contiene sia i progetti in formato apk e sia gli stessi sviluppati in Eclipse.

- PHP-Vulnerability-test-suite[12] [23]: è un benchmark di 27,000 file contenti piccole porzioni di codice PHP, tra cui alcuni non vulnerabili e altri contenenti vari tipi di vulnerabilità ma, siccome il focus della tesi è sull'SQL injection, sono stati eliminati i file che non la riguardavano.

4.2 Risultati Esecuzione Tool sui Benchmark

In questa sezione verrà presentato, per ogni tool disponibile online, il lavoro realizzato per poterlo configurare, installare ed eseguire sui vari benchmark selezionati. Poiché la maggior parte dei tool presenta un'interfaccia a riga di comando, per la loro configurazione ed esecuzione è stata usato un ambiente Ubuntu versione 22.04.02, tranne per RIPS che è dotato di un'interfaccia web. Per ogni tool/benchmark, è stata effettuata una clone del repository in locale. Dopodiché, sono state seguite le istruzioni indicate nei vari file README, presenti nei repository originali, per comprendere meglio come funzionassero e i prerequisiti necessari per la loro configurazione/installazione. Finito ciò, ciascun tool è stato eseguito solo sui benchmark adatti ad esso poichè, ad esempio, un tool che rileva vulnerabilità nelle Mobile application risultava inefficace su un benchmark contenente applicazioni PHP. I tool installati e lanciati con successo sono WAP, SQL-SCAN e RIPS, per i quali testimonieremo il loro funzionamento tramite Figure.

4.2.1 RQ2.1 - Usabilità/Eseguibilità dei Tool

SECUCHECK [5]. Per buildare SECUCHECK, bisogna accedere a due profili GITHUB; si parte da quello secondario, per cui è presente anche un collegamento nel file README del principale. I passi sono stati eseguiti proprio come descritti nella descrizione, ovvero:

```
git clone https://github.com/CodeShield-Security/SPDS.git
git checkout develop
mvn clean install -DskipTests
```

[12]

Quest'ultimo comando, generava un errore che riguardava l'assenza di un file jar nel sito principale di MAVEN [13], ovvero **de.fraunhofer.iem.secucheck:query:jar:1.3.0**. Il percorso completo della destinazione del jar porta, però, ad una pagina web che indica il messaggio **404 Not Found**, il che vuol dire che MAVEN non riesce a risolvere le dipendenze per un progetto presente nel repository, precisamente "commons-utility", perchè non è in grado di trovare tale artifact nel repository Maven centrale. Non si è riuscito a risolvere, nonostante svariati tentativi di ricerca su altri profili GITHUB.

SQL-SCAN [22]. Per configurare SQL-SCAN bisogna eseguire solo i passaggi descritti nel file README del suo profilo GITHUB. I passi eseguiti, quindi, sono stati:

```
pip install -r requirements.txt
python3 main.py -d product.php?id=
```

Quest'ultimo comando, serve per mostrare il comportamento del tool su un sito scritto in PHP. Osservando il suo comportamento con alcuni file presenti nei benchmark, ci si è accorto che il tool andava a scansionare siti che non avevano alcun senso con il file fornito in input. Di conseguenza, siccome l'obiettivo è diventato capire cosa non funzionasse del tool, è stato creato appositamente un file php contenente una vulnerabilità di SQL injection e, dandolo in input ad SQL-SCAN, ha sbagliato la previsione, ovvero non ha rilevato l'SQL injection oltre alla scansione dei siti non pertinenti con il file di input. Per una comprensione più a grana fine del suo comportamento, il tool è stato testato anche su un sito web presente online attualmente, ovvero Amazon, e ci si è accorti che il tool andava a scansionare tutti i siti annessi ad Amazon, tra l'altro non rilevando mai SQL injection. Si è concluso, quindi, che il tool non funziona bene per semplici file php bensì per grosse applicazioni web. Siccome non si trovano in letteratura benchmark di questo tipo, si è deciso di scartare il tool in quanto non utilizzabile per un confronto empirico con altri tool. Le due immagini 4.8 e 4.9 sono una rappresentazione di un'esecuzione di prova per testimoniare il suo comportamento funzionale.

