CL-005

Web アプリケーションテストを用いた SQL クエリのホワイトリスト自動作成手法
Automatic Whitelist Generation for SQL Queries Using Web Application Test

野村 孔命[1)]　力武 健次[2)]　松本 亮介[1)]
Komei Nomura　Kenji Rikitake　Ryosuke Matsumoto

1　はじめに

Web アプリケーションの脆弱性を利用した攻撃は後を絶たず [10]，Web サービスが保有する個人情報やサービス特有の機密情報を窃取するセキュリティインシデントが発生している [12]．このような攻撃は，Web アプリケーションが通常稼動時に発行しない不正クエリをデータベースで実行させることによって引き起こされる．また，1 つの不正クエリの実行が大規模な情報漏洩につながることもあり，結果的に，サービスの信頼性低下を招いてしまう．そのため，不正クエリはデータベースで実行される前に検知する必要がある．しかし，Web サービスの運営においては，不正クエリ対策導入によって既存のシステム構成や開発プロセスに変更があり，サービスの開発や運用に支障がでることは避けたい．また，PHP や Ruby など様々なプログラミング言語が実装に用いられる Web サービスにおいては，Web アプリケーションの実装に依存せず汎用的に利用できる不正クエリ対策が求められている．

Web アプリケーションが通常稼働時に発行しない不正クエリをデータベースで実行される前に検知するには，ネットワーク攻撃検知に用いられる不正検知 [13] が応用可能である [7]．不正検知には，既知の不正なパターン（ブラックリスト），もしくは既知の正常なパターン（ホワイトリスト）を定義して，パターンマッチングにより検知する方法がある．ブラックリスト方式では，不正クエリのパターンを正確に定義できた場合は誤検知率が少なくなり，既知の不正クエリ検知には有効である．しかし，不正クエリのパターンを定義するために要求される攻撃に関する事前知識が膨大であることや，全ての既知の不正クエリのパターンを定義できたとしても未知の不正クエリを検知することができないという課題がある．一方で，ホワイトリスト方式では，Web アプリケーションが発行するクエリをホワイトリストに定義し，定義されていないクエリが不正クエリとして検知される [3]．この方法は，未知の不正クエリが発行されたとしても，そのクエリがホワイトリストに定義されていない場合は検知可能である．

不正クエリを検知するためのホワイトリスト方式の適用例として，Web アプリケーションが発行するクエリのホワイトリストを開発者が手動で作成する方法がある．しかし，大規模で複雑な Web アプリケーションでは発行クエリ数が膨大となり，開発者が全ての発行クエリを把握するのは困難である．さらに，作成したホワイトリストは Web アプリケーションの更新によって発行されるクエリが変化するため，その都度更新しなければならず，運用者への負担は大きい．そのため，Web アプリケーションが発行するクエリのホワイトリストを自動で作成する手法が提案されている [2][9]．しかし，これらの手法には，Web アプリケーション稼動後にホワイトリストを作成する期間が必要となり稼働後即時で検知できない課題や，Web アプリケーションの実装に依存することによる汎用性が低いという課題がある．

本研究では，Web アプリケーションの動作テストをテストコードとして管理しながら，開発者が Web アプリケーションの変更に追従してテストコードを整備していく開発プロセスにおいて，テスト実行時に発行されるクエリを用いてホワイトリスト自動作成手法を提案する．提案手法は，Web アプリケーションが稼働する前のテストの段階においてホワイトリストを自動作成することで，Web アプリケーション稼働後に即時不正クエリを検知可能な状態にできる．また，ホワイトリスト作成に必要なクエリの収集はデータベースの前段にデータベースプロキシ配置し行うことで，Web アプリケーションの実装に依存しないホワイトリスト作成を実現する．提案手法は，開発者によって Web アプリケーションの動作テストが追加されることによってホワイトリストの網羅性が向上させることができ，作成されたホワイトリストはテスト時に発行されなかったクエリを検知できる．このようなクエリは開発者の想定漏れによってテストされなかったクエリ，もしくは，Web アプリケーションが通常稼働時発行しない不正クエリに限定される．提案手法はこれらのクエリを検知し開発者に通知し対処することで，攻撃の起点となる Web アプリケーションの脆弱性の長期化を防ぐことができる．そのため，提案手法のテストとホワイトリストの関係性を明確にするための実験を行い，Web アプリケーション稼動時に発行されたクエリの中でホワイトリストに登録されていなかったクエリに関して考察した．

