情報処理学会論文誌 Vol. 50 No. 9 2048–2061 (Sep. 2009)

ホワイトリストコーディングによるSQLインジェクション攻撃耐性保証方法と実装

渡 邉 悠[†1] 松 浦 幹 太[†1]

Web アプリケーションの安全性が重要となっている昨今において，Web アプリケーションに脆弱性が存在しないことを開発者任せにせずにいかに保証するかは重大な問題となっている．本論文では，安全なコーディング方法を開発者に対して提供しておき検証時にプログラムがそのコーディング方法のみを利用して開発されていることを技術的に保証することでソフトウェアの安全性を保証するという考え方を提案する．そして，その考え方に基づいて Web アプリケーションの SQL インジェクション攻撃に対する安全性を保証する方法の提案を行う．また我々が PostgreSQL を拡張して実際に作成した提案手法の実装について紹介を行う．

Assured Resistance against SQL-injection Attacks: A Whitelist-coding Approach and Implementation

Yu Watanabe[†1] and Kanta Matsuura[†1]

Because importance of information managed by web applications increases, it is very important to assure security of web applications without relying on ability of programmers. In this paper, we propose a new approach to assure that there are no SQL injection vulnerabilities in a web application and introduce an implementation of our method implemented using PostgreSQL. The concept of our approach is that we can assure security of web applications by allowing programmers to write programs using only predetermined secure coding methods.

†1 東京大学生産技術研究所
Institute of Industrial Science, the University of Tokyo

1. はじめに

1.1 Web アプリケーションと脆弱性

ソフトウェアによって管理される情報の重要性が増大しているのにともない，ソフトウェアによって管理されている情報の流出や改竄が起こると大きな問題となるようになっている．そのためソフトウェアの安全性の鍵となる脆弱性対策が非常に重要になっている．またインターネットの普及と高速化によって，Web アプリケーションとしてネットワークを通じてユーザに提供されるソフトウェアの利用が急速に広まっている．そのため脆弱性対策の中でも特に Web アプリケーションを対象とした脆弱性対策がより重要になってくると考えられる．また Web アプリケーションの脆弱性の報告数の 1 位と 2 位はクロスサイトスクリプティング（XSS）と SQL インジェクション攻撃（SQLIA）であり，この 2 つの脆弱性だけで Web アプリケーションの脆弱性の過半数を占めているといわれている[1),2)]．本論文では特に SQLIA に注目し，SQLIA に対して脆弱な Web アプリケーションの開発を防止するための方法と技術の提案を行う．

1.2 SQL インジェクション攻撃

SQL は Web アプリケーションを構築する際に広く利用されているリレーショナルデータベース管理システム（RDBMS）に対して問合せを行うための言語であり，RDBMS を利用する Web アプリケーションの内部ではユーザ入力値に基づいて SQL クエリの構築が行われる．そして，そのクエリの構築処理にバグが存在すると開発者が意図していない方法でユーザが SQL クエリを改変することが可能となりアプリケーションに脆弱性が生じてしまうことがある．これが一般に SQL インジェクション攻撃に対する脆弱性といわれるものである．SQLIA に対して脆弱なコードの具体例として，データベースに格納されたユーザ情報を利用してユーザ認証を行う関数の疑似コードを示す（**図 1** 左）．この関数ではユーザ情報が格納されたテーブル users 上の ID とパスワードがユーザ入力値と一致する行の数をカウントすることでユーザ認証が行うものであり，この関数を定義したプログラマは，たとえば ID に “`yunabe`”，パスワードに “`pass`” という文字が入力された場合に生成される SQL の条件部分が `id = 'yunabe' and password = 'pass'` となりユーザ認証が正しく行われると想定している．しかしユーザが入力したパスワードが “`' or 'a' = 'a`” であると生成される SQL の条件部分は `id = 'yunabe' and password = '' or 'a' = 'a'` というプログラマの想定に反してつねに true となる論理式になってしまう．その結果 SQL を評価した結果は 0 ではなくなるため，このユーザは `yunabe` のパスワードを知らないにもかかわら