[13] https://repo.maven.apache.org/maven2

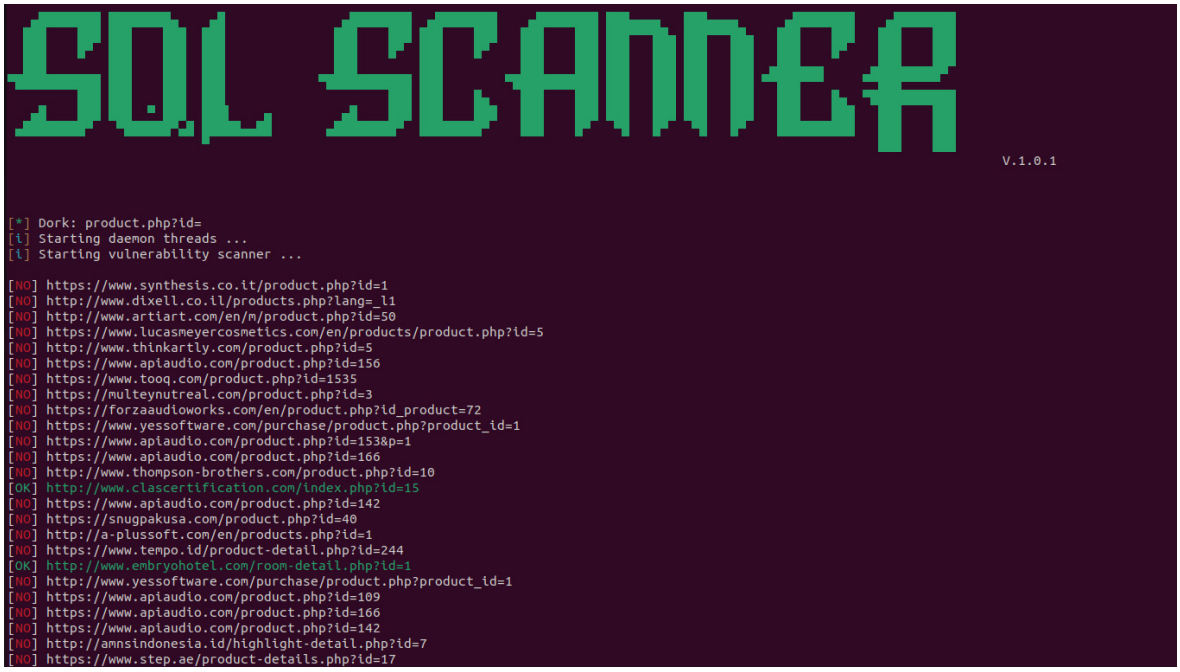

Figura 4.8: Esecuzione di SQL-SCAN tramite il comando principale.

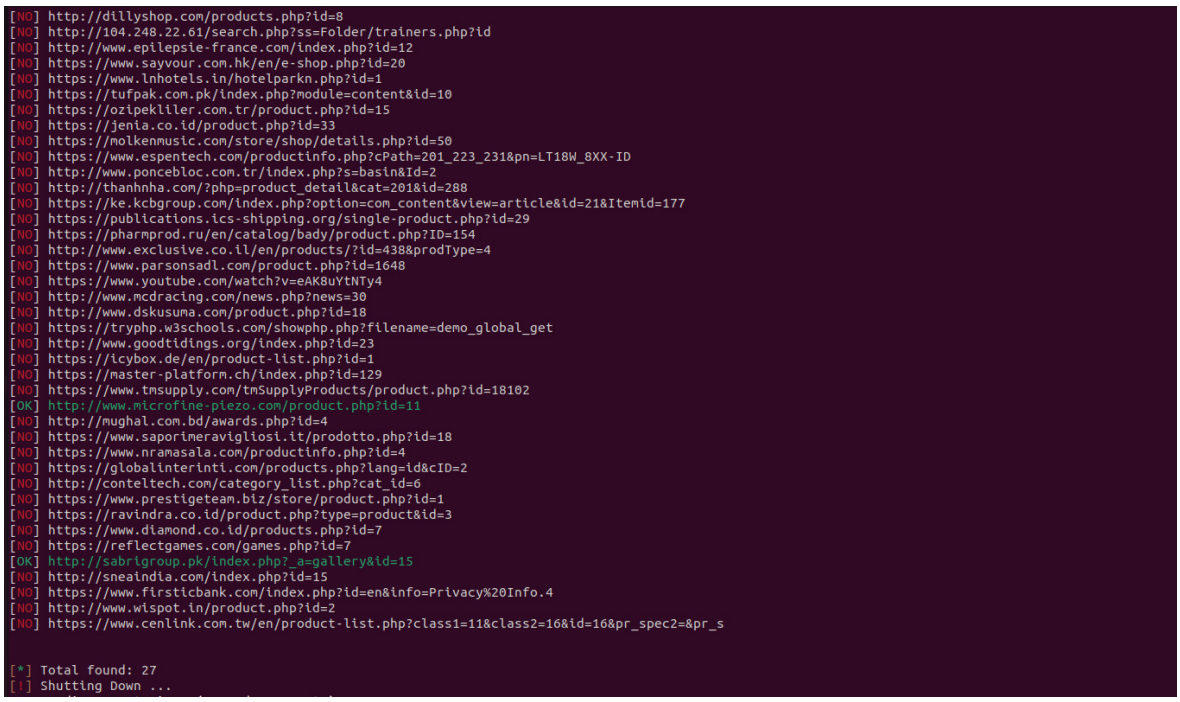

Figura 4.9: Continuo esecuzione di SQL-SCAN tramite il comando principale.

AUSERA [1]. AUSERA richiede molteplici requisiti, i quali sono stati tutti ben rispettati e installati correttamente grazie anche alla inaspettata esaustività delle istruzioni. Dopodiché, era necessario solo lanciare il comando del tool specificando tutti i

parametri richiesti, relativi alla macchina in locale. In pratica, nel repository di AUSERA, è presente una cartella contenente piccole applicazioni mobile, le quali vengono usate per scopo dimostrativo del comportamento del tool. Come per SECUCHECK, però, a causa di un errore di assenza di un file jar nel repository, si è decisi di scartare il tool. L'errore, in questione, riguardava l'assenza di **APKEngine.jar** nella cartella "engine-configuration", sotto directory di "ausera-main". Infatti, nel file README, è presente un collegamento ad un video presentazione su youtube che mostra una piccola configurazione iniziale da rispettare, per buildare correttamente AUSERA, e si nota che, nella cartella "engine-configuration", è presente tale file **APKEngine.jar**, ma non nel repository principale.

