סמינר בהנדסת תוכנה

SQL Injection - Defining Code-injection Attacks

Donald Ray Jay Ligatti | January 2012

קישור לGitHub של הפרויקט שחזור -https:/github.com/tamarhayat/SQLInject

א- מבוא

תקציר

פרויקט זה עוסק ביצירת זיהוי פורמלי איזה קלט נחשב לsql injection באמצעות ניתוח תחבירי של שאילתות sql תוך שימוש בANTLR ו-Python. הגישה של המאמר מבוססת על שיטת תיוג של קלטים מסוכנים על ידי כך שמעבירים אותם כמה מבחנים שבסופם מחליטים האם איפשהו בשאילתה לאחר קבלת הקלט נוצר קוד או שזה שאילתה לגיטימית.

האלגוריתם נוצר על ידי מציאת בעיות בפתרונות קיימים, ניתוח מה הבעיה הכללית ויישום באלגוריתם שלנו.

אנחנו המרנו את האלגוריתם לpython והרצנו עליו את הבדיקות שאלגוריתמים קיימים לא מזהים והאלגוריתם עומד בכל הבדיקות.

וכך יצרנו אלגוריתם שמגן מפני מתקפות הזרקת קוד sql.

רקע

SQL Injection (הזרקת SQL) היא אחת מהמתקפות הידועות, הנפוצות והמסוכנות ביותר בעולם אבטחת המידע, היא מתרחשת כאשר תוקף מצליח "להזריק" פקודות SQL זדוניות לתוך שאילתות הנשלחות למסד הנתונים של יישום אינטרנטי. מתקפה זו נובעת לרוב מחוסר טיפול תקין בקלט מהמשתמש, שמוזן ישירות לתוך שאילתות ה-SQL ללא סינון, וללא שימוש במנגנוני הגנה כמו שאילתות מוכנות מראש (Prepared Statements).

במילים פשוטות, כאשר מערכת מקבלת קלט מהמשתמש כמו שם משתמש, סיסמה או מזהה פריט ומשתמשת בו כדי לבנות שאילתת SQL כחלק מהקוד, ישנה סכנה שהתוקף יכניס קוד זדוני במקום הנתון המצופה. לדוגמה, במקום להזין שם משתמש רגיל, התוקף עשוי להזין משהו כמו:

' ' OR 1=1 --'

כאשר קלט כזה משולב ישירות בשאילתת SQL, התוצאה עלולה להיות ביצוע שאילתה תקפה שהופכת את תנאי האימות לבלתי תקף כלומר הקלט יעקוף את כל תנאי האימות, וייתכן שהתוקף יקבל גישה למערכת מבלי להזין סיסמה תקינה כלל.

השלכות של מתקפת SQL Injection עלולות להיות הרסניות. הן עשויות לכלול:

- חשיפת מידע רגיש (כמו פרטי משתמשים, סיסמאות, מידע פיננסי ועוד).
- שינוי או מחיקה של נתונים.
- גישה לחשבונות מנהלים.

- הרצת פקודות מסוכנות על שרת מסד הנתונים.
- השתלטות על השרת כולו.

מתקפות אלו אפשריות כמעט בכל מערכת שמבוססת על אינטראקציה עם מסדי נתונים כולל מערכות ניהול תוכן (CMS), אתרי מסחר אלקטרוני, מערכות הרשמה, מערכות ניהול משתמשים ועוד.

הגנה מפני SQL Injection מתחילה בטיפול נכון בקלט מהמשתמש: יש להשתמש בשאילתות מוכנות מראש (Prepared Statements) עם פרמטרים, לנקות ולסנן קלט, ולהגביל את הרשאות המשתמשים במסד הנתונים כך שגם אם תהיה פריצה, הנזק הפוטנציאלי יהיה מוגבל.

לסיכום SQL Injection ,מדגיש את החשיבות הקריטית של פיתוח מאובטח והתייחסות לכל קלט מהמשתמש כאל קלט לא אמין שדורש טיפול מוקפד. הגנה נכונה יכולה למנוע פרצות חמורות ולשמור על שלמות המידע והמערכת כולה.

מהי הבעיה שפותרים הכותבים?