本稿の構成を述べる．2 章では，Web アプリケーションが発行するクエリのホワイトリスト作成の課題を整理する．3 章では，提案手法の開発プロセスにおける位置付けと提案手法の設計を述べ，提案手法による検知の特性について述べる．4 章では，テストカバレッジとホワイトリストの関係性の実験結果と考察を述べ，5 章でまとめを述べる．

2　Web アプリケーションが発行するクエリのホワイトリスト作成の課題

Web アプリケーションが発行するクエリのホワイトリスト作成の課題を整理する．Web アプリケーション通常稼働時に発行されない不正クエリがデータベースに発行されたことを検知するために，Web アプリケーションが発行するクエリのホワイトリストを作成して検知する方法がある．ホワイトリストの作成方法として，

1）　GMO ペパボ株式会社 ペパボ研究所
Pepabo R&D Institute, GMO Pepabo, Inc., Tenjin, Chuo ku, Fukuoka 810-0001 Japan
2）　力武健次技術士事務所
Kenji Rikitake Professional Engineer's Office, Toyonaka City, Osaka 560-0043 Japan

表 1　Ruby コードと発行されるクエリの対応例

Ruby コード	発行されるクエリ
User.find(1)	SELECT * FROM users WHERE (users.id = 1) LIMIT 1
User.first	SELECT * FROM users ORDER BY users.id ASC LIMIT 1

開発者が手動で作成する方法がある．このとき，ホワイトリストには，Web アプリケーションが発行するクエリがユーザ入力により変化することから，ユーザ入力が含まれるクエリのリテラル部分をプレースホルダーに置き換えたクエリ構造が用いられる [1]．そのため，開発者は Web アプリケーションのソースコードから全てのクエリ発行処理を特定し，そこから発行されるクエリのクエリ構造を把握しなければならない．しかし，Web サービス開発における Web アプリケーションの以下の特徴から，手動で Web アプリケーションが発行するクエリのホワイトリストを作成することが困難になっており，開発者への負担は増加する傾向にある．

1. Web アプリケーションの大規模化
2. Web アプリケーションの複雑化
3. Web アプリケーションの更新頻度が高い [11]
4. Web アプリケーションの実装にオブジェクトリレーショナルマッピング（ORM）[8] の利用

1. により，Web アプリケーションが発行するクエリ数は増加する．また，2. により，複雑なクエリ発行処理によって発行クエリは多岐に渡るようなり，さらに発行クエリ数は増加する．このことから，開発者が把握しなければならない発行クエリは膨大となり，ホワイトリストを作成が困難となる．また，Web アプリケーションが発行するクエリは開発の進行によって変化が生じるため，3. の特徴から，頻繁に発行クエリが変化する．発行クエリに変化があった場合は，その都度変化に応じてホワイトリストの更新を行わなければならないため，不正クエリ対策によって Web アプリケーションの機能開発に支障がでてしまう．さらに，4. により，開発者は Web アプリケーションが発行するクエリを意識することが少なくなっている．ORM[8] はオブジェクト指向言語におけるオブジェクトとデータベースのレコードを関連づける機能を提供している．これにより，開発者はデータベースのレコードをオブジェクトとして扱えるため，直接 SQL 文を記述することが少なくなる．例えば，Web アプリケーションの実装に Web アプリケーションフレームワークである Ruby on Rails を用いた場合，ORM として ActiveRecord が利用される [5]．開発者は，ActiveRecord を用いて，データベースのレコードと Ruby のクラスオブジェクトを関連づけることで，クラスオブジェクトを用いてクエリの発行処理を記述できる．クラスオブジェクト User がデータベースの users テーブルのレコードに関連付けされている場合の Ruby コードと発行されるクエリは表 1 のようになる． このように，ORM を用いると，開発者はオブジェクトを用いてデータベースからデータを読み出すので，実際に Web アプリケーションが発行するクエリを把握していることは少なくなる．そのため，ホワイトリストを定義するためには，ORM の処理を理解し発行されるクエリを把握しなければならず，これは不正クエリ対策が ORM を使った Web アプリケーション開発の妨げとなっていることを意味する．