© 2009 Information Processing Society of Japan

```
public bool Authenticate(string id, string password)
{
    string query = "select count(*) from users where id = '" + id
                + "' and password = '" + password + "'";
    using (SqlConnection conn = new SqlConnection(connectionString))
    using (SqlCommand cmd = new SqlCommand(query, conn))
    {
        conn.Open();
        int result = (int)cmd.ExecuteScalar();
        return result != 0;
    }
}
```

```
public bool Authenticate(string id, string password)
{
    string query = "select count(*) from users where "
                + "id = @id and password = @pass";
    using (SqlConnection conn = new SqlConnection(connectionString))
    using (SqlCommand cmd = new SqlCommand(query, conn))
    {
        cmd.Parameters.Add("@id", id);
        cmd.Parameters.Add("@pass", password);
        conn.Open();
        int result = (int)cmd.ExecuteScalar();
        return result != 0;
    }
}
```

図 1　SQLIA に対して脆弱なコードとバインド機構を用いた安全なコードの例
Fig. 1　An examples of code vulnerable for SQLIA and secure code with bind variables.

ず yunabe として認証されてしまうことになる．SQLIA の方法は，例であげた条件式をつねに true になる式に改竄する方法以外にもいくつも存在するが[3]，いずれの攻撃方法もアプリケーション中の SQL クエリの構築処理のバグを悪用するという点が共通している．また SQLIA に関して議論を行う際，Web サーバ上で動作するプログラムについてのみ注目されるされることが多いが，ストアドプロシージャと呼ばれる RDBMS 上で動作するプログラムに SQLIA に対する脆弱性が生じる場合もあるため注意が必要である[4]．

2. 既存の対策技術

2.1　安全なコーディング

プログラムがユーザ入力値の検証および変換を正しく行わずにユーザ入力値を SQL に組み込んでしまうことが，SQLIA に対する脆弱性が生じてしまう根本原因である．そのため，検証と変換を正しく行ったうえでユーザ入力値を SQL に組み込むようにプログラムを作成すれば脆弱性を防ぐことができる．たとえばユーザが入力した整数を SQL に埋め込む場合には入力値が確かに整数であることを確認したうえで SQL に埋め込めばよい．またユーザが入力した文字列を SQL に文字列として埋め込む場合には入力値を SQL 上で文字列リテラルとして扱われる文字列に変換したうえで SQL に埋め込むようにすれば SQLIA に対する脆弱性は生じない．このように入力値の適切な検証と変換を行うことで SQLIA に対して安全なプログラムを記述する方法は複数存在するが，その中でも JDBC や ADO.NET などで標準的にサポートされており，実際に広く用いられている手法にバインド機構を利用した開発方法がある．これはユーザ入力値によって決定されるリテラル部分を値が未割当てであることを表すプレースホルダと呼ばれる記号で代用した SQL クエリの雛型を用意しておき，データベース接続ライブラリにこの雛型と雛型の中のプレースホルダにバインドする値を渡すことで SQL を RDBMS に実行させる方法である（図 1 右）．この手法ではデータベース接続ライブラリが受け取った値の型に合わせて適切な変換を行い SQL を構築する．そのためデータベース接続ライブラリの実装に脆弱性が存在しなければプログラマは SQLIA を防ぐために必要な検証・変換処理について精通していなくても図 1 右のようにプログラムを書くことで SQLIA に対して安全なプログラムが記述できる．また，バインド機構とは異なるアプローチでプログラマに対して SQLIA に対して安全なプログラムを記述する方法を提供する研究も存在している[5)–8)]．

確かにこうした手法を開発者が正しく利用すれば SQLIA に対する脆弱性の存在しないアプリケーションを作成することが可能である．しかしながら，プログラマが正しい利用方法を理解していなかったり何らかのミスによって利用方法を誤る，もしくはまったく利用しなかったりした場合，当然のことではあるがこうした手法は脆弱性の防止手段としてまったく機能しない．そのためこうした手法のみでアプリケーションの安全性を保証しようした場合，開発時にコード規約を定めるなどの運用面でそうした欠点をカバーしない限りはアプリケーションの安全性の保証が完全にプログラマ任せになってしまうということを意味しており，単一の脆弱性技術として十分なものであるとはいえない．