הכותבים מבקשים לפתור את הבעיה של חוסר ההגדרה המדויקת והאחידה למתקפות מסוג **Sql Injection**. אומנם קיימים פתרונות חלקיים למתקפה אבל הבעיה היא שאין הגדרה מוסכמת וברורה שמסבירה מתי מתקפה אכן נחשבת להזרקת קוד, ומה מבדיל אותה מהתנהגות רגילה של מערכת בתגובה לקלט נורמטיבי. חוסר הדיוק הזה מקשה על חוקרי אבטחת מידע, מפתחי מערכות וכלים אוטומטיים לזהות מתקפות להוכיח את קיומן או לבנות הגנות יעילות. לפיכך, הכותבים מציעים מסגרת פורמלית לניתוח מתקפות מהסוג הזה, שמבוססת על זיהוי נכון של המיקום הבעייתי ובדיקת השייכות למחרוזות חוקיות.

מדוע הבעיה מעניינת?

הבעיה שמעלים הכותבים מעניינת במיוחד מכיוון שהיא נוגעת בלב העיסוק באבטחת תוכנה מודרנית. מתקפות Code Injection הן מהנפוצות והמסוכנות ביותר בעולם אבטחת המידע, והן משפיעות על אינספור יישומים החל מאתרים פשוטים ועד למערכות קריטיות. הבהרת המושג מאפשרת ניתוח שיטתי של סוגי המתקפות, זיהוי מכנה משותף ביניהן, והנחת יסוד לתכנון מנגנוני אבטחה חזקים יותר.

ב- שיטות ואלגוריתמים

כמו שאמרנו, במאמר מוצגת שיטה לגילוי פגיעויות מסוג SQL Injection באמצעות ניתוח תחבירי של שאילתות SQL . השיטה מתמקדת בזיהוי ביטויים בתוך השאילתה אשר מהווים **קוד פעיל (code)**, ומתוך כך מסיקה האם מדובר בהזרקת קוד.

הגדרת Inject

ביטוי ייחשב כ inject-אם הוא מתקיים בו שני תנאים:

1. הוא מוכר כ־ **taint** (כלומר מקורו בקלט חיצוני בלתי מבוקר)
2. הוא מזוהה כ־ **code** (כלומר, כחלק פעיל במבנה התחבירי של השאילתה)

בעבודה זו אנו מניחות שהחלק המסומן כ־ taint ידוע מראש, ומתמקדות בזיהוי האם ביטוי כלשהו שנתון שהוא taint בשאילתה, הוא גם code.

הגדרת Code

כדי לקבוע אם ביטוי exp מתוך השאילתה p נחשב כ־code, עוברים על כל התווים בו (למשל עבור כל אינדקס i בתוך exp) ומבצעים בדיקה של code(p, i).

- הפונקציה code(p, i) תחזיר **true** אם ורק אם כל **תת-המחרוזות** של p אשר כוללות את התו i, אינן נחשבות לערכים לא-קודיים (**NCV**) .
- כלומר, מספיק שתת-מחרוזת אחת מתוך כל האפשריות שמכילות את iתהיה **NCV**, ואז code(p, i) יחזיר **false**.

במילים אחרות: אם יש אפילו תת-מחרוזת אחת סביב התו שנראית כמו ערך ולא כמו קוד (למשל מחרוזת או מספר), אז i לא מוגדר כקוד.

הגדרת NCV (Non-Code Value)

הבדיקה NCV(p, l, h) מחזירה **true** אם תת-המחרוזת p[l..h] עומדת בתנאים הבאים:

1. **Val (p, l, h) :** הקטע p[l..h] שייך לערך תחבירי (כמו מחרוזת, מספר, קבוע לוגי).
2. **FV(p, l, h) = Ø**: אין בתוך הקטע הזה משתנים חופשיים.

רק אם שני התנאים הנ"ל מתקיימים, הקטע נחשב ל־ **NCV**, כלומר אינו קוד אלא ערך פסיבי.

