Vol.2021-CSEC-93 No.7
Vol.2021-IOT-53 No.7
2021/5/13

DPSを用いたSQLインジェクション対策手法

長谷川　直哉[1,a)]　今泉　貴史[2,b)]

概要：Web アプリケーションの脆弱性の 1 つに SQL インジェクションがある．本研究では SQL インジェクションの対策手法として DPS（Dual Proxy to prevent SQL Injection Attack）を提案し実装した．DPS は HTTP プロキシとデータベースプロキシを組み合わせたアーキテクチャであり Web サーバを挟む構成になっている．DPS では，ユーザインプット（ユーザからの入力値）と Web アプリケーションから送られるクエリを基に SQL インジェクションを検知する．DPS が実際に SQL インジェクションを防止できるかどうかを確かめるための実験を行い，DPS が SQL インジェクションを防げることを確認した．

Countermeasure method of SQL injection using DPS

1. はじめに

インターネットは生活をする上で必要不可欠なものとなっており，インターネット上には生活を支える多種多様な Web アプリケーションが存在している．しかし，IPA が毎年発表している情報セキュリティ 10 大脅威 2021 によると，「インターネット上のサービスからの個人情報の盗取」の項目が 7 位にランクインする．[1] このことからも，未だに Web アプリケーションから個人情報が流失してしまう脅威があることは事実である．

Web アプリケーションから情報を盗取される脆弱性の 1 つに SQL インジェクションがある．これは，攻撃者が不正な入力値を Web アプリケーションに送信することで，想定外のクエリを発生させ，データベースに不正にアクセスできてしまう脆弱性である．SQL インジェクションが起きてしまうと，データベース内の個人情報が盗取される可能性があり，サービス提供者の信頼を落としかねない甚大な被害をもたらす．既存の対策として，WAF やプリペアドステートメントが挙げられる．前者は，シグネチャを基に検知を行うが，全ての攻撃に合わせてシグネチャをチューニングすることは容易ではない．後者は使用すれば，SQL インジェクションを防ぐことはできるが，Web アプリケーションのソースコードを改修する必要がある．しかし，既存の Web アプリケーションのソースコードの改修は必ずしも容易ではなく，ソースコードを改修できない Web アプリケーションではプリペアドステートメントを利用できない．

そこで本研究では，SQL インジェクションの対策手法として DPS（Dual Proxy to prevent SQL Injection Attack）を提案する．DPS では，シグネチャではなく構文構造を用いて SQL インジェクションを検知するため，シグネチャの更新やチューニング作業をする必要がない．また，DPS を利用することでソースコードの改修が不可能な Web アプリケーションであっても，ソースコードの改修を行うことなく SQL インジェクションの検知が可能である．よって，DPS では既存手法の問題を解消できる．

以下，2 章では，SQL インジェクションについて説明する．3 章では，提案手法について説明する．4 章では提案手法を用いた実験を行い，5 章で実験についての考察を行う．最後に 6 章でまとめと今後の課題について述べる．

2. SQL インジェクション

2.1 概要

Web アプリケーションでは，ユーザからのユーザインプット（ユーザからの入力値）を用い SQL 文を構成し，データベースにクエリを送信することによってデータを取得する．SQL インジェクションとは，攻撃者が不正なユーザインプットを Web アプリケーションに送信することで，

[1] 千葉大学大学院融合理工学府
Graduate School of Science and Engineering, Chiba Uniersity
[2] 千葉大学統合情報センター
Institute of Management and Information Technologies, Chiba University
[a)] n.hase-7fe@chiba-u.jp
[b)] imaizumi_takashi@faculty.chiba-u.jp

想定外のクエリを発生させ，データベースに不正にアクセスできてしまう脆弱性である．具体例を用い SQL インジェクションを説明する．

例えば，ログイン機能を有する Web アプリケーションでは Web アプリケーション内で，SELECT * FROM test WHERE users = ’（ユーザインプット 1）’ AND pass = ’（ユーザインプット 2）’ という SQL 文が構成される．ユーザインプット 1，ユーザインプット 2 はユーザからのリクエストのパラメタを基に構成される．正常なユーザであれば，ユーザインプット 1 に「ユーザ名」，ユーザインプット 2 に「パスワード」を表すパラメタがリクエストに含まれる．よって，正常のユーザの場合，SELECT * FROM test WHERE users = ’ユーザ名’ AND pass = ’パスワード’ というクエリがデータベースに送信される．データベース内でこの，ユーザ名とパスワードに一致するユーザ情報を保持しているかどうかの確認が行われ，結果を Web アプリケーションに返す．