2.2　攻撃の検出

そのためプログラマがプログラムに SQLIA に対する脆弱性を作り込んでしまった場合にもアプリケーション全体の安全性が保たれるようにするための技術が必要となる．そうした技術の 1 つとして，プログラムの外部に攻撃を検出するシステム（IDS，WAF）を設置し，ユーザ入力をフィルタリングして攻撃を排除することでプログラムを SQLIA から保護する手法があげられる．ただし誤検出・検出漏れをまったく起こさないフィルタリング条件を定義することは難しく，しかも何が攻撃として扱われ何が正常な入力として扱われるべきかは保護の対象となるプログラムの仕様によって決定されるため，フィルタリングの条件をプログラムごとに定義しなければならない．そのため，こうしたシステムのみで完璧に SQLIA を防ぐのは難しい．より進んだ攻撃の検出手法としてプログラムの静的解析と動的解析を組み合わせた手法が研究として提案されている[9)–11)]．こうした手法はプログラムを解析することで攻撃を検出するためのルールを自動的に生成し，生成された攻撃検出ルールを利用して実行時に攻撃の検出を行う．そのためプログラムごとにフィルタリング条件を人手で定義するという作業が不要となる．またプログラムの静的解析が正確に行うことができ適切な攻撃検出のルールを生成できる場合には，正確に攻撃を検出することが可能となる．しかし実際にはあらゆる種類のコードを正確に静的解析することは困難なため，アプリケー

© 2009 Information Processing Society of Japan

ションの構造によっては適切な攻撃検出のルールが生成できず，攻撃の誤検出や検出漏れが生じてしまう危険性がある[3]．

2.3 脆弱性の検出

安全性の保証をプログラマ任せにせずに保証するためのもう 1 つの技術として脆弱性検出技術があげられる．プログラマが脆弱性をプログラムに埋め込んでしまったとしても，攻撃を受ける前に脆弱性検出技術を用いて脆弱性を発見し除去すれば問題がないため脆弱性検出技術は安全なプログラムを作成するうえで重要な技術である．また脆弱性検出技術はプログラムに対して実際に攻撃を行うことで脆弱性を検出するブラックボックス的な手法とプログラムに対して静的解析や動的解析を行うことで脆弱性を発見するホワイトボックス的な手法とに大別することができる．ホワイトボックス的な手法としては Perl や Ruby で利用可能な汚染検出モード[12] が有名である．ただしブラックボックス的な手法は実際に攻撃を行って脆弱性を検出するという技術の性質上，すべての脆弱性が検出されていることを保証することは難しい．またホワイトボックス的な手法は脆弱性を誘発しやすい典型的な危険なコードの書き方を検出することで脆弱性を検出するため，典型的なパターンから外れた書き方をされたコードの中に含まれる脆弱性を検出することは難しく，やはり脆弱性の検出漏れが問題となる[3]．

3. 提 案 手 法

3.1 モチベーション

2 章で述べた技術はいずれも SQLIA 対策として有益な技術である．しかしながら，2.1 節で述べた安全なプログラムを記述するための技術のみにアプリケーションの安全性が依存していると，安全性の保証がプログラマ任せになってしまうという問題が生じる．一方で安全性の保証がプログラマ任せになる事態を防ぐために 2.2 節で述べた攻撃検出技術や 2.3 節で述べた脆弱性検出技術を利用する場合には，攻撃・脆弱性の検出精度やプログラムの静的解析の精度の限界による検出漏れや誤検出が問題となる．もちろんこうした問題はプログラマに対する教育を徹底することや精度の高い検出技術を研究開発することによって緩和することが可能であり，実際に何人もの研究者によって研究が行われている．しかしこうした問題を改善するための取り組みが研究として成立することが示すように，プログラマのミスや検出漏れ・誤検出はそれぞれの対策技術が本質的にかかえる問題であり，こうした問題を完璧に防ぐことは難しい．そこで今回我々は攻撃の検出や脆弱性の検出によってアプリケーションの SQLIA に対する安全性を確保するという手法とはまったく異なる考え方に基づいており，かつアプリケーションの安全性をプログラマ任せにせず保証することが可能な新しい脆弱性対策手法の提案を行う．

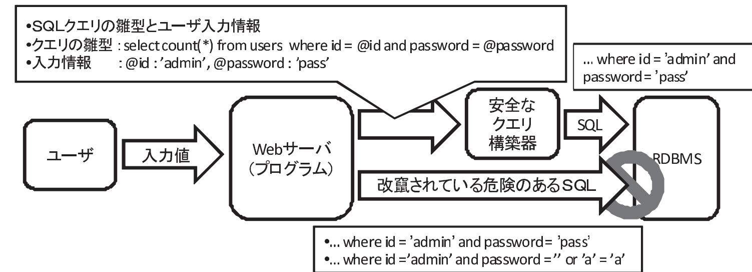

図 2　本提案手法における Web アプリケーションの構造
Fig. 2　Structure of web applications in our proposal.