WAP [21]. WAP è tra i tool che si sono riusciti a configurare correttamente. Dopo l'installazione e l'estrazione del file zip del tool, ci si è spostati nella cartella dove è stata fatta tale operazione. Il comando eseguito per far funzionare il tool è stato:

```
./wap -sqli -a percorsoProgettoPHP
```

dove **-sqli** sta ad indicare il tipo di vulnerabilità di interesse, nel nostro caso SQL injection. Da notare che nella descrizione del comando sul sito di WAP, non sono presenti i due caratteri **./** prima di **wap**, il che sollevava un errore in quanto il sistema operativo non riusciva a trovare l'eseguibile "wap", il quale, in realtà, era presente nella corrente directory. Il motivo è che quando si esegue un comando senza specificare un percorso completo o senza indicare una directory specifica, il sistema operativo cerca l'eseguibile nelle directory elencate nel **$PATH** (variabile d'ambiente che contiene le directory in cui il sistema deve cercare eseguibili). Usando il "punto-barra", si sta dicendo esplicitamente al sistema operativo di cercare l'eseguibile nella directory corrente. Inoltre, per usare **'./'** senza dover aggiungere la directory corrente al **$PATH**, bisogna avere i permessi di esecuzione su quel file, i quali si possono assegnare utilizzando il comando **chmod** in questo modo: "chmod +x *myprogram*". Dopo aver eseguito WAP sul benchmark di riferimento (PHP-vulnerability-test-suite), non ha restituito buoni risultati. Innanzitutto, si è deciso di usare tale benchmark in quanto WAP è un tool che rileva vulnerabilità solo in PHP application. Il procedimento per automatizzare l'esecuzione di WAP sul benchmark è iniziato con la creazione di uno script Python che facesse uso del Subprocess per lanciare istruzioni da riga di

comando; questo è servito per illustrare il comportamento del tool con i file php presenti nel benchmark. Ogni volta che WAP scansiona un codice php contenuto in un file, richiede la pressione del tasto INVIO per visualizzare a video se è stata rilevato un possibile attacco SQL injection o meno; se ci si trova nel primo caso, allora WAP richiede un ulteriore pressione del tasto INVIO per consigliare, a colui che sta usando il tool, come va modificato il codice sorgente per evitare un possibile attacco SQL injection; nel secondo caso, invece, WAP risponde solo che non ha rilevato un possibile attacco SQL injection. Di conseguenza, bisognava cliccare il tasto INVIO almeno tante volte per quanti erano i file presenti nel benchmark, ovvero 9,552. Per far ciò, allora, è stato creato uno piccolo script python che simulasse la pressione del tasto INVIO all'infinito; lo si è fatto partire in concomitanza dello script python implementato inizialmente, così da automatizzare ancora di più a grana fine l'operazione. Sono stati salvati gli orari attuali sia prima che dopo l'esecuzione del tool in modo da visualizzare il suo tempo di esecuzione, in secondi, su quel benchmark. La Figura 4.10 è una rappresentazione di un'esecuzione di prova per testimoniare il suo comportamento funzionale.

```
  + Type of Analysis: SQLI
     > Summary:
        - Time of analysis: 102 ms
        - Number of vulnerabilities detected: 1
           - Real vulnerabilities: 1
           - False positives: 0
        - Number of vulnerable files: 1
        - List of vulnerable files:
            /home/angelo/Scrivania/Tesi/wap-2.1/PHP-Vulnerability-test-suite/Injection/CWE_89/unsafe/CWE_89__POST__no_sanitizing__multiple_select-inte
rpretation_simple_quote.php


Press enter to view vulnerabilities...


> > > >  File: /home/angelo/Scrivania/Tesi/wap-2.1/PHP-Vulnerability-test-suite/Injection/CWE_89/unsafe/CWE_89__POST__no_sanitizing__multiple_select-i
nterpretation_simple_quote.php < < < <
     > Information:
        - Number of Lines of Code: 64
        - It is a include file: no
        - Included files: none
        - Defined user function: none
        - Number of Vulnerabilities detected: 1
           - Real Vulnerabilities: 1
           - False positives: 0

        = = = =  Vulnerability n.: 1  = = = =
        Vulnerable code:
        45: $tainted = $_POST['UserData'];
        49: $query = "SELECT * FROM COURSE c WHERE c.id IN (SELECT idcourse FROM REGISTRATION WHERE idstudent=' $tainted ')";
        56: $res = mysql_query($query); //execution

        Corrected code:
        45: $tainted = $_POST['UserData'];
        49: $query = "SELECT * FROM COURSE c WHERE c.id IN (SELECT idcourse FROM REGISTRATION WHERE idstudent='".san_sqli(0, $tainted)."'";
        56: $res = mysql_query($query); //execution
```

Figura 4.10: Esecuzione di WAP sul file *CWE_89__POST__no_sanitizing__multiple_select-interpretation_simple_quote.php* contenuto nel benchmark *PHP-Vulnerability-Test-Suite*

TCHECKER [7]. Per configurare TCHECKER, era richiesto la navigazione su altri due profili GITHUB: il primo, configurato correttamente seguendo le istruzioni elencate nel file README, mentre il secondo richiedeva, a sua volta, la navigazione su altri profili GITHUB per poter rispettare tutti i requisiti. Tralasciando tutta la parte iniziale e partendo dal secondo profilo GITHUB, sono stati eseguiti i seguenti passaggi:

- git clone https://github.com/nikic/php-ast
- cd php-ast
- git checkout 701e853
- phpize: questo comando però richiedeva l'installazione di PHP e la configurazione delle variabili d'ambiente sulla macchina, il che non era descritto nel file README.
- ./configure
- make
- sudo make install
- aggiunta della linea \textbf{extension=ast.so} nel file php.ini ./php2ast product.php: dove product.php si trattava di una piccola porzione di codice contenente un possibile attacco SQL injection
- installazione di maven e dell'ultima versione di JDK in quanto si trattavano di prerequisiti presenti in un altro profilo github, quello relativo a Joern.
- wget https://github.com/joernio/joern/releases/latest/download/joern-install.sh
- chmod +x ./joern-install.sh
- sudo ./joern-install.sh
- joern
- si è tornati alla pagina antecedente
- git clone https://github.com/octopus-platform/joern
- gradle build

Al lancio di quest'ultimo comando, l'errore in questione è stato di compilazione ed ha compromesso l'usabilità di TCHECKER; in pratica, l'errore diceva che la versione di Java che si stava usando nella macchina (jdk20) non era compatibile con la versione per la quale era stato compilato un file java presente tra le cartelle di TCHECKER con la jdk16. Dopo spudorati tentativi, si è decisi di contattare, via email, il proprietario del tool, nonché proprietario del profilo GITHUB di TCHECKER, per chiedere suggerimenti su cosa si potesse fare per risolvere il problema. Egli ha risposto che,

effettivamente, il problema risiede nell'assenza di bytecode (file .class) nel repository. A causa di ciò, il comando per lanciare TCHECKER non può essere eseguito direttamente. Inoltre, ha espresso il suo rammarico nel comunicare che non sarà in grado di mantenere questo repository nel futuro a causa dell'importante impegno di tempo che richiede il tool.