一方，攻撃者の場合はユーザインプット 1 に「ユーザ名」，ユーザインプット 2 に「パスワード’ OR ’1’ = ’1」という値を Web アプリケーションに送信する．すると Web アプリケーションからデータベースに，SELECT * FROM test WHERE users = ’ユーザ名’ AND pass = ’パスワード’ OR ’1’ = ’1’ というクエリが送信される．このクエリでは，OR 以下の条件である’1’ = ’1’ が評価されてしまい，ユーザ名とパスワードの正当さに関わらず常に真となる．よって，攻撃者は正当なユーザ名とパスワードを知らずともログインできてしまい，個人情報の盗取に繋がる恐れがある．

2.2 既存の対策手法

SQL インジェクションの既存の対策手法として WAF，プリペアドステートメントがある．

2.2.1 WAF

WAF（Web Application Firewall）とは，Web アプリケーションの脆弱性に対する攻撃を防御するセキュリティシステムである．WAF では，Web アプリケーションのリクエストに対してシグネチャに基づき検知を行う．検知で使われるシグネチャの設定方法には，主にブラックリスト方式とホワイトリスト方式が用いられる．WAF の利点として，脆弱性がある Web アプリケーションであってもコードを改修することなく WAF を導入するだけで対策できる点が挙げられる．しかし，欠点として新しい攻撃パターンが出現するごとにシグネチャを更新しなければならない点や Web アプリケーション毎にシグネチャをチューニングする必要がある．

2.2.2 プリペアドステートメント

プリペアドステートメント（静的プレースホルダ）では，値がバインドされる前の SQL 文を事前にデータベース側に渡しておき，データベース上でコンパイルなどの実行準備が行われ SQL 文を確定する．そして，データベース上でバインド値を当てはめた後に SQL 文が実行される．よって，利点として，プレースホルダの状態で SQL 文がコンパイルされるため SQL 文が変更される可能性はなく SQL インジェクションは起こらない．[2] 新たに Web アプリケーションを作成する場合には導入することは可能であるが，既存の Web アプリケーションでプリペアドステートメントを利用するには，ソースコードの改修が必要になる．しかし，この改修は必ずしも容易ではなく，実際にプリペアドステートメントを用いない Web アプリケーションも多数存在する．このように，ソースコードの改修が困難な Web アプリケーションでは，プリペアドステートメントは利用できない．

2.2.3 既存研究

Buehrer らが提案した SQLGuard は，ユーザインプットを含まない SQL 文の構造と，実際にユーザから送られてきたユーザインプットを含んだ SQL 文の構造を比較する．2 つの SQL 文の構造を比較し構造が一致しなかった場合に SQL インジェクションと判断する．[3]Halfond らの調査によると，SQLGuard を使用することで SQL インジェクションを防止できることが確認されている．[4] しかし，SQLGuard を使用するためには Web アプリケーションのソースコードの一部を改修しなければならない．

A Liu らが提案した SQLProb は，Web サーバとデータベースの間に挟むアーキテクチャである．Web アプリケーションから流れてくるクエリからユーザインプットを推測し，クエリとユーザインプットを基に良性のクエリか悪性のクエリかを構文構造を利用して判断する．[5]SQLProb により SQL インジェクションを防ぐことはできるが，問題点としてユーザインプットを正確に推測できない点がある．SQLPronb では，ユーザインプットとクエリを基に検知を行うため，ユーザインプットを正確に推測できない場合は正確に検知できない可能性が考えられる．

3. DPS

3.1 概要

本研究では SQL インジェクションの対策手法として DPS（Dual Proxy to prevent SQL Injection Attack）を提案する．DPS は HTTP プロキシとデータベースプロキシを組み合わせたアーキテクチャであり Web サーバを挟む構成になっている．図 1 は DPS の概略図である．HTTP プロキシの役割をもつ DPSh とデータベースプロキシの役割を持つ DPSd によって構成される．DPS では，ユーザからのユーザインプットと Web アプリケーションから送られるクエリを基に検知アルゴリズムを用い SQL インジェクションを検知する．