3.2 概　　要

多くの Web アプリケーションの内部で，ユーザ入力値に基づいて SQL の作成を行いそれを RDBMS に対して発行するという処理が行われる．そしてその際に必要となる SQL の中にユーザ入力値を埋め込む処理は間違いを誘発しやすい処理であるうえに，間違いを犯した場合にアプリケーションに脆弱性をもたらしてしまう非常に危険な処理である．それにもかかわらずユーザ入力値の埋め込み処理の実装が個々のプログラマに任せられてしまっているため，多くのプログラマが実際に間違いを犯し脆弱なアプリケーションを開発してしまっているのが現状である．そこで我々の提案手法では，安全に SQL を構築することが可能なコンポーネントをプログラムの外部に用意し，プログラマが SQL を構築する処理をその外部のコンポーネントに委ねることができるようにシステムを設計する．また単に SQL を安全に構築するためのコンポーネントをプログラマに対して提供するだけではなく，このコンポーネントを通じて安全に SQL を構築しない限りは RDBMS に対して SQL を発行することができないようにシステムを設計し，プログラムの内部では SQL の構築が行えないようにする．

システムの構成の概要を 1.2 節であげた ID とパスワードをユーザから受け取り ID とパスワードが一致するデータの件数を取得する SQL を作成・実行することでユーザ認証を行うアプリケーションを具体例として用いて説明する（**図 2**）．まず我々の提案手法では RDBMS を拡張し，ユーザ入力値から安全に SQL を構築するための機能を RDBMS に追加する．具体的には，SQL クエリの雛型と，雛型から SQL クエリを作成するために必要な情報をクライアントから受け取り，SQL クエリを作成，それを SQL クエリの評価器に受

© 2009 Information Processing Society of Japan

け渡し実行する機能を RDBMS に追加する（図 2 上部）．それと同時に，プログラムの内部で独自の処理によってユーザ入力値から動的に生成された SQL が RDBMS で実行させることを防止するために，SQL クエリを直接受け取って評価・実行する機能がプログラムから利用不可能なようにシステムを設計する（図 2 下部）．このようにシステムを設計することにより，プログラマは 1.2 節で脆弱性の原因となった ID とパスワードを SQL に埋め込む処理を RDBMS が持つ安全に SQL を構築するための機能に委託することが可能となり，SQL の生成という危険な処理をプログラムに実装しないで済むようになる．またプログラム内部で生成した SQL を RDBMS に実行させることが不可能なようにシステムが設計されているため，プログラマは RDBMS が提供する機能を利用して SQL を構築するようにプログラムを記述せざるをえなくなる．そのため SQL の生成という危険な処理がプログラムから強制的に排除されることになり，それにともなって SQLIA に対する脆弱性もプログラムから排除されることとなる．以上が我々が本論文で提案する，SQLIA に対して安全な Web アプリケーションを構成するためのアプリケーション構築手法の概観である．また，技術的に明確な形で定義可能な安全なコーディング方法をあらかじめ定めておき，プログラマにそのコーディング方法を利用してプログラムを記述することのみを許すことでアプリケーションの安全性を保証するこの方法を，我々はホワイトリストコーディングと呼ぶことにする．ただしこの手法を用いて SQLIA に対する脆弱性を防ぐためには

- クエリの雛型に何を利用するのか．どういった処理でクエリの雛型から実際に実行するクエリを生成するのか，
- クエリの雛型に対してインジェクション攻撃が起こり雛型が攻撃者に改竄されてしまうと，いかにクエリの雛型からクエリを生成する処理が SQLIA に対して安全であったとしても何の意味もない．そのためクエリの雛型がアプリケーション開発時に定義された静的なものであり，攻撃者による改竄を受けていないということを保証する必要がある，
- プログラム内部で生成された SQL を RDBMS に渡して実行させる処理の禁止をどのように実現するか，