WIRECAML [26]. Per configurare ed installare WIRECAML, bastava seguire le istruzioni descritte nel suo file Readme. Innanzitutto, erano presenti tre comandi per poter ottenere il dataset. Successivamente, era presente un altro comando per poter installare tutte le librerie necessarie affinché il tool possa risultare usabile ma, infine, non è descritto il comando per eseguirlo su qualche file php. Un piccolo paragrafo, descritto alla fine del file Readme, indica di recarsi al file di configurazione, precisamente "config.ini", in quanto ci sono dei suggerimenti per lanciare il tool in maniera personalizzata. Infatti, c'è la possibilità di scegliere il dataset su cui lanciarlo, selezionare una qualsivoglia vulnerabilità e il parametro "model". Nel file di configurazione "config.ini", però, è presente il seguente commento, prima di poter scegliere il dataset su cui lanciare WIRECAML: *Options are 'SAMATE', 'NVD' or 'both'*; ciò fa intendere che il tool è stato realizzato solo per lavorare con questi tre dataset e non è possibile testarlo sui benchmark rilevati durante la literature review. Di conseguenza, si è deciso di scartare il tool e di non ammetterlo alla fase successiva.

RIPS [17]. RIPS si è rivelato l'unico tool non dotato di interfaccia a riga di comando, bensì di un'interfaccia web. Per usarlo, era necessario, quindi, l'uso di un web server; si è usato **Wamp64**. Dopo l'installazione del web server, però, il tool non riusciva ad aprirsi. Ci si è accorti, dal file Readme, che bisognava tener installato, come requisito, l'ultima versione di PHP ma il tool è stato rilasciato nel 2016 quindi, sicuramente, non corrispondeva alla versione più recente attuale di PHP. Si è provati, allora, a modificarla passando dalla versione 8 alla 7 e il tool è partito, in localhost. Dotato di un'interfaccia semplice e abbastanza intuitiva, si è scelti il tipo di vulnerabilità (in questo caso SQL injection) e forniti in input tutti i file presenti nel benchmark. Grazie ad un ricco e corposo report e grazie alla conoscenza dell'oracolo, è stato possibile ricavare il tempo di esecuzione del tool e tutti i parametri riguardanti

veri/falsi positivi/negativi. La Figura 4.11 è una rappresentazione di un'esecuzione di prova per testimoniare il suo comportamento funzionale.

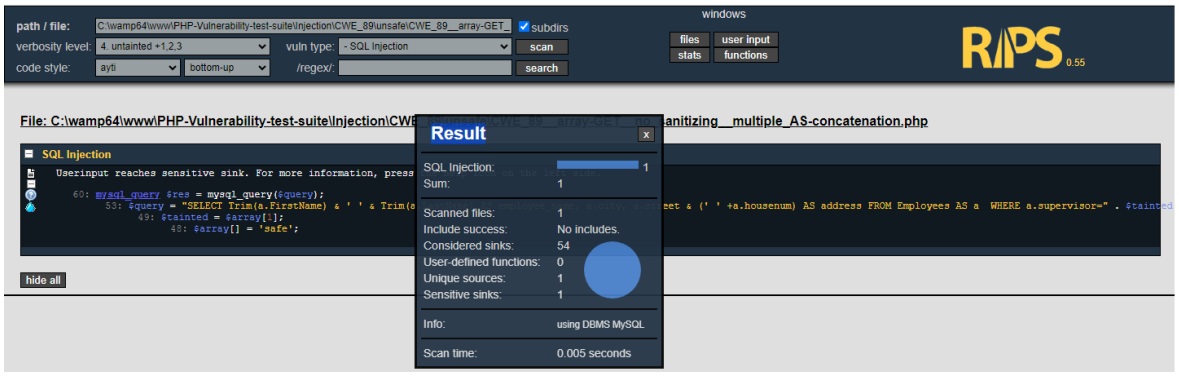

Figura 4.11: Esecuzione di RIPS sul file *CWE_89_unsafe_CWE_89__array-GET__no_sanitizing_-_multiple_AS-concatenation.php* contenuto nel benchmark *PHP-Vulnerability-Test-Suite*

4.2.2 RQ2.2 - Prestazioni dei Tool

Siccome soltanto i tool RIPS e WAP sono stati eseguiti con successo, presenteremo i valori risultanti dalle varie metriche di valutazione in merito al test sul benchmark "PHP-Vulnerability-Test-Suite". È importante notare che l'**Accuracy** generale è ragionevolmente alta per entrambi i tool, attestandosi all' 82% per WAP e 81% per RIPS. Tuttavia, una panoramica più dettagliata delle altre metriche solleva alcune preoccupazioni. La **Precisione** del 12% per WAP e 15% per RIPS indica che su tutte le istanze classificate come positive dai tool, solo una piccola percentuale è effettivamente una vulnerabilità reale e, ciò, potrebbe indicare che hanno tendenza a segnalare molti falsi positivi. La **Recall** del 14% per WAP e del 22% per RIPS suggerisce che solo una percentuale ristretta di tutte le vere vulnerabilità è stata correttamente individuata dai tool. Inoltre, l'**F1-Score** del 13% WAP e del 16% per RIPS enfatizza ulteriormente la sfida di trovare un equilibrio tra la precisione delle segnalazioni positive e la capacità di individuazione delle vere vulnerabilità. La **Matrice di Confusione** ha mostrato la distribuzione dei veri e falsi positivi e negativi per entrambi i tool. Le Tabelle 4.4 e 4.5 rappresentano la Matrice di Confusione e vanno lette in maniera tale che la prima riga rappresenta i VN e FP mentre la seconda rappresenta gli FN e VP.