DPS では，Web サーバを挟むアーキテクチャ構造によっ

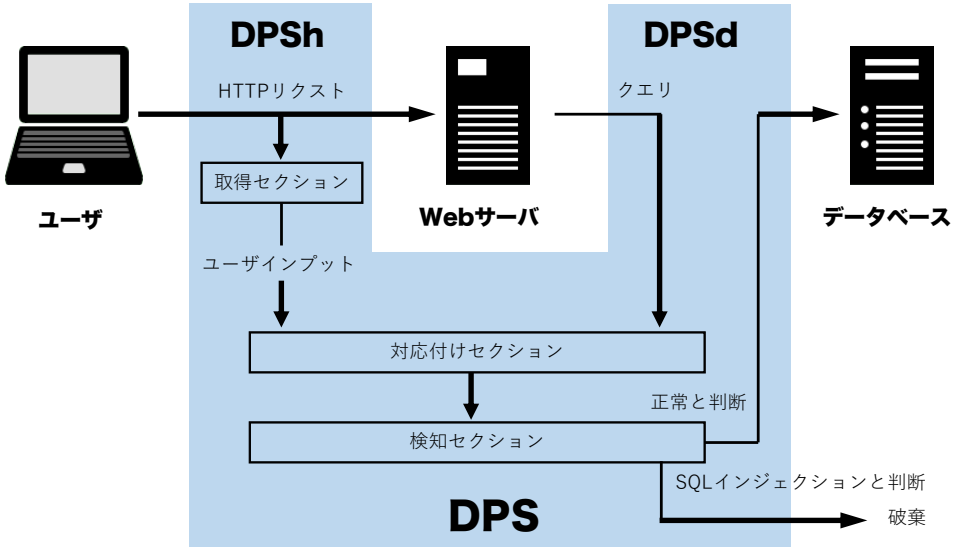


図 1: DPS の概略図
Fig. 1 Overview of the DPS system architecture

て SQL インジェクションの対策を行うため，既存の Web アプリケーションであってもソースコードを改修をせずに利用できる．また，シグネチャではなく構文構造を基に SQL インジェクションの検知を行うため，シグネチャの更新やチューニングを行う必要がない．従って，2.2 既存の対策手法で挙げた問題点も解消できる．

SQL インジェクションの検知の流れを説明する．先ず取得セクションにおいて，DPSh でユーザから送られてくる HTTP リクエストの中からユーザインプット（ユーザからの入力値）を抽出し，キューに保存する．次に，対応付けセクションにおいて，DPSd で取得したクエリとキューに保存したユーザインプットを用い対応付けを行う．最後に対応付けが完了したクエリとユーザインプットを検知セクションにおいて SQL インジェクションかどうかを検知する．SQL インジェクションと判定された場合は，該当クエリを破棄する．

3.2 取得セクション

DPSh においてユーザから送られた HTTP リクエストを取得する．その中からユーザインプットを抽出し，キューに保存する．ここでのキューは，キューサイズの値によって保持できるユーザインプットの個数を変化させることができる．例えば，キューサイズが 5 である場合は，最新に取得したユーザインプットから遡って 5 つのユーザインプットを保持できる．具体的な例を表 1 に載せる．この場合，

表 1: キューの保持例
Table 1 A example of holding the queue

No.	ユーザインプット
1	hasegawa，1234567，ログイン
2	oshima，ashkeifd89，ログイン
3	chibau，lKltY24kYed，ログイン
4	labima，asdfghj，ログイン
5	dps，abc’ OR ’1’=’1，ログイン

最新に取得したものが No.5，一番古いものが No1 である．このキューサイズを大きくすればするほど多くのユーザインプットを保持することは可能である．しかし，キューサイズを大きくすればするほど，この後の対応付セクションでの処理が増大する．

3.3 対応付けセクション

先ず，DPSd で Web アプリケーションからデータベースへ送られるクエリを取得する．このクエリと取得セクションで保持したユーザインプットとの対応付けを行う．対応付けは，クエリの中にユーザインプットを含んでいるか否かで判断する．対応付けができたユーザインプットとクエリだけが次の検知セクションに進む．

3.4 対応付けアルゴリズム

Algorithm1 に対応付けアルゴリズムを記載する．$Queue$ は最大で $QueueSize$ のユーザインプットを保持できる．

Algorithm 1 対応付けアルゴリズム

Require: $Queue$, $Query$
  $Queue$ のキューサイズを $QueueSize$ とする
  $UserInput_n \in Queue$, ただし $n \leq QueueSize$ となり,
  $UI_m \in UserInput_n$ とする
  for $i \leq QueueSize$ **do**
    for $j \leq m$ **do**
      if $UI_j \in Query$ **then**
        $UserInput_i$ と $Query$ を対応付ける
      end if
    end for
  end for