などの課題が存在する．そこで 4 章ではこうした課題が存在することをふまえたうえで，本提案手法ではどのように RDBMS を拡張するのかについて詳細を述べる．また本提案手法ではプログラムが RDBMS に対して SQL 実行の要求を行う方法を大きく変えることになるため，既存のライブラリとコーディング手法を本提案手法に合わせて拡張しなくてはならない．このクライアントサイドでのライブラリの拡張およびコーディング手法の拡張については 5 章で言及する．

4. サーバサイド（RDBMS）の構成

4.1 SQL クエリの雛型と SQL クエリの構築

本提案手法では SQL クエリを安全に構築する手段を利用することで SQLIA に対して安全なプログラムを記述し，その利用をプログラマに対して強制付けることで安全性の保証を行う．そのため SQL クエリを安全に構築する処理をどのように設計するかが重要となる．もしもクエリの雛型から実際に利用したいクエリを作成する処理の自由度が大きすぎると，攻撃者に対して任意のクエリ改竄を許す可能性を与えてしまうことになる．一方でクエリ作成の処理の自由度が小さすぎると Web アプリケーションを開発するのに不可欠なクエリの動的生成まで禁止してしまうことになり，アプリケーションの開発の妨げになってしまう．そのため雛型からクエリを作成する処理の自由度は Web アプリケーションを開発するのに必要十分なものでなくてはならない．

Web アプリケーションに必要な処理　アプリケーションがデータ永続化を行う場合には，一般にデータの生成・読み取り・更新・削除の 4 つの基本機能の実装が必要となる．この 4 つの機能は Create，Read，Update，Delete の頭文字をとって CRUD と呼ばれ，データ永続化に RDBMS を利用する場合にはそれぞれの処理が SQL の INSERT 文・SELECT 文・UPDATE 文・DELETE 文を通じて行われる．そのためデータの生成・読み取り・更新・削除を行う際に，アプリケーション内部ではそれぞれの処理に対応する INSERT 文・SELECT 文・UPDATE 文・DELETE 文の組み立てが行われることになる．またデータを作成する際には作成するデータの値が，データを読み取る際には読み取るデータを指定するための検索条件と読み取ったデータをソートする順序が，データを更新する際には更新するデータを指定するための検索条件と新しい値が，データを削除する際には削除するデータを指定するための検索条件がユーザの入力情報に基づいて実行時に決定できる必要がある（**表 1**）．さらに検索条件を実行時に決定する場合でも任意の検索条件を実行時に定めることができる必要はなく，現実の Web アプリケーションで実行時に制御できる必要がある検索条件の要素は，検索式内の値（リテラル）・検索条件の有無・検索条件の繰返しのみに限定できると考えられる．そこで本提案手法では，Web アプリケーションで実行時に決定できる必要のある SQL の要素はリテラル・条件の有無・条件の繰返し・ソート順序に限定することができると仮定し，クエリの雛型ではこれらの要素のみを未定にしておき，クエリ構築処理でユーザ入力値に基づく情報から決定することができるように機能の設計を行う．なおこの仮定の妥当性の検証と，この仮定が成立しない場合に本提案手法にどのような影響

© 2009 Information Processing Society of Japan

表 1　Web アプリケーションで必要となるデータ操作と SQL 文の関係
Table 1　Relationships between data manipulation and SQL statements in web applications.

機能	SQL 文	動的な要素
Create	INSERT 文	挿入する値
Read	SELECT 文	検索条件・ソート順序
Update	UPDATE 文	検索条件・新しい値
Delete	DELETE 文	検索条件

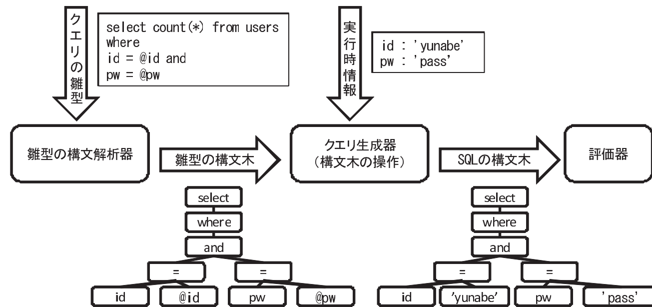

図 3　クエリの雛型からクエリを生成する処理の概要
Fig. 3　Overview of query generation.