	Predizione: NO	Predizione: SI	Totale
Output: NO	7,686 (89%)	954 (11%)	8,640
Output: SI	780 (86%)	132 (14%)	912
Totale	8,466	1,086	

Tabella 4.4: Matrice di Confusione di WAP.

	Predizione: NO	Predizione: SI	Totale
Output: NO	7,521 (87%)	1,119 (13%)	8,640
Output: SI	708 (78%)	204 (22%)	912
Totale	8,229	1,323	

Tabella 4.5: Matrice di Confusione di RIPS.

Va notato che i veri positivi rappresentano solo il 14% per WAP e il 22% per RIPS di tutte le vere vulnerabilità e i falsi negativi suggeriscono che molte vulnerabilità non sono state rilevate dai tool; ciò rafforza l'idea che potrebbero tendere a segnalare meno frequentemente la presenza di vulnerabilità, portando ad un eccesso di veri e falsi negativi. In altre parole, quando i tool affermano che una certa istanza non è una vulnerabilità di SQL injection, è più probabile che questa affermazione sia corretta per l'elevata percentuale dei VN.

Particolare attenzione si ripercuote sul **Tempo di esecuzione**, che si è attestato a 4279.41 secondi per WAP su 9,552 file, ovvero circa 1 ora e 18 minuti; questo indica una considerevole durata delle analisi, influenzando l'efficienza pratica dell'uso del tool. Il **Tempo di esecuzione** per RIPS, invece, varia di pochi secondi ogni volta che si prova a dare in input tutti i file del benchmark. Probabilmente, il tempo di scansione dipende dai processi attivi nella macchina: più processi sono attivi e più tempo richiede il tool, meno processi sono attivi e meno tempo impiega per scansionare il benchmark. Il tempo massimo rilevato è di 40.154 secondi su 8,640 file non vulnerabili e 3.551 secondi su 912 file vulnerabili, per un totale di 43.705 secondi sui

9,552 file del benchmark "PHP-Vulnerability-Test-Suite". Questo indica un'eccellente durata dell'analisi che influisce, positivamente, sull'usabilità del tool.

Mostriamo un esempio di un file php del benchmark, precisamente chiamato *CWE_89__POST__no_sanitizing__multiple_select-interpretation_simple_quote.php*, vulnerabile ad un attacco SQL Injection. WAP ha effettivamente indovinato:

```php
<?php

$tainted = $_POST['userData'];

//no sanitizing
$query = "SELECT * FROM COURSE c WHERE c.id IN
(SELECT idCourse FROM REGISTRATION WHERE idStudent=' $tainted ')";

$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password');
mysql_select_db('dbname');
echo "query : ". $query ."<br /><br />";

$res = mysql_query($query);

while($data = mysql_fetch_array($res)){
print_r($data);
echo "<br />";
}
mysql_close($conn);
?>
```

Questo codice è vulnerabile perchè la variabile "tainted" prende l'input dall'utente, il quale potrebbe essere un malintenzionato che inietta codice malevolo. Siccome non è presente nessun sanitizzatore, il codice è considerato vulnerabile. WAP ha risposto correttamente affermando che ha trovato una sola vulnerabilità.

Mostriamo un esempio di un file php del benchmark, precisamente chiamato *CWE_89__array-GET__no_sanitizing__multiple_AS-concatenation.php*, vulnerabile ad un attacco SQL injection. RIPS ha effettivamente indovinato:

```php
<?php

$array = array();
$array[] = 'safe' ;
$array[] = $_GET['userData'] ;
$array[] = 'safe' ;
$tainted = $array[1] ;

//no_sanitizing
```

```
$query = "SELECT Trim(a.FirstName) & ' ' & Trim(a.LastName) AS
employee_name, a.city, a.street & (' ' +a.housenum) AS address FROM
Employees AS a  WHERE a.supervisor=". $tainted . "";

//flaw
$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password'); //
Connection to the database (address, user, password)
mysql_select_db('dbname') ;
echo "query : ". $query ."<br /><br />" ;

$res = mysql_query($query); //execution

while($data =mysql_fetch_array($res)){
print_r($data) ;
echo "<br />" ;
}
mysql_close($conn);
?>
```

Questo codice è vulnerabile perchè la variabile "tainted" viene assegnata alla seconda cella dell'array che conterrà un input preso dall'utente, il quale potrebbe essere un malintenzionato che inietta codice malevolo. Siccome la variabile viene usata direttamente nella query senza passare per un sanitizzatore, il codice è considerato vulnerabile. RIPS ha risposto correttamente affermando che ha trovato una sola vulnerabilità.