$Queue$ が保持している $UserInput_n$ の要素を UI_m とする．UI_m が DPSd に到着した $Query$ に含まれている時，$UserInput_n$ と $Query$ は対応付け関係があるとする．例えば，`SELECT * FROM test WHERE users = 'dps' AND pass = 'abc' OR '1'='1'` というクエリが DPSd に到着し、この時保持しているユーザインプットが表 1 であるとする．Algorithm1 より，この例ではクエリと対応するユーザインプットは No.5 となる．今回の例では，クエリとユーザインプットの対応付けがとれたが，どのユーザインプットとも対応付けができないクエリの存在も考えられる．そのクエリはユーザからの影響が及んでいないと考えられるため，この時点で SQL インジェクションではないと判断できる．

3.5 検知セクション

対応付けセクションで対応関係のあるユーザインプットとクエリを検知アルゴリズムに通し，SQL インジェクションかどうかを判定する．SQL インジェクションと判定された場合は，該当クエリを破棄し，SQL インジェクションと判定されなかったクエリのみをデータベースに送信する．

3.6 検知アルゴリズム

DPS では，WAF のようなシグネチャを基にした検知ではなく構文構造を基に検知を行う．Su らの研究では，悪意のあるユーザは Web アプリケーションの設計者が意図した制約を超えてクエリを実行しようとするのに対し，正常なユーザはそのような制約を破ろうとしないことに着目し，この違いを SQL インジェクションかどうかの違いと定義した．[6] 本研究では，この考え方のようにユーザインプットの影響で構文木構造が変化したかどうかを調べることで SQL インジェクションを判定する．つまり，ユーザインプットの影響で構文木構造が変化しないクエリを正常なクエリ，構文木構造が変化したクエリを SQL インジェクション（悪性クエリ）とする．ここで，ユーザインプットオルトを定義する．ユーザインプットオルトとは，ユーザインプットの要素をシングルクォート又は空白で区切り生成したものである．

Algorithm 2 検知アルゴリズム

Require: $UserInput$, $Query$
  $UserInput$ をシングルクォート又は空白で区切り
  $UserInputAlt$ を生成する
  $Query$ をパースし葉ノードの集合を $Leaf$ とする
  各々の要素を $UI_n \in UserInput$,
  $UIAlt_m \in UserInputAlt$, $Lf_l \in Leaf$ とする
  for $i \leq n$ **do**
    for $j \leq l$ **do**
      if $UI_i == Lf_j$ **then**
        $count$++
      end if
    end for
  end for
  for $i \leq m$ **do**
    for $j \leq l$ **do**
      if $UIAlt_i == Lf_j$ **then**
        $countAlt$++
      end if
    end for
  end for
  if $count == countAlt$ **then**
    SQL インジェクション攻撃ではない
  else
    SQL インジェクション攻撃である
  end if

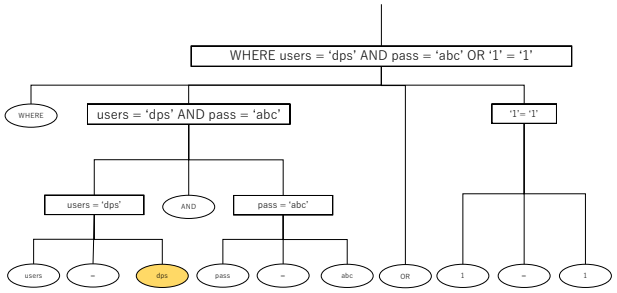
構文木構造が変化しかどうかを，ユーザインプット，ユーザインプットオルト及びクエリをパースした構文木の葉ノードに着目し判定する．対応関係のあるユーザインプットとクエリでは，ユーザインプットによって構成される葉ノードが必ず 1 つ以上存在する．この時，ユーザインプットオルトによって構成される葉ノードの個数が変化する場合がある．この葉ノードの個数が変化した時，構文木構造が変化したと定義する．つまり，ユーザインプットによって構成される葉ノードの個数とユーザインプットオルトによって構成される葉ノードの個数が一致しない時，SQL インジェクションであると定義する．