הגדרת VAL

Val(p, l, h) מחזיר **true** אם קיים מבנה תחבירי קצר ובר תוקף שמכיל את כל התווים בין i ל־h, והוא מוגדר כערך (value) , לדוגמה:

- מחרוזת תווים- כגון 'hello'
- מספרים - כגון 42, 3.14
- קבועים לוגיים- כמו TRUE, FALSE, NULL

אם הקטע הוא חלק מביטוי גדול יותר (כגון משתנה, פונקציה או ביטוי מתמטי), הפונקציה תחזיר false.

הגדרת FV – Free Variables

FV(p, l, h) מחזירה את קבוצת **המשתנים החופשיים** בתוך המבנה התחבירי הקטן ביותר שמכיל את התווים בין l ל h. משתנים חופשיים הם מזהים (identifiers) כמו x, id, username, שאינם מילים שמורות בשפה ואינם מוגדרים בערך תחבירי סגור.

סיכום המהלך

השיטה מבצעת את השלבים הבאים:

1. מקבלת ביטוי שמוגדר כ־ taint בתוך שאילתה.
2. בודקת האם הביטוי הוא גם code באמצעות בדיקת code(p, i) לכל תו בו.
3. אם כן – מדובר בהזרקת קוד(SQL Injection) .

שיטה זו מתבססת על ניתוח תחבירי של שאילתות SQL , שמטרתה לזהות מקרים של SQL Injection על ידי איתור קטעים בתוך השאילתה אשר מוגדרים כ־ **taint** וגם כ־**code**.

כאמור, הניתוח אינו מבצע איתור עצמאי של מקורות taint, אלא מקבל מראש את הקטע החשוד (taint) עבור כל שאילתה, ומטרתו לבדוק האם קטע זה מהווה גם קוד פעיל (code), מה שמעיד על הזרקת קוד פוטנציאלית.

שיטות חלופיות

קיימות שיטות רבות לזיהוי SQL Injection, ביניהן:

- **ניתוח סטטי מבוסס זרימת מידע – (taint analysis)** מזהה את מסלול הנתונים מקלט המשתמש ועד להרצת השאילתה, תוך סימון של taint propagation .
- **למידת מכונה** –מודלים המסווגים שאילתות על בסיס מאפיינים תחביריים או מבניים.
- **חתימות ורגולריים** –זיהוי דפוסים מוכרים להזרקות (כמו ' OR 1=1).
- **כלים בזמן ריצה** (כמו – SQLMap) ניסיון להזריק ולזהות תגובות חריגות.

היתרון בשיטה של המאמר הוא ההתמקדות בהקשר התחבירי של ה־ taint ולא רק בצורה החיצונית או המקור שלו, מה שמאפשר דיוק גבוה יותר באיתור מקרים אמיתיים של הזרקה.

מקור הקוד

הקוד שפותח לצורך הבדיקות מבוסס על המובא והמתואר במאמר ומומש באופן עצמאי בשפת Python. במאמר מוזכר רק פסאודו-קוד רלוונטי או תיאור במילים.
הקלט לקוד הוא קובץ המכיל את שאילתות הבדיקה.

ג - הנתונים

לצורך ניסוי ובחינת השיטה, נעשה שימוש בקובץ נתונים הכולל **11 שאילתות SQL** שמופיעות במאמר המקורי. מדובר בשאילתות שעל ידיהן המאמר משווה בינו לבין תוצאות של מאמרים אחרים, ולכן המטרה היא להגיע לתוצאות זהות לתוצאות המאמר.

	1	2	3	4	5	6	7	8	9	10	11
This paper	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes	No
SQLCHECK [34]	Yes	No	No	Yes	No	No	No	No	No	No	No
CANDID [3]	Yes	Yes	Yes	No	No	No	Yes	No	No	No	Yes
WASP [8] and Nguyen-Tuong et. al. [25]	Yes	Yes	Yes	No	No	No	No	No	No	No	Yes
Xu et. al. [38]	Yes	Yes	Yes	No	No	No	No	No	No	No	No

Figure 2. A comparison of definitions for partitioning code and noncode. Column numbers refer to the example output programs enumerated in Section 3.3, row names indicate partitioning techniques, and cells specify whether any of the underlined symbols are considered code.

לכל שאילתה נלווה הביטוי שנכנס כקלט ולכן נחשב כ־ **taint**, ועליו נערכת הבדיקה האם הוא מהווה גם קוד.