ホワイトリストの手動作成は，大規模で複雑な Web アプリケーションでは発行クエリ数が膨大になってしまうことや，Web アプリケーションの改修による発行クエリの変化によってホワイトリストの更新が必要となることから，開発者への負担が大きい対策となる．開発者の負担を軽減するための解決策として，Web アプリケーションが発行するクエリのホワイトリストを自動作成する手法が提案されている．

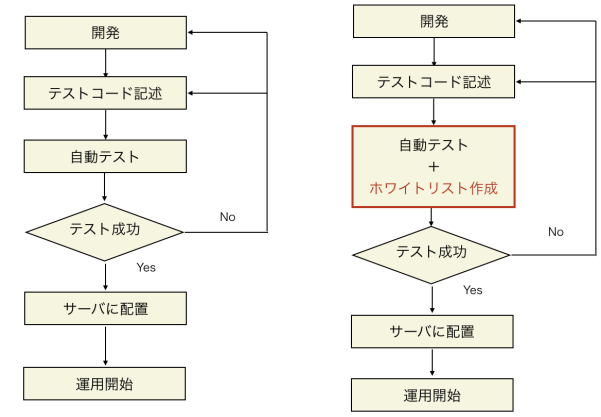
2.1　稼働時のクエリ解析を用いたホワイトリスト自動作成

Web アプリケーションの稼動時に発行されるクエリを収集し，構文解析を行いクエリ構造に変換することで，ホワイトリストを定義する手法が提案されている [2]．この手法には，クエリを収集しホワイトリストを作る学習モードと作成したホワイトリストを用いて検知を行う検知モードがあり，学習モード中は不正クエリの検知を行うことができない．この手法を用いることで，Web アプリケーションの稼動時に自動でホワイトリストを作成することができる．しかし，ホワイトリストの作成には学習期間を要するため，更新頻度が高い Web アプリケーションにおいては，頻繁に再学習を行わなければならず，その都度検知を行えない期間が発生してしまう．これは，Web アプリケーション稼動時にホワイトリストを作成していることが原因で発生し，Web アプリケーション稼動後即時に不正クエリの検知を行うためには，稼動前の段階でホワイトリストを作成する必要がある．

2.2　Web アプリケーションの静的解析を用いたホワイトリスト自動作成

Web アプリケーションのソースコードからクエリの発行処理を特定し，クエリ発行処理を解析することで，ホワイトリストを自動で作成する手法が提案されている [9]．この手法を用いることで，Web アプリケーション稼動前にホワイトリストを作成することができ，Web アプリケーション稼動後即時に不正クエリの検知を行える．一方で，この手法では，ソースコードのクエリ発行処理の解析を行うため，Web アプリケーションの実装に依存してしまい，それぞれの Web アプリケーションに対して解析器を実装しなければならない．これは Web アプリケーションの実装言語が多様化していることと，様々な ORM が利用されるようになっていることが原因で課題となる．

筆者が所属する GMO ペパボ株式会社では，PHP や Ruby のような様々な言語や Web アプリケーションフレームワークを用いて Web サービスの開発が行われている．このような状況において，不正クエリへの対策をすることは重要であるが，対策を行うことによってサービスの運営を妨げることは避けたい．そのため，導入時のシステムの構成や開発プロセスへの影響を抑えつつ，Web アプリケーションの特性などに依存しない汎用的な不正クエリへの対策を行うことが求められる．このことから，Web アプリケーションの実装に依存せず汎用的に利用でき，かつ，現状のシステム構成や開発プロ