Invece, un esempio per il quale c'è vulnerabilità ma sia WAP che RIPS hanno risposto scorrettamente è la seguente porzione di codice presente sempre in un file php del benchmark, precisamente chiamato *CWE_89_unserializefunc_mysql_real_escape_string_multiple_AS-interpretation.php*:

```
<?php

$string = $_POST['UserData'] ;
$tainted = unserialize($string);

$tainted = mysql_real_escape_string($tainted);

$query = "SELECT Trim(a.FirstName) & ' ' & Trim(a.LastName) AS
employee_name, a.city, a.street & (' ' +a.housenum) AS address FROM
Employees AS a  WHERE a.supervisor= $tainted ";


$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password');
mysql_select_db('dbname') ;
echo "query : ". $query ."<br /><br />" ;

$res = mysql_query($query); //execution

while($data =mysql_fetch_array($res)){
print_r($data) ;
```

```
echo "<br />" ;
}
mysql_close($conn);
?>
```

Questa porzione di codice è vulnerabile, nonostante l'uso della funzione **mysql_real_escape_string()** per sanificare i dati, perchè ci si effettua una deserializzazione di dati non fidati senza alcuna verifica. Ciò apre la porta ad attacchi di deserializzazione malevoli, noti come attacchi di "deserialize exploitation". Gli attaccanti potrebbero inserire dati dannosi nel payload serializzato, che potrebbero compromettere l'applicazione. Inoltre, la query usata non è parametrica; sebbene si sia usato la funzione **mysql_real_escape_string()**, non risulta sicura la concatenazione della variabile "tainted" direttamente alla query. Questo approccio non è sufficiente per prevenire attacchi di SQL injection, specialmente quando si tratta di tipi di dati complessi come stringhe contenenti caratteri speciali.

Da notare che WAP, per il file vulnerabile *CWE_89_unserializefunc_mysql_real_escape_string_multiple_AS-interpretation.php*, ha indovinato mentre RIPS ha risposto scorrettamente ed è per questo che abbiamo usato un altro file del benchmark per mostrare l'individuazione positiva di un'istanza da parte di RIPS. Approfondire il motivo per il quale RIPS ha sbagliato su tale file, non fa parte del lavoro di tesi. Tuttavia, c'è da dire che il report mostrato a video da RIPS asserisce che il codice Object-Oriented non è supportato dal tool; ciò potrebbe significare che la colpa non è, completamente, di RIPS per alcuni dei file che non riesce a rispondere correttamente, bensì dalla sua natura che non supporta l'Object-Oriented. In conclusione, mentre WAP e RIPS dimostrano una buona accuratezza generale, è evidente che ci sono molti spazi di miglioramento in termini di precisione, recall e F1-Score. Questo può avere un impatto significativo sulla fiducia nell'output dei tool e può richiedere ulteriori sforzi per verificare manualmente ciascuna segnalazione.

CAPITOLO 5

Conclusioni

I tool si sono mostrati tecnologicamente immaturi; purtroppo, la maggior parte di loro ha dimostrato di essere difficoltosa da installare a causa della documentazione carente o di numerosi prerequisiti richiesti. Ciò ha sicuramente limitato la capacità di eseguire test approfonditi su tali tool. Gli unici due che sono stati selezionati sono stati WAP e RIPS, i quali, testandoli sullo stesso benchmark di riferimento, non hanno prodotto buonissimi risultati soprattutto dal punto di vista di parametri come precision, recall, ecc. Il confronto empirico è possibile effettuarlo in quanto si trattano di due tool che sono stati testati sullo stesso benchmark. I risultati delle valutazioni hanno dimostrato che RIPS è dotato di prestazioni leggermente superiori rispetto a WAP in termini di Precision, Recall ed F1-Score, indicando una maggiore capacità di identificare correttamente le istanze di SQL injection, sia positive che negative. Inoltre, è risultato nettamente sovrastante dal punto di vista del tempo di esecuzione. Pertanto, RIPS si è dimostrato il vincitore in termini di prestazioni. Si può concludere, però, che è molto importante avere una documentazione chiara quando si decide di rendere pubblico un qualsiasi lavoro effettuato e, soprattutto, di ridurre o semplificare il processo di installazione e configurazione del tool. Si riconoscono le limitazioni dello studio, in particolare il fatto di aver avuto accesso a solo due tool funzionanti ma dall'altra parte è stato appreso molto da questa esperienza a causa degli ostacoli

incontrati e le difficoltà dell'ottenere risultati validi. Si suggerisce, assolutamente, di arricchire le documentazioni e assottigliare i passi di installazione da eseguire e, come futura implementazione, di realizzare un classificatore di Machine Learning per facilitare la scelta del miglior tool attraverso l'analisi delle metriche di valutazione ottenute.

Bibliografia

[1] Sen Chen, Yuxin Zhang, Lingling Fan, Jiaming Li, Yang Liu, "Ausera: Automated security vulnerability detection for android apps," Conference ASE: Automated Software Engineering, 2022. (Citato alle pagine iii, 38, 39, 40 e 54)

[2] Abbas Naderi-Afooshteh, Anh Nguyen-Tuong, Mandana Bagheri-Marzijarani, Jason D. Hiser, Jack W. Davidson, "Joza: Hybrid taint inference for defeating web application sql injection attacks," IEEE/IFIP International Conference on Dependable Systems and Networks, 2015. (Citato alle pagine iii, 36, 38, 40 e 41)

[3] Yan Huang1, Chaohui He2, Chenglong He1, Chaoyong Wang1, "Effective dynamic taint analysis of java web applications," International Conference on Bigdata Blockchain and Economy Management (ICBBEM), 2022. (Citato alle pagine iii, 36, 38, 43 e 44)

[4] William G.J. Halfond, Alessandro Orso, Panagiotis Manolios, "Using positive tainting and syntax-aware evaluation to counter sql injection attacks," Conference FSE: Foundations of Software Engineering, 2006. (Citato alle pagine iii, 36, 38, 44 e 45)

[5] Goran Piskachev, Ranjith Krishnamurthy, Eric Bodden, "Secucheck: Engineering configurable taint analysis for software developers," International Working

Conference on Source Code Analysis and Manipulation (SCAM), 2021. (Citato alle pagine iii, 36, 38, 45, 46, 51 e 52)