מבנה הנתונים

הקובץ כולל עבור כל דוגמה:

- שורת שאילתה מלאה.
- שורת taint נפרדת, המייצגת את הקלט החיצוני שחשוד כהזרקה.

לדוגמה:

```
SELECT balance FROM acct WHERE password=exit()

exit()
```

הדוגמאות של המאמר כנראה נבחרו בצורה שבה הן יכסו מגוון תרחישים:

- קלטים מסוג מספרים שלמים / מחרוזות / ערכים לוגיים.
- שאילתות עם תנאים פשוטים (WHERE name='x') לעומת תנאים מורכבים.
- שאילתות עם קלט תקני אך מסוכן.
- שאילתות שגויות תחבירית לצורך בדיקת עמידות המערכת.

מטרת הגיוון הייתה לוודא שהמערכת מזהה בצורה נכונה לא רק קלטים "ברורים" אלא גם מקרים גבוליים ומטעים.

הבדיקה בוצעה על ידי סקריפט אוטומטי שרץ על הקובץ input.txt ויוצר פלט מתועד ב-output.txt.

כמות הדוגמאות

- סך הכול קיימות 11 שאילתות בדאטאסט הנוכחי.
- מתוכן, חלק מה־ taint מזוהה כ־ code והשאילתה תסווג כ־ inject, ואחרות אינן מזוהות כך.
- כתוספת על המאמר, הוספנו קובץ של 50 דוגמאות מסווגות על מנת לאמת מולן את יעילות האלגוריתם.

פעולות עיבוד וניקוי

- כל שאילתה עוברת תהליך ניתוח תחבירי(Parsing) .
- הטקסט מחולק לרשימת תווים, עליהם מוחל האלגוריתם לבדיקת code .
- אין צורך בניקוי או טרנספורמציה של הנתונים שכן הם סינתטיים ומובנים היטב מראש.

תתי-קבוצות

- לא בוצעה חלוקה לאימון ובדיקה, כיוון שלא מדובר באימון מודל למידת מכונה אלא באלגוריתם דטרמיניסטי.

מקור הנתונים

- הנתונים הראשוניים נלקחו מתוך מאמר המחקר
- בינה מלאכותית שיצרה קובץ עם 50 דוגמאות מגוון ומתאים עפ"י אתרים שונים.

כל שאילתה נבדקת תחבירית באמצעות ANTLR4 וה-grammar של MySQL, ומנותחת לפי הפונקציות:
Code(p, i) -ו ,Val(p, l, h), FV(p, l, h), NCV(p, l, h).

כרגע לא נעשה שימוש ב Wayback Machine-או בארכיונים חיצוניים, אך ניתן בעתיד להשתמש בו לצורך השוואה לדוגמאות היסטוריות של התקפות SQL Injection (למשל מתקפות אמיתיות שתועדו ונמחקו מאתרים).

ד - תוצאות

לצורך שחזור תוצאות המאמר ובחינת האלגוריתם שהוצע בו, יישמנו את הבדיקות בעזרת קוד תוכנה שהשתמש ב־ ANTLR (פארסר לשפת SQL). תחילה ווידאנו כי הפארסר סורק ומאמת תקינות תחבירית של השאילתה, ולאחר מכן שילבנו בו את הלוגיקה הנוספת לאיתור ביטויים החשודים כ־ SQL Injection, בהתאם לשיטה שתיארנו קודם.

<u>סטים של נתונים שנבדקו:</u>

בדקנו שני קבצים:

- קובץ ראשון: 11 שאילתות מתוך המאמר שעליהן הוא מסתמך על מנת להוכיח את צדקתו.
- קובץ שני: קובץ נוסף שהכיל 50 דוגמאות חדשות לשאילתות SQL שחלקן מכילות ניסיון להזרקה, וחלקן שאילתות לגיטימיות עם קלטים חוקיים.

קריטריון לסיווג השאילתה-

עבור כל שאילתה, האלגוריתם קובע:

- האם ה־ taint נחשב גם כ־code.
- אם כן – השאילתה מוגדרת כ־ inject (כלומר חשודה כהזרקת קוד)
- אחרת – השאילתה נחשבת תקינה.