Algorithm2 に検知アルゴリズムを記載する．$UserInput$ の要素である UI_n と $Leaf$ の要素である Lf_l が一致した個数 $count$ と $UserInputAlt$ の要素である $UIAlt_m$ と Lf_l が一致した個数 $countAlt$ が等しい時，ユーザインプットによってクエリの構文木構造が変化していないため SQL インジェクション攻撃ではないと判定する．

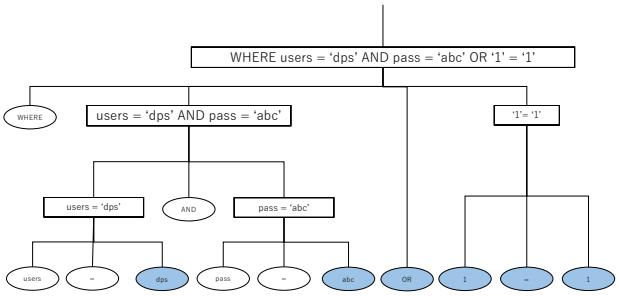
3.6.1 SQL インジェクションの検知例

具体例を，3.3 対応付けセクションで挙げた，以下のユーザインプットとクエリを用いて説明する．また，$UserInputAlt$ は，Algorithm2 に則して記載した．

$UserInput$
　dps，abc' OR '1'='1，ログイン
$UserInputAlt$
　dps，abc，OR，1，=，1，ログイン
$Query$

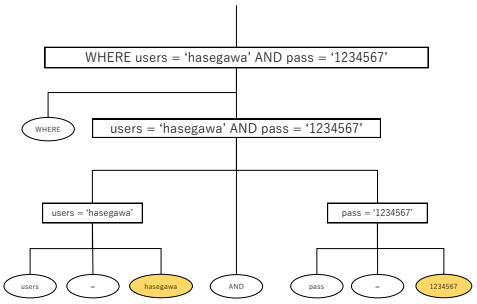


(a) $UserInput$ と $Leaf$ の一致

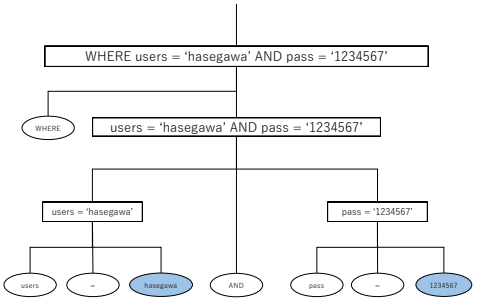


(b) $UserInputAlt$ と $Leaf$ の一致

図 2: SQL インジェクションの例
Fig. 2 A example of SQL injection



(a) $UserInput$ と $Leaf$ の一致



(b) $UserInputAlt$ と $Leaf$ の一致

図 3: 正常なクエリの例
Fig. 3 A example of successful queries

SELECT * FROM test WHERE users = 'dps' AND pass = 'abc' OR '1'='1'

図 2 は，$Query$ をパースした構文木の部分木を表している．丸で表したノードが葉ノードであり，この葉ノードが Lf_l である．図 2a は，$UserInput$ と $Leaf$ の一致部分を示している．黄色で示した葉ノード，dps が一致した箇所である．よって，$count$ は 1 である．図 2b は，$UserInputAlt$ と $Leaf$ の一致部分を示している．青色で示した葉ノード，dps，abc，OR，1，=，1 が一致した箇所である．よって，$countAlt$ は 6 である．従って，$count \neq countAlt$ であるため，この例のクエリは SQL インジェクションと判断する．

3.6.2 正常なクエリの検知例

3.6.1 SQL インジェクションの検知例と同様に，以下のユーザインプットとクエリを用いて説明する．

$UserInput$

hasegawa，1234567，ログイン

$UserInputAlt$

hasegawa，1234567，ログイン

$Query$

SELECT * FROM test WHERE users = 'hasegawa' AND pass = '1234567'

図 3a は，$UserInput$ と $Leaf$ の一致部分を示している．黄色で示した葉ノード，hasegawa，1234567 が一致した箇所である．よって，$count$ は 2 である．図 3b は，$UserInputAlt$ と $Leaf$ の一致部分を示している．青色で示した葉ノード，hasegawa，1234567 が一致した箇所である．よって，$countAlt$ は 2 である．従って，$count == countAlt$ であるため，この例のクエリは正常なクエリと判断する．