が及ぶかについての考察を 8.2 節で行う[*1]．次に上の議論をふまえたうえでクエリ構築処理の具体的な構成方法について述べる．

詳細　SQL を拡張して上述の実行時に決定できる必要のある要素が未定のまま記述できる言語を作成し，それを SQL クエリの雛型を表現するための言語として利用する．SQL を構築する際には，まず拡張された SQL の構文解析器を使って SQL クエリの雛型を構文木に変換する．そして雛型からクエリを生成するためのユーザ入力に基づく動的な情報を利用して雛型の構文木を操作することで，生成したい SQL クエリの構文木を作成し，それを SQL の評価器に渡して評価を行う（**図 3**）．

(1)　**バインディング**

すでに 3.2 節の図 2 で例示しているように，雛型から SQL の構築を行う際に最も基本となるリテラルの決定処理には 2.1 節で紹介したバインド機構の考え方を流用する．すなわち実行時に決定されるべきリテラルについては SQL の雛型の中では，値が未定であることを表す「プレースホルダ」にしておき，プレースホルダにバインドされるリテラルを実行時に定義することで RDBMS 上で実行される SQL を作成する．なお下記で述べる特殊なケースを除き，プレースホルダは SQL の変数参照と同じ位置にのみに出現するように雛型の言語の文法を定義し，たとえば SQL の文法上文字列リテラルの指定しか許されていない部分にプレースホルダを記述することはできないようにしなければならない．また SQL の雛型上でプレースホルダと SQL の変数参照が紛れること防ぐため，雛型の字句解析器がプレースホルダと SQL の変数参照を異なる種類のトークンに区別できるようにプレースホルダの表現を選択する．なお，プレースホルダにバインドすることが可能なものは SQL 上の定数もしくは変数参照のみである．そのためたとえば `select * from users where @cond` という SQL の雛型の `@cond` に対して `id = 'yunabe' and pw = 'pass'` という式をバインドして `select * from users where id = 'yunabe' and pw = 'pass'` という SQL を作成する処理は許されず，そうした入力が `@cond` にバインドされた場合エラーとなり SQL の生成・実行は行われない．

(2)　**Optional Condition**

実行時に有無が定まる検索条件（optional condition）をサポートするために，SQL クエリの雛型から SQL クエリを作成する際に条件式の一部を除去することができるようにする．そのために雛型の言語を拡張し条件式に `OPTIONAL` という特殊な演算を追加する．この演算は `OPTIONAL`(*expr*, *is_enabled*) という形式をしており，第 1 引数 *expr* で条件式を指定し第 2 引数 *is_enabled* でその条件式を有効にするかを指定する．そして雛型からクエリを作成する際に，*is_enabled* が true の場合には `OPTIONAL` 演算は *expr* に変換され *is_enabled* が false の場合には *removed* という特殊値に変換される．この *removed* というのは式の一部が除去されたことを表すための特殊値で，*removed* を子要素に持つ構文木ノードは，クエリ生成処理の中で

- 二項演算 AND/OR を表す構文木ノードの子要素のいずれかが *removed* である場合，その構文木ノードは *removed* ではないほうの子要素の構文木ノードに変換される．両方の子要素が *removed* である場合には，構文木ノードは *removed* に変換される，
- AND/OR 演算以外の構文木ノードの子要素のいずれかが *removed* である場合，構文木ノードは *removed* に変換される，

という規則に従って変換される．また `OPTIONAL` 演算の糖衣構文としてキーワード `OPTIONAL`

---

*1 CRUD 処理を基本としないデータベース管理用のツールはこの仮定にあてはまらないが，そうしたアプリケーションは SQLIA 以前にそもそもアプリケーションが攻撃者のアクセスを受けること自体が問題外であるため，本提案手法ではそうしたアプリケーションは対象外としている．