בס"ד

טבלת התוצאות

#	query	taint	Inject?
1	SELECT balance FROM acct WHERE password='' OR 1=1--'	' OR 1=1 --	yes
2	SELECT balance FROM acct WHERE pin= exit()	exit()	yes
3	…WHERE flag=1000>GLOBAL	1000>GLOBAL	yes
4	SELECT * FROM properties WHERE filename='f.e'	f.e	no
5	…pin=exit()	exit()	yes
6	…pin=aaaaa()	aaaa()	yes
7	SELECT * FROM t WHERE flag=TRUE	TRUE	no
8	SELECT * FROM t WHERE flag=aaaa	aaaa	yes
9	SELECT * FROM t WHERE password=password	password (השני)	yes
10	CREATE TABLE t (name CHAR(40))	40	yes
11	SELECT * FROM t WHERE name='x'	'x'	no

הסבר:

- בטבלה מוצגות כל השאילתות מהמאמר.
- העמודה "Inject" נגזרת אוטומטית משילוב של taint+code .
- **ניתן לראות שהתוצאות יצאו אחד לאחד כמו תוצאות המאמר שהראנו בטבלה לעיל.**

מצורפות תמונות חלקיות מקבצי הקלט הנוספים שהרצנו, יחד עם קבצי הפלט:

דוגמה לקלט מהקובץ:

```
SELECT * FROM t WHERE created_at > NOW();
NOW()
SELECT * FROM t WHERE created_at > '2025-01-01';
2025-01-01
UPDATE users SET role='admin' WHERE id=1;
1
UPDATE users SET role='admin' WHERE id=1 OR 1=1;
1 OR 1=1
```

הפלט:

```
=== Test 45 ===
Query: SELECT * FROM t WHERE created_at > NOW();
Taint: NOW()
TAINT position: [35:39]
❌ The SQL query is invalid. {syntax is valid but taint contains code}

=== Test 46 ===
Query: SELECT * FROM t WHERE created_at > '2025-01-01';
Taint: 2025-01-01
TAINT position: [36:45]
✅ The SQL query is valid.

=== Test 47 ===
Query: UPDATE users SET role='admin' WHERE id=1;
Taint: 1
TAINT position: [39:39]
✅ The SQL query is valid.

=== Test 48 ===
Query: UPDATE users SET role='admin' WHERE id=1 OR 1=1;
Taint: 1 OR 1=1
TAINT position: [39:46]
❌ The SQL query is invalid. {syntax is valid but taint contains code}
```

הקוד אכן ניתח **בצורה נכונה** את כל 11 השאילתות שמוזכרות במאמר, וכן הצליח לפרסר ולנתח בצורה מלאה ונכונה גם את קובץ הנתונים הנוסף שהבאנו לו.

כמובן שהקוד המלא של האלגוריתם, כולל קבצי הקלט והפלט, נמצאים בקישור SQLInject.

ה - בעיות בהן נתקלנו ומטלות שביצענו

בעיות ואתגרים טכניים

1. **שילוב והבנה של הפארסר מגיטהאב (ANTLR)**
בתחילת הדרך הורדנו גרסה של Grammar עבור SQL מאתר GitHub, אך תהליך העבודה עמו היה לא פשוט כלל:
   - לקח זמן רב עד שהמערכת הצליחה להריץ שאילתה פשוטה ולזהות את מבנה העץ שלה – עברנו שלב מתיש של תיעוד שגיאות, הגדרות נתיב (classpath), ותיקוני תאימות.

2. **פונקציות ללא מימוש – רק תיאור מילולי**
הפונקציות הקריטיות כמו val(p, l, h) ו־FV(p, l, h) הופיעו במאמר ללא כל מימוש פורמלי או דוגמת קוד.
במקרים רבים הן תוארו רק במילים כלליות (למשל: "הקטע הקצר ביותר המכיל את הסימבולים מ־l עד h אשר מהווה ערך תחבירי").
מכיוון שכך, נדרשנו לפרש את הכוונה מאחורי כל פונקציה ולבנות אותה מאפס בפייתון:
   - היה עלינו להבין מה נחשב ערך (val) לפי תחבירSQL .
   - אילו משתנים נחשבים חופשיים ואילו לא — (FV) הגדרה לא טריוויאלית כלל.
   - ליצור תהליך של חיפוש תת־מחרוזות.