自動テストを用いた開発プロセス　　提案手法を組み込んだ開発プロセス

図 1　自動テストを用いた Web アプリケーションの開発プロセス

セスに影響の少ない不正クエリ対策が必要となる．

3　提案手法

2 章で述べた課題を解決するために，提案手法は以下の要件を満たす必要がある．

- Web アプリケーションの稼動前にホワイトリストを作成できる
- Web アプリケーションの実装に依存せずホワイトリストを作成できる
- 導入時の開発プロセスへの影響が小さい
- 導入時のシステム構成への影響が小さい

提案手法は，Web アプリケーションの動作テストをテストコードとして管理しており，テストが自動で実行される開発プロセスを想定する．また，テストコードは開発者が Web アプリケーションの変更に追従して更新する．提案手法は，開発プロセスへの影響を抑えつつ，Web アプリケーションの稼動後即時に不正クエリの検知を行うために，テスト時に発行するクエリを利用してホワイトリストを自動で作成する．また，提案手法は，導入時のシステム構成への影響を抑えつつ，Web アプリケーションの実装に依存せず，ホワイトリスト作成を行うために，データベースの前段にデータベースプロキシを配置しクエリを収集する．

3.1　自動テストを採用した開発プロセスにおける提案手法の位置付け

自動テストを採用した開発プロセスと提案手法の位置付けを図 1 に示す．

図 1 で示した自動テストを用いた開発プロセスについて説明する．まず，開発者は Web アプリケーションの新機能の開発や既存機能の修正を行う．次に，開発した機能が既存の他の機能に影響を与えている場合や，開発した機能に対してのテストが行われていない状態が発生している場合は，開発者はテストコードの記述を行う．ここで，テストコードには，Web アプリケーションをテストするときの実行手順であるテストケースと期待される動作結果を記述する．そして，自動テストの段階で，全てのテストコードを用いてテストを実行し，Web アプリケーションが仕様通りに動作しているかの検証を行う．このとき，テストが失敗した場合は，開発した機能の動作が仕様通りでない，もしくはテストコードの記述，すなわち仕様の定義に誤りがあることが分かる．この場合，開発者は原因を特定し，Web アプリケーションのソースコードもしくはテストコードの修正を行う．テストが成功した場合は，開発した機能が仕様通りに動作していることを確認できたとみなし，サーバに新しい Web アプリケーションのソースコードを配置し運用を開始する．

この開発プロセスの特徴は，開発された機能に対して，開発者が想定する Web アプリケーションの動作をテストコードに記述していくことであり，これにより Web アプリケーションの動作を保障する．しかし，テストコードに記述されているテストケースには漏れが生じる可能性があるため，機能が定義したテストケースに対して正常に動作していないことは分かるが，機能の動作に欠陥がないこと保証することはできない．このことから，テスト時に発行されるクエリは，開発者が想定した Web アプリケーションの動作の過程で発行されたクエリであり，開発者が想定できているクエリである．そのため，提案手法は Web アプリケーション稼動時における開発者の想定できていないクエリを検知することができる．

次に，提案手法を組み込んだ開発プロセスについて説明する．提案手法は自動テスト時に Web アプリケーションから発行されるクエリを用いてホワイトリストの作成を行う．そして，テスト成功後に，新しい Web アプリケーションのソースコードと共にそれに対応したホワイトリストをサーバに配置する．このようにすることで，Web アプリケーションが発行するクエリとホワイトリストの整合性を保つことができ，かつ，新しい Web アプリケーションの運用を開始するときには，不正クエリを即時に検知できる状態にすることができる．また，テスト時に自動でホワイトリストが作成されるため，導入時の既存の開発プロセスへの影響を低減することができる．