[6] Ricardo Morgado, Iberia Medeiros, Nuno Neves, "Towards web application security by automated code correction," International Conference on Evaluation of Novel Approaches to Software Engineering, 2020. (Citato alle pagine iii, 36, 38, 46 e 47)

[7] Changhua Luo, Penghui Li, Wei Meng, "Tchecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in php applications," CCS: ACM SIGSAC Conference on Computer and Communications Security, 2022. (Citato alle pagine iii, 36, 38, 49 e 57)

[8] Abdullah Mujawib Alashjee, Salahaldeen Duraibi, Jia Song, "Dynamic taint analysis tool: A review," International Journal of Computer Science and Security (IJCSS), vol. 13, 2019. (Citato a pagina 2)

[9] Junbin Zhang, Yingying Wang, Lina Qiu, Julia Rubin, "Analyzing android taint analysis tools: Flowdroid, amandroid, and droidsafe," IEEE Transactions on Software Engineering (Journal), 2021. (Citato alle pagine 2 e 27)

[10] Achmad Fahrurrozi Maskur, Yudistira Dwi Wardhana Asnar, "Static code analysis tools with the taint analysis method for detecting web application vulnerability," International Conference on Data and Software Engineering (ICoDSE), 2019. (Citato a pagina 19)

[11] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, Patrick McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," ACM Transactions on Information and System Security, 2014. (Citato alle pagine 20 e 25)

[12] Dan Boxler, Kristen R. Walcott, "Static taint analysis tools to detect information flows," Conference Software Engineering Research and Practice (SERP), 2018. (Citato a pagina 27)

[13] Atefeh Tajpour, Maslin Masrom, Mohammad Zaman Heydari, Suhaimi Ibrahim, "Sql injection detection and prevention tools assessment," International Conference on Computer Science and Information Technology, 2010. (Citato alle pagine 27 e 36)

[14] Mahmoud Baklizi, Issa Atoum, Nibras Abdullah, Ola A. Al-Wesabi, Ahmed Ali Otoom, Mohammad Al-Sheikh Hasan, "A technical review of sql injection tools and methods:a case study of sqlmap," International Journal ofINTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING, 2022. (Citato a pagina 27)

[15] Ehsan Edalat, Babak Sadeghiyan, Fatemeh Ghassemi, "Considroid: A concolic-based tool for detecting sql injection vulnerability in android apps," IET Research Journals, 2019. (Citato alle pagine 36, 38, 47 e 51)

[16] Vivek Haldar, Deepak Chandra, Michael Franz, "Dynamic taint propagation for java," Annual Computer Security Applications Conference (ACSAC), 2005. (Citato a pagina 36)

[17] Johannes Dahse, Thorsten Holz, "Simulation of built-in php features for precise static code analysis," Internet Society, ISBN, 2014. (Citato alle pagine 36, 38, 50 e 58)

[18] Saurabh Doshi, Ashwini Padale, Chaitali Parekh, Devata Anekar, "Sidp-sql injection detector and preventer," International Journal of Computer Application (IJCA), 2012. (Citato alle pagine 36, 38 e 40)

[19] Elisa Burato, Pietro Ferrara, Fausto Spoto, "Security analysis of the owasp benchmark with julia," In Proceedings of the First Italian Conference on Cybersecurity (ITASEC17), 2017. (Citato a pagina 36)

[20] Fauso Spoto, Elisa Burato, Micheal D.Ernst, Pietro Ferrara, Alberto Lovato, Damiano Macedonio, Ciprian Spiridon, "Static identification of injection attacks in java," ACM Transactions on Programming Languages and Systems, 2019. (Citato a pagina 36)

[21] Jeevan Kumar, Srikanth Reddy, P.Mohan, P.Sravan Kumar Reddy, "Web application recognition and statistical analysis to reduce vulnerabilities in data mining," Industrial Engineering Journal, 2021. (Citato alle pagine 36, 38, 42 e 55)

[22] Narottam Chaubey, Sumit Sharma, "Sqlscan: A framework to check web application vulnerability," International Institute for Science, Technology and Education (IISTE), 2016. (Citato alle pagine 36, 38, 41 e 53)

[23] Bertrand Stivalet, Elizabeth Fong, "Large scale generation of complex and faulty php test cases," IEEE International Conference on Software Testing, Verification and Validation (ICST), 2016. (Citato alle pagine 36, 51 e 52)

[24] Dan Boxler, Kristen Walcott, "Static taint analysis tools to detect information flows," Conference Software Engineering Research and Practice, 2018. (Citato alle pagine 36 e 51)

[25] M. Martin, B. Livshits, M. S. Lam, "Finding application errors and security flaws using pql: A program query language." ACM SIGPLAN Notices, 2005. (Citato alle pagine 36, 38 e 39)

[26] Jorrit Kronjee, Arjen Hommersom, Harald Vranken, "Discovering software vulnerabilities using data-flow analysis and machine learning," Conference ARES: Availability, Reliability and Security, 2018. (Citato alle pagine 38, 48 e 58)

Ringraziamenti

Desidero ringraziare, in primis, il professore Andrea De Lucia e il dottor Emanuele Iannone per la loro guida preziosa e il loro continuo supporto durante la realizzazione del lavoro di tesi. I loro consigli puntuali e le loro critiche costruttive hanno contribuito all'arricchimento del mio percorso accademico. Sono grato per la loro fiducia e pazienza mostrata nei miei confronti e per l'opportunità di lavorare su questo affascinante progetto di ricerca.