3. **המרת פסאודו-קוד לקוד אמיתי**
חלקים מהאלגוריתם המרכזי ובמיוחד ((code(p, i הוצגו בפסאודו-קוד בלבד.
זה דרש מאיתנו:
   - להבין את הלוגיקה האיטרטיבית של החיפוש בתתי-מחרוזות.
   - לזהות נקודות בהן ניתן לבצע אופטימיזציה.
   - לוודא שהמימוש עוקב במדויק אחרי הכוונה המקורית גם בהיעדר דוגמת קוד.

4. **טיפול בשגיאות פנימיות של הפארסר**
נדרשנו לתפוס חריגות ולוודא מה מקורן והאם הן רלוונטיות אלינו או לא.

מטלות שביצענו:

1. **הורדה והתקנה של פארסר SQL מבוסס ANTLR**
כולל התקנות סביבת Java, אנטלר, והרצת הפקודות ליצירת המחלקות הדרושות לשפת Python.

2. **ניתוח תחבירי לשאילתות**
ווידוא ש־ ANTLRמסוגל לפענח כל שאילתה, לזהות רכיבים ולבנות עץ תחביר תקין.

3. **מימוש של פונקציות ,ival, FV, NCV־ code בפועל**
כולל הגדרה פורמלית לפי פירוש תיאורי המאמר, כתיבת טסטים פנימיים ובחינה של קלטים שונים.

4. **בדיקה על הדוגמאות מהמאמר**
עבור 11 שאילתות – כל אחת נותחה לפי הקוד שלנו, סווגה לפי/inject לא inject, והתוצאה נבדקה מול התוצאה הצפויה.

5. **הרחבת הבדיקה לדאטאסט נוסף**
יצרנו קובץ דוגמאות נוסף המכיל שאילתות חדשות — גם תקניות וגם חשודות להזרקה — כדי לבדוק את עמידות האלגוריתם למקרים חדשים.

6. **ניתוח תוצאות והצגתן בטבלאות**
עבור כל שאילתה נרשמו: הטקסט המלא, ה־ taint, והאם זוהה כהזרקה.

7. **שיפור רציף של הקוד**
כולל דיבאגינג, שיפור ביצועים, ושכתוב חלקים מהאלגוריתם שנכתבו בצורה לא יעילה בתחילה.

ו - ביבליוגרפיה

המאמר הנוכחי-

- **Donald Ray and Jay Ligatti. 2012. Defining code-injection attacks. In Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '12). Association for Computing Machinery, New York, NY, USA, 179–190.**

השתמשנו על מנת לבנות את קוד השחזור:

- ANTLR4 MySQL Grammar – [https://github.com/antlr/grammars-v4/tree/master/sql/mysql]

מראי המקום של המאמר:

- [1] C. Anley. Advanced SQL injection in SQL server applications. White paper, Next Generation Security Software, 2002.
- [2] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. Candid: preventing SQL injection attacks using dynamic candidate evaluations. In Proceedings of the ACM Conference on Computer and Communications Security, pages 12–24, 2007.
- [3] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. ACM Trans. Inf. Syst. Secur., 13(2):1–39, Feb. 2010.
- [4] M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. Science of Computer Programming, 75(7): 473–495, July 2010.
- [5] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using parse tree validation to prevent sql injection attacks. In SEM ’05: Proceedings of the 5th international workshop on software engineering and middleware, pages 106–113, 2005.
- [6] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In Proceedings of the ACM International Symposium on Software Testing and Analysis, pages 196–206, 2007.