前述した自動テストを用いた開発プロセスにおいてテストコードを整備することと，ホワイトリストに手動でクエリ構造を登録し整備することの開発者への負担の差について述べる．テストコードには想定される Web アプリケーションの動作であるテストケースを記述していくのに対して，ホワイトリストには Web アプリケーションが動作の過程で発行するクエリ構造を定義していく．テストコードを書く場合，開発者は Web アプリケーションの動作を理解する必要がある．一方で，ホワイトリストを作成する場合，開発者は Web アプリケーションの動作を理解した上で，その過程で発行されるクエリ発行の動作を理解しなければならない．そのため，テストコードを整備することの方がホワイトリストの整備することに比べ，要求される Web アプリケーションの動作の知識が少ないため，開発者への負担が小さいと考えられる．また，提案手法には，テストを追加することによってホワイトリストの網羅性が向上する特性があるが，ホワイトリスト網羅性向上のための余分なテストを追加することは避けたい．そのため，開発者はホワイトリストの網羅性向上のためにテストを余分に追加するのではなく，提案手法によって検知されたクエリから必

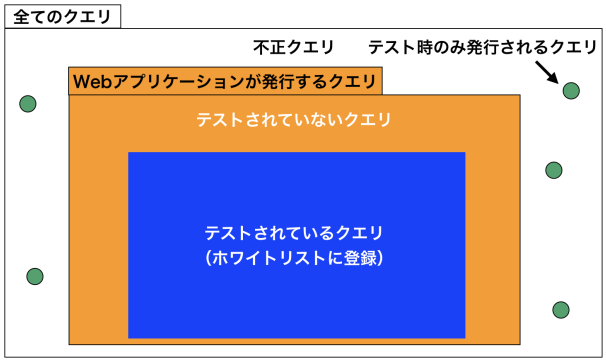


図 2　クエリの内包関係

表 2　提案手法で検知できないクエリの例

通常	SELECT * FROM users LIMIT 30
異常	SELECT * FROM users LIMIT 1000

要なテストの欠落を発見し追加を行う．

3.2　提案手法による検知の特性

提案手法による検知の特性について述べる．提案手法は，テスト時に，テストコードを元に Web アプリケーションを動作させ，その過程で発行されたクエリを用いてホワイトリストを作成している．そのため，提案手法によって作成されたホワイトリストで検知されるクエリは，テスト時に発行されなかったクエリとなる．このようなクエリにはテストされていないクエリと不正クエリが含まれる．テストされていないクエリは，テストコードを記述した開発者が想定できていないテストケースが Web アプリケーション稼働時に実行されたことによって生じ，不正クエリは，Web アプリケーションの脆弱性を利用した攻撃によって生じる．また，テスト時には，動作テストに用いるテストデータを登録するクエリや登録したテストデータを削除するクエリなどのテスト時にしか発行されないクエリが発行されるため，Web アプリケーションが通常稼働時に発行しないクエリがホワイトリストも登録される．図 2 に前述したクエリの内包関係の概略図を示す．

図 2 より，提案手法は，Web アプリケーションのテストカバレッジを向上させることによって，テストされているクエリ領域を拡大し，Web アプリケーションが発行するクエリ領域に近づけることができると考えられる．つまり，Web アプリケーションの改修に追従してテストが更新される開発プロセスにおいては，提案手法によって検知した開発者が想定できていないクエリを通知し対処することで，より正確な Web アプリケーションの動作保障ができ，かつ，不正クエリの検知精度を向上させることができると考えられる．

提案手法のホワイトリストにはクエリのリテラル部分をプレースホルダーに置き換えたクエリ構造を登録する．提案手法は，ホワイトリストに登録されていないクエリ構造をもつクエリを検知することができるが，クエリ構造は同じでリテラル部分が不正であるようなクエリを検知することはできない．例えば，表 2 のようなクエリの違いを検知することができない． LIMIT 句で指定されている数値リテラルは異なるが，このようなリテラル値の違いは提案手法では検知することができない．

そのため，このようなクエリに対しては，リテラル値の異常を検知するような方法を提案手法とは別に検討する必要がある．

3.3　提案手法の設計

提案手法は，テスト時と Web アプリケーション稼働時において，どちらも同じアーキテクチャを用いて，テスト時の Web アプリケーションが発行するクエリのホワイトリスト作成と稼働時の発行されるクエリの検知を行う．提案手法のテスト時のホワイトリスト作成フローと Web アプリケーション稼動時の検知フローを図 3 を用いて説明する．