4. 実験

実験に際し，Docker 上に DPS 及び SQL インジェクションの脆弱性を持つ標的 Web アプリケーションを実装した．SQL 文のパーサーとして，Python のモジュールである sqlpase[7] を用い，また，攻撃ツールとして脆弱性診断ツールである OWASP ZAP[8] を用いた．実装環境は表 2 の通りである．

今回の実験では，DPS が SQL インジェクションを防止できるかを確かめる実験，及びキューサイズの増大に伴う

表 2: 実装環境
Table 2 Implementation environment

仮想環境	Docker 20.10.2
DPS	Python 3.9.1
標的 Web アプリ	PHP 7.4.13 Mysql 8.0.22
脆弱性診断ツール	OWASP ZAP2.8.0

表 3: 検知結果
Table 3 The result of detection

	DPS	SQL インジェクション
実験 1	未使用	あり
実験 2	使用	防止
実験 3	使用	防止

オーバーヘッドの増大を確認する実験を行った．

4.1 検知実験

DPS が SQL インジェクションを防止できるかどうかを確かめるために，SQL インジェクションの脆弱性が内在する標的 Web アプリケーションを用意した．この標的 Web アプリケーションに DPS を利用することで SQL インジェクションを防止できるかを確かめる．以下の 3 つの状況を想定して検知実験を行った．また、この時のキューサイズは 5 とした．SQL インジェクションの脆弱性が内在する Web アプリケーションに対して DPS を利用する．この時，SQL インジェクションの脆弱性を確認できない場合は DPS によって SQL インジェクションを防止できたことを意味する．

実験 1
標的 Web アプリケーションに OWASP ZAP を使用し脆弱性診断をする

実験 2
標的 Web アプリケーションに DPS を利用し実験 1 と同様に ZAP を使用する

実験 3
標的 Web アプリケーションに DPS を利用し 2 台から同時に ZAP を使用する

実験結果は表 3 の通りである．実験 1，実験 2 の比較より，SQL インジェクションの脆弱性が内在する Web アプリケーションであっても，ソースコードを改修することなく DPS を利用することで SQL インジェクションを防止することができた．また，実験 2，実験 3 の比較より 2 つの異なるリクエストが混在している場合であっても SQL インジェクションを防止することができた．

4.2 オーバーヘッド計測

次に，キューサイズを増やすことによるオーバーヘッドの増大について調べる実験を行う．キューサイズを増やすと，3.2 取得セクションにおいて保持できるユーザインプットの数が増えることになる．保持できるユーザインプット数が増えるため，3.3 対応付けセクションの処理が増大する．従って，キューサイズを増やすことにより 1 リクエストあたりのオーバーヘッドが増大すると予想できる．今回の実験では，4.1 検知実験の実験 2 と同様な状況において，キューサイズを 1，5，25，・・・と増やしていった場合のラウンドトリップタイムに着目しオーバーヘッドの増加を調べる．ここでのラウンドトリップタイムは，ZAP が出したリクエストに対し到着したレスポンスまでの時間のことを言う．

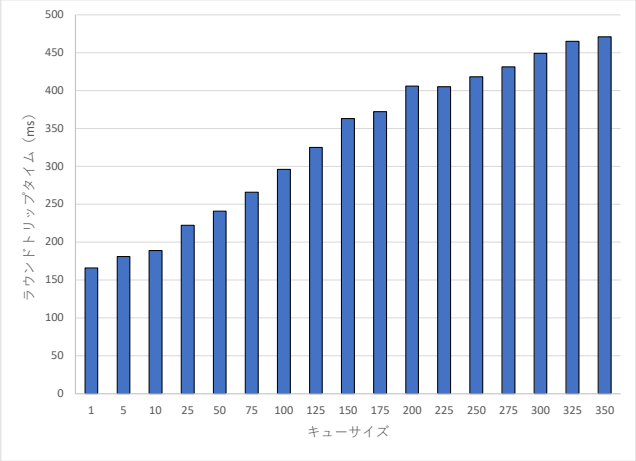


図 4: オーバーヘッドの増加
Fig. 4 Increased overhead

結果を図 4 に示す．図 4 より，キューサイズを増やすほど，ラウンドトリップタイムが増加する事が分かる．例えば，キューサイズが 150 である時はキューサイズが 1 である時に比べてラウンドトリップタイムが約 2 倍になっている．このことから，キューサイズの大きさがオーバーヘッドに影響する要因であることが分かる．また，これら全ての実験結果において SQL インジェクションは防止できていた．