Ringrazio la mia splendida famiglia per avermi supportato e sopportato in questo lungo percorso, soprattutto nei momenti in cui pensavo di non farcela e di non riuscire a raggiungere l'obiettivo prefissatomi. Ringrazio mia mamma che ha sempre dimostrato un interesse tangibile per gli argomenti dei corsi; a giudicare dalla sua espressione, sembrava quasi convinta di aver capito tutto ciò che studiavo, offrendosi come volontaria nel diventare la mia tutor personale. Ringrazio mio padre, il campione indiscusso nel Concorso Nazionale di Battute a Momenti Inopportuni. Grazie per aver dimostrato incredibile maestria nel farmi ridere anche quando avrei preferito lanciare i libri attraverso la finestra. Insomma, hai reso più leggeri quei momenti turbolenti. A proposito, ringrazio calorosamente mio fratello Cristian, il quale ha contribuito alla grande durante i momenti turbolenti, concedendomi l'onore di essere il suo professore a domicilio; spero che le lezioni private siano state all'altezza delle tue aspettative, anche senza lo stipendio da insegnante.

Un caloroso ringraziamento va alla mia squadra di colleghi e amici universitari, i veri eroi della mia carriera accademica: Dario, Francesco, Nicola, Savino, Simone, Valerio e Vito. Siamo stati un vero team, un'élite dell'umore e della condivisione di appunti. Un gruppo coeso pronto ad aiutarsi uno con l'altro nei momenti di difficoltà e nel realizzare svariati progetti per il superamento degli esami. Insieme, abbiamo trasformato l'arte dello studio in un'epica avventura. Grazie per le risate che ci hanno tenuto svegli durante le maratone di studio; senza di voi, la laurea sarebbe stata solo una parola nel vocabolario, ma ora è una medaglia sul petto di quasi tutti noi. Mi auguro che il prossimo capitolo delle nostre vite sia pieno di successi e soddisfazioni personali!

Ringrazio le uniche persone che si sono rivelati veri e fedeli amici dopo il percorso di scuola superiore: Paolo, Davide ed Antonio. Siamo, insieme ad Eljon, il gruppo di uscite serali (e non) più bello che potessi avere. Quando sto con voi, so già che il divertimento è assicurato. Non posso esimermi nell'augurarvi un futuro luminoso e che il percorso di vita che abbiamo scelto, ci porti a realizzare i nostri sogni più audaci, superando le nostre aspettative.

Un caloroso ringraziamento va al gruppo ZoomClick Eventi che è stato il mio rifugio e la valvola di sfogo dopo una settimana intensa di studio. Grazie per aver trasformato le sere in una vera terapia musicale. Ogni parola scambiata, ogni battuta o risata condivisa e ogni ritmo musicale è stato un momento di pausa rigenerante, consentendomi di ricaricare le energie e affrontare la settimana successiva con rinnovato entusiasmo. Grazie di cuore ad ognuno di voi per aver reso questo percorso speciale e memorabile.

Un affettuoso ringraziamento va ai miei nonni, veri angeli custodi del mio percorso accademico. Ringrazio mio nonno Angelo, un uomo saggio e generoso che mi ha sempre spronato a portare alto il nostro nome e cognome. Le parole di incoraggiamento e la fiducia che hai riposto in me sono state la spinta determinante per ogni esame da sostenere. Grazie per avermi trasmesso il senso di orgoglio e per

avermi insegnato che il successo richiede impegno e dedizione; inoltre, grazie per aver trasformato ogni esame in un vero e proprio lavoro! Ogni voto che ho ottenuto ha rappresentato una sorta di 'salario', perché sapevo che la mazzetta dipendeva da esso. A parte gli scherzi, spero che questo successo contribuisca a portare gloria al nostro nome, proprio come hai sempre auspicato. Ci sono persone speciali che segnano il percorso della nostra vita in modo indelebile e mio nonno Gennaro è sicuramente una di queste. Anche se il destino ha fatto in modo che non fosse presente per tutta la durata del mio percorso accademico, la sua gioia immensa per ogni voto alto che riuscivo a guadagnare ha alimentato la mia ambizione a raggiungere tale traguardo. Non posso dimenticare il tuo umorismo contagioso, che forse hai trasmesso anche a me, che ha reso il mio cammino accademico più leggero e spensierato. Eri lì pronto con le tue battute sagaci a regalarmi un sorriso e, anche se non posso sentire la tua risata in questo momento, so che dall'alto, con il tuo solito stile, mi diresti qualcosa di divertente per celebrare questa vittoria. Sono sicuro che da lassù non hai mai smesso di essere il mio più grande sostenitore, esultando per tutti i voti che ho ricevuto e che riceverò; sono convinto che il tuo amore e la tua gioia accompagneranno ogni passo del mio cammino, spronandomi ancor di più a raggiungere obiettivi sempre più importanti. Questa Laurea è, senz'altro, dedicata a te.

Come ultimo ringraziamento, ma non per importanza, non posso che ringraziare me stesso per il coraggio, la perseveranza e la determinazione che ho dimostrato nel mio percorso. Nonostante la vita mi abbia posto di fronte numerosi ostacoli in passato, ho continuato a mantenere il mio sguardo fisso sugli obiettivi e se sono arrivato a questo traguardo, non posso fare altro che ringraziare me stesso, ovvero il miglior alleato in quest'avventura. Ho imparato che la fiducia in se stessi è una delle chiavi più importanti per il successo e che la capacità di fissare delle priorità e il precludersi di un'uscita in più, è fondamentale per evitare distrazioni, mantenendo sempre la visione chiara del traguardo e consentendomi di concentrare le energie dove erano necessarie. Infine, ringrazio tutte le persone che hanno incrociato il mio cammino, sia coloro che mi hanno dimostrato amore e sostegno, facendomi sentire amato e supportato, sia coloro che non credevano in me o che manifestavano ostilità. Ho trasformato quest'ultima in motivazione e determinazione, utilizzandola come

carburante per dimostrare il mio valore; è stata una conferma che stavo intraprendendo la strada giusta e che stavo raggiungendo traguardi degni di nota. Il viaggio continua.