テスト時のホワイトリスト作成フローを説明する．図 3 において，まず，テストが実行され Web アプリケーションからデータベースプロキシにクエリが発行される．データベースプロキシが受け取ったクエリは，テストの実行を妨げないようにするために，そのままデータベースに渡される．このとき，データベースプロキシは通過したクエリを記録しておく．全てのテスト終了後，データベースプロキシに記録されたクエリを用いて，クエリのリテラル部分をプレースホルダーに置き換えたクエリ構造に変換し，ホワイトリストに登録する．このように，データベースプロキシを用いてクエリの収集を行うことで，Web アプリケーションの実装に依存せず，ホワイトリストを作成することができる．

Web アプリケーション稼動時の検知フローについて説明する．図 3 において，まず，Web アプリケーションがユーザからの入力を受けクエリを発行する．次に，発行されたクエリをデータベースプロキシが受け取り，データベースプロキシは構文解析を行いクエリ構造に変換し，クエリ構造とホワイトリストの照合を行う．このとき，ホワイトリストに登録されていた場合は，Web アプリケーションから発行されたクエリをデータベースに渡し実行する．ホワイトリストに登録されていなかった場合は，開発者に検知したクエリを通知する．

Web アプリケーション稼動時の検知には，発行されたクエリとホワイトリストの照合処理が必要なため，Web アプリケーションとデータベース間のレイテンシーの増大が考えられる．ホワイトリストの照合処理が低速だった場合，データベースプロキシで発行されたクエリが停滞し，結果的に，Web アプリケーションのユーザへのレスポンス時間が増大することが懸念される．そのため，提案手法の実装時には発行されたクエリとホワイトリストの照合処理を考慮する必要がある．Web アプリケーションから発行されたクエリに対してホワイトリストを全探索した場合，ホワイトリストの照合処理の計算量はホワイトリストに登録されているクエリ構造の数 n に対して $O(n)$ となる．これを避けるために，ホワイトリストの作成時には，クエリ構造をキーとしてハッシュテーブルに登録しておき，ホワイトリストの照合処理を行うことで，探索の計算量は $O(1)$ に抑えられ，Web アプリケーションとデータベース間のレイテンシーの増大を抑えられると考えられる．

4　実験

Web アプリケーションの動作テストと提案手法によって作成されるホワイトリストの関係性を確認するために，図 4 に示す実験環境を構築し，テストカバレッジ

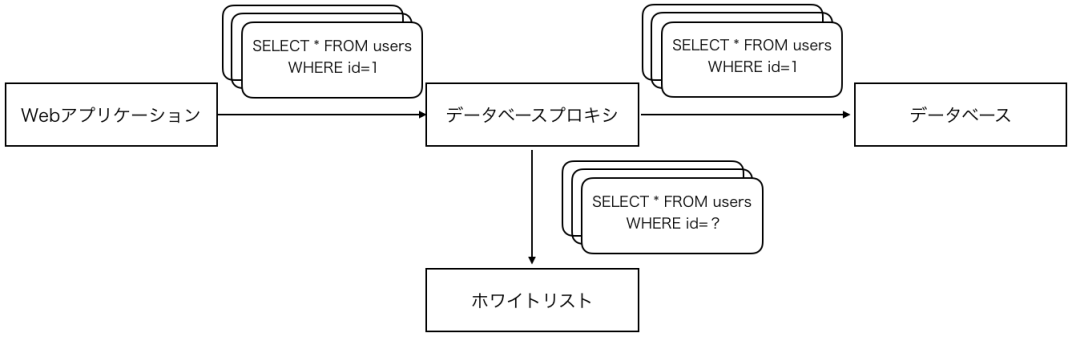


図 3　提案手法のアーキテクチャ

図 4　実験環境の構成

表 3　article テーブル

article テーブル
title
content

に対するホワイトリストの網羅率とテストに対して作成されたホワイトリストを確認した．ここで，ホワイトリストの網羅率とは，Web アプリケーションが発行する全てのクエリがホワイトリストに含まれる割合を示すものとする．

データベースプロキシとして ProxySQL[1)]を採用した．ProxySQL には，受け取ったクエリの収集しクエリ構造に変換して保存する機能があり，この機能をホワイトリストの作成に利用した．また，データベースには ProxySQL がサポートしている MySQL を採用し，表 3 のように article テーブルに title カラムと content カラムを作成した．Web アプリケーションには Ruby on Rails を採用し，表 4 のように article テーブルに CRUD の操作を行うメソッドを実装し，それぞれのメソッドに対して動作テストを記述した．

4.1　テストカバレッジに対するホワイトリストの網羅率

Rails アプリケーションのテストカバレッジの測定には，SimpleCov[2)]を利用し，行カバレッジを測定した．ホワイトリストの網羅率を算出するために，手動で Rails アプリケーションに対して HTTP リクエストを送り全ての発行されるクエリを取得した．また，ホワイトリストはテストを実行し，ProxySQL が収集したクエリのクエリ構造を用いて作成した．

実験方法として，Rails アプリケーションに実装した CREATE メソッドのテストを削除し，削除前後のテストカバレッジとホワイトリストの網羅率の比較を行った．実験結果を表 5 に示す．

実験結果より，CREATE メソッドのテストを削除したことによって，ホワイトリストの網羅率は低下し，それに伴ってテストカバレッジも低下していることがわかる．このことから，テストカバレッジとホワイトリストの網羅率には関係があり，提案手法はテストカバレッジを向上させることによってホワイトリストの網羅率を向上することが確認できた．また，テスト削除後にホワイトリストに含まれていなかったクエリは，article をデータベースに追加するための INSERT 文のクエリであった．これは，提案手法によって INSERT 文のクエリが Web アプリケーション稼働時に検知されることを意味し，これによって開発者は動作テストの欠如を認知することができ，原因の特定を行うことができる．提案手法は，開発者の想定漏れの動作の過程で発行されたクエリを検知し，開発者に通知することで，より正確な Web アプリケーションの動作保障をすることができ，Web アプリケーションの脆弱性の早期発見に繋がると考えられる．

1） http://www.proxysql.com/
2） https://github.com/colszowka/simplecov

表 4　Rails の実装メソッド一覧

HTTP	URL	操作内容
GET	articles	全ての article を表示
POST	articles	article を一つ作成
GET	articles/:id	特定の article を表示
PATCH	articles/:id	特定の article を更新
DELETE	articles/:id	特定の article を削除

表 5　テストカバレッジに対するホワイトリストの網羅率

状態	テストカバレッジ	ホワイトリストの網羅率
テスト削除前	88.24%	81.25%
テスト削除後	73.53%	75%

4.2　テストに対して作成されたホワイトリスト

4.1 節で述べたテスト削除前の状態において，ホワイトリストに含まれなかったクエリを確認した．ホワイトリストに含まれなかったクエリを以下に示す．

- UPDATE ‘articles‘ SET ‘content‘ = ?, ‘updated_at‘ = ? WHERE ‘articles‘.‘id‘ = ?
- UPDATE ‘articles‘ SET ‘title‘ = ?, ‘updated_at‘ = ? WHERE ‘articles‘.‘id‘ = ?
- UPDATE ‘articles‘ SET ‘title‘ = ?, ‘content‘ = ?, ‘updated_at‘ = ? WHERE ‘articles‘.‘id‘ = ?