5. 考察

キューサイズを $QueueSize$，DPS に同時接続するユーザ数を U_{num} とすると，DPS では $QueueSize \geq U_{num}$ の場合でしか SQL インジェクションを防止できない．これは，キューサイズ分しかユーザインプットを保持できないためであり，キューサイズを超えるユーザからのリクエストには正常に対処できない．従って，同時通信するユーザが増えれば増えるほどキューのサイズも大きくしていかなければならない．しかし，4.2 オーバーヘッドの計測よりキューサイズが増大すればするほど 1 リクエストあたりのオーバーヘッドが増大してしまう．よって，キューサイズの増大はオーバヘッドに影響し Web アプリケーション

のサービス低下に繋がってしまう．DPS を利用しながら Web アプリケーションのサービスの質を維持するには，同時通信できるユーザ数を把握し，それに伴う適当なキューサイズを決定しなければならない．

また，DPS で SQL インジェクションかどうか検知できないクエリとして検索エンジンで用いられる AND 検索，OR 検索の機能が考えられる．これはユーザインプット内に SQL の SELECT 文の WHERE 句で使われる論理演算子（AND，OR）を含むため，正常なクエリであっても構文木構造が変化する．DPS では，構文木構造が変化するクエリを SQL インジェクションと判定するため，正常なクエリも SQL インジェクションと判定してしまう．

6. まとめ

本研究では，既存の SQL インジェクション対策の問題点を改善した DPS（Dual Proxy to prevent SQL Injection Attack）を提案し実装した．DPS では，プリペアドステートメントのように Web アプリケーションのソースコードを改修することなく SQL インジェクションを防ぐことができた．また，WAF のようにシグネチャを利用するのではなく構文木構造を利用して SQL インジェクションを検知することができた．

DPS はアーキテクチャの構造上，DPSh と DPSd で Web サーバを挟む形になっているため様々な拡張を期待できる．今後の課題として生成クエリ推測機能がある．これは，DPSh から様々なユーザインプットを流し，DPSd で実際に Web アプリケーションで生成されたクエリを観察することで，Web アプリケーションのソースを見ることなく生成されるクエリを予測する機能である．また，脆弱性診断機能も考えられる．DPSh に予め既知の SQL インジェクションの攻撃パターンを登録する．この攻撃パターンを擬似攻撃として DPSh から流し，DPSd で流れてきたクエリを取得する．この時，検知アルゴリズムにかけ SQL インジェクションかどうかを判定する．SQL インジェクションと判定された場合は，その Web アプリケーションに SQL インジェクションの脆弱性があると判断できる．DPS は他のネットワークに繋がなくとも単体として機能するのでテスト段階の Web アプリケーションを安全に検査したり，自動で診断を行える．

参考文献

[1] IPA 独立行政法人情報処理推進機構：情報セキュリティ 10 大脅威 2021, IPA 独立行政法人情報処理推進機構（オンライン），入手先 ⟨https://www.ipa.go.jp/files/000088835.pdf⟩（参照 2021-3-17）.
[2] 徳丸　浩：体系的に学ぶ安全な Web アプリケーションの作り方第 2 版，SB クリエイティブ株式会社 (2018).
[3] Buehrer, G., Weide, B. W. and Sivilotti, P. A.: Using parse tree validation to prevent SQL injection attacks, *Proceedings of the 5th international workshop on Software engineering and middleware*, pp. 106–113 (2005).
[4] Halfond, W. G., Viegas, J., Orso, A. et al.: A classification of SQL-injection attacks and countermeasures, *Proceedings of the IEEE international symposium on secure software engineering*, Vol. 1, IEEE, pp. 13–15 (2006).
[5] Liu, A., Yuan, Y., Wijesekera, D. and Stavrou, A.: SQLProb: a proxy-based architecture towards preventing SQL injection attacks, *Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 2054–2061 (2009).
[6] Su, Z. and Wassermann, G.: The essence of command injection attacks in web applications, *Acm Sigplan Notices*, Vol. 41, No. 1, pp. 372–382 (2006).
[7] andialbrecht: andialbrecht/sqlparse: A non-validating SQL parser module for Python, , available from ⟨https://github.com/andialbrecht/sqlparse⟩ (accessed 2021-3-22).
[8] OWASP: OWASP ZAP Zed Attack Proxy, OWASP (online), available from ⟨https://owasp.org/www-project-zap/⟩ (accessed 2021-3-22).