© 2009 Information Processing Society of Japan

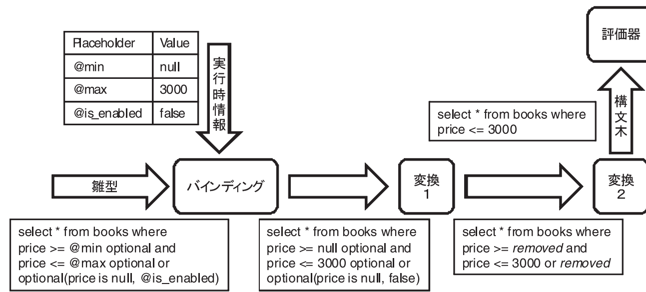

図 4　実行時に有無が決定される条件式を含む構文木の変換処理
Fig. 4　Conversion of a parse tree that include optional conditions.

で修飾されたプレースホルダを導入する．これはプレースホルダをキーワード OPTIONAL で修飾できるようにし，OPTIONAL で修飾されたプレースホルダに NULL がバインドされた場合にはプレースホルダを NULL ではなく *removed* に変換するという機能である．**図 4** に本を価格の上限と下限を指定して検索する Web アプリケーションで SQL を実行する場合の例をあげる．この Web アプリケーションでは価格の上限と下限のどちらか一方のみを指定して検索することが可能であり，また検索結果に価格情報が登録されていない（列 price に null が格納されている）本の情報を含めるかどうかを選択することが可能である．そのため図 4 に示すように，価格の上限と下限に対応する@max，@min というプレースホルダが OPTIONAL で修飾されており，条件式 price is null が OPTIONAL 演算の子要素となっているクエリの雛型が利用される．そしてたとえばユーザが価格が 3,000 円以下の本を検索した際には，価格の上限が 3,000 なので@max に 3,000 が，価格の下限は検索条件に含まれないので@min には NULL が，価格情報を持たない本は検索結果には含めないので@is_enabled には false がバインドされる．そして null optional と optional(price is null, false) が *removed* に変換され，さらに *removed* を子要素に持つ構文木ノードが上述した変換規則で変換され，最終的に価格が 3,000 円以下の本を検索するクエリ select * from books where price <= 3000 が作成される．なお条件式全体が *removed* となってしまった場合，つまり検索条件が 1 つもない場合，条件式を真と見なすべきか偽と見なすべきか判断することができないのでエラーとなりクエリの生成に失敗する．

(3)　**条件の繰返し**

条件の繰返しをサポートするために，クエリの雛型の言語に MAPREDUCE という特殊な演算を追加する．この演算は MAPREDUCE(*op*, *expr* FOR *placeholder* IN *array*) という形式をしており，クエリ生成時に *expr* 中のプレースホルダ *placeholder* に対して配列 *array* の各要素をバインド（map）した式の配列を二項演算子 *op* で結合（reduce）したものに変換される．たとえばユーザが入力した複数のキーワードのいずれかに一致するデータを検索するアプリケーションを作成する場合には，まず SQL クエリの雛型の WHERE 節を where mapreduce(or, keyword = @k for @k in @keywords) とする．そしてユーザが “security” と “privacy” というキーワードを選択して検索した際には，@keywords に列 ['security', 'privacy'] がバインドされ，生成される SQL クエリの WHERE 節は where keyword = 'security' or keyword = 'privacy' となる．なお *array* にバインド可能なのは定数の列もしくは変数参照のみであり，定数もしくは変数参照以外を含む配列がバインドされた場合エラーとなり処理が中断する．また *array* に指定された列の要素数が 0 の場合には MAPREDUCE 演算は *removed* に変換され，前項の *removed* の変換規則に従って条件式から除去される．

(4)　**ソート順序**

データ取得時のソート順序の動的な変更をサポートするためにクエリの雛型の言語の ORDER BY 節の文法を拡張して，ORDER BY *placeholder* という形式でソート式の本体をプレースホルダにすることができるようにする．そしてクエリ生成時には *placeholder* にソート式の本体をバインドすることで実行時にソート順序を制御できるようにする．ただし，SQL 自体はソート式として任意の式に ASC もしくは DESC を付与したものの列（*expr* [ASC|DESC])+を受理するが，バインドされるソート式に任意の式 *expr* が指定可能であると，攻撃者