ホワイトリストに含まれていなかったクエリは全て article の情報を更新するためのものであった．しかし，表 4 における PATCH リクエリストを送った時に動作する UPDATE メソッドに対してのテストは記述されており，テストカバレッジの算出にも含まれていた．

UPDATE メソッドのテストは，「PATCH リクエストを送り更新成功後，特定の URL にリダイレクトされるか」という内容で，データベース上の更新前の article データと更新しようとしている article データが同一であった．更新前のデータと更新しようとしているデータが同一の場合，Rails アプリケーションは UPDATE のクエリを発行しないが，article を更新する動作は成功したとみなされるため，テストは成功し UPDATE メソッドのテストがテストカバレッジの算出に含められる．これらのことから，テスト内容によっては実行の過程でクエリが発行されない場合があるため，テストカバレッジ 100%を実現しても完全に網羅したホワイトリストを作成することができないことがわかった．しかし，提案手法は，テストカバレッジで認知することができなかったテストケースの漏れを，クエリレベルで検知し開発者に通知できることがわかった．そのため，提案手法は Web アプリケーションの開発において，開発者が想定できていない動作を検知できる点において有効であると考えられる．

5 まとめ

本稿では，Web アプリケーションが利用するデータベースに対して，通常稼働時に発行されない不正クエリの実行を検知するために，開発プロセスのテストに着目し，テスト時に発行されるクエリを用いて Web アプリケーションが発行するクエリのホワイトリストを自動で作成する手法を提案し，実験により，提案手法の検知の特性を評価した．提案手法は，開発者が開発時点で想定できなかったテストケースの漏れを，クエリレベルで検知し，Web アプリケーションの脆弱性の長期化を防ぐことができる．

実験から，テストカバレッジ向上によるホワイトリストの網羅率が向上すること，テストカバレッジ 100%を実現が Web アプリケーションが発行するクエリの全網羅を意味しないという 2 つの結果を得た．実験結果から，提案手法は，テストカバレッジで認知することができなかったテストケースの漏れを開発者に指摘できる点において，有効性があることを確認した．

今後は，提案手法のホワイトリスト作成と検知の実装を進め，提案手法の導入による Web アプリケーションとデータベース間のレイテンシーへの影響の検証を行う．今後の課題としては，データベースにテストデータを登録するもしくは登録したテストデータを削除するようなテスト時のみに発行されるクエリへの対処方法の検討が挙げられる．

参考文献

[1] D Kar, S Panigrahi, Prevention of SQL Injection attack using query transformation and hashing, Advance Computing Conference (IACC), 2013.

[2] F. Valeur, D. Mutz and G. Vigna, A Learning-Based Approach to the Detection of SQL Attacks, In Proceedings of the Conference on Detection of Intrusions and Malware Vulnerability Assessment (DIMVA), July 2005.

[3] K Kemalis, T Tzouramanis, SQL-IDS: a specification-based approach for SQL-injection detection, Proceedings of the 2008 ACM symposium on Applied computing, Pages 2153-2158, March, 2008.

[4] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, Randy Stafford, Patterns of Enterprise Application Architecture, Addison-Wesley, 2002.

[5] M Bachle, P Kirchberg, Ruby on rails, 2007.

[6] Rietta, Frank S., ”Application layer intrusion detection for SQL injection.”, Proceedings of the 44th annual Southeast regional conference. ACM, 2006.

[7] Scott W. Ambler, Mapping objects to relational databases What you need to know and why Ronin International, July 2000.

[8] William G.J. Halfond and Alessandro Orso, AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks, In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE ’05), ACM, New York, NY, USA, 174-183, 2005.

[9] IPA, 安全なウェブサイトの作り方 改訂第 7 版, 2015 年, 3 月.

[10] 青井 翔平, 坂本 一憲, 鷲崎 弘宜, 深澤 良彰, DePoT:Web アプリケーションテストにおけるテストコード自動生成テスティングフレームワーク, 情報処理学会論文誌 Vol.56 No.3 835–846, March, 2015.

[11] 情報セキュリティインシデントに関する調査報告書 個人情報漏洩編 第 1.2 版, NPO 日本セキュリティ協会セキュリティ被害調査ワーキンググループ, 長崎県立大学情報システム学部情報セキュリティ学科, 2017 年 6 月.

[12] 藤田 直行, 侵入検知に関する誤検知低減の研究動向, 電子情報通信学会論文誌 B Vol.J89–B No.4 pp.402–411, 2006.