- [7] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. Ccured in the real world. SIGPLAN Notices, 38:232–244, May 2003.
- [8] W. Halfond, A. Orso, and P. Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. IEEE Trans. Softw. Eng., 34(1):65–81, 2008.
- [9] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQLInjection Attacks and Countermeasures. In Proceedings of the IEEE International Symposium on Secure Software Engineering, March 2006.
- [10] R. Hansen and M. Patterson. Stopping Injection Attacks with Computational Theory, July 2005. In Black Hat USA.
- [11] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In Proceedings of the General Track of the USENIX Annual Technical Conference, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.
- [12] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In Proceedings of the IEEE Symposium on Security and Privacy, pages 258–263, 2006.
- [13] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic cre- ˙ ation of SQL injection and cross-site scripting attacks. In Proceedings of the International Conference on Software Engineering, May 2009.
- [14] K. Kline and D. Kline. SQL in a Nutshell, chapter 4. O'Reilly, 2001.
- [15] D. E. Knuth. On the translation of languages from left to right. Information and Control, 8(6):607–639, 1965.
- [16] P. J. Landin. The mechanical evaluation of expressions. Computer Journal, 6(4):308–320, 1963.
- [17] Z. Luo, T. Rezk, and M. Serrano. Automated code injection prevention for web applications. In Proceedings of the Conference on Theory of Security and Applications, 2011.
- [18] Microsoft. SQL Minimum Grammar, 2011. http://msdn. microsoft.com/en-us/library/ms711725(VS.85).aspx.
- [19] Microsoft. CREATE FUNCTION (Transact-SQL), 2011. http: //msdn.microsoft.com/en-us/library/ms186755.aspx.
- [20] CWE/SANS Top 25 Most Dangerous Software Errors. The MITRE Corporation, 2009. Document version 1.4, http://cwe.mitre. org/top25/archive/2009/2009_cwe_sans_top_25.pdf.
- [21] CWE/SANS Top 25 Most Dangerous Software Errors. The MITRE Corporation, 2010. Document version 1.08, http://cwe.mitre. org/top25/archive/2010/2010_cwe_sans_top25.pdf.
- [22] CWE/SANS Top 25 Most Dangerous Software Errors. The MITRE Corporation, 2011. Document version 1.0.2, http://cwe.mitre. org/top25/archive/2011/2011_cwe_sans_top25.pdf.
- [23] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. ACM Trans. Program. Lang. Syst., 27:477–526, May 2005.
- [24] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In Proceedings of the Network and Distributed System Security Symposium, Feb. 2005.
- [25] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In Proceedings of the IFIP International Information Security Conference, pages 372–382, 2005.
- [26] G. Ollmann. Second order code injection attacks. Technical report, NGS Software, 2004.
- [27] Oracle. How to write injection-proof PL/SQL. An Oracle White Paper, December 2008. URL http://www.oracle. com/technetwork/database/features/plsql/overview/ how-to-write-injection-proof-plsql-1-129572.pdf. Page 11.
- [28] Oracle. CREATE FUNCTION Syntax for User-Defined Functions, 2011. http://dev.mysql.com/doc/refman/5.6/en/ create-function-udf.html.
- [29] Oracle. CREATE FUNCTION, 2011. http://download.oracle. com/docs/cd/E11882_01/server.112/e17118/statements_ 5011.htm.
- [30] php. phpMyAdmin. http://www.phpmyadmin.net.
- [31] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In Proceedings of Recent Advances in Intrusion Detection (RAID), 2005.
- [32] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. Theoretical Computer Science, 1(2):125–159, 1975.

- [33] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In Proceedings of the IEEE Symposium on Security and Privacy, May 2010.
- [34] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In Proceedings of the 33rd ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, pages 372–382, 2006.
- [35] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 87–97, 2009.
- [36] S. Tzu. The art of war. The Project Gutenberg eBook. Translated by Lionel Giles. http://www.gutenberg.org/cache/epub/17405/ pg17405.txt.
- [37] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2007.
- [38] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In Proceedings of the 15th USENIX Security Symposium, 2006.
- [39] Y. Younan, P. Philippaerts, F. Piessens, W. Joosen, S. Lachmund, and T. Walter. Filter-resistant code injection on ARM. In Proceedings of the ACM Conference on Computer and Communications Security, pages 11–20, 2009.
- [40] X. Zhang and Z. Wang. A static analysis tool for detecting web application injection vulnerabilities for ASP program. In International Conference on e-Business and Information System Security (EBISS), pages 1 – 5, May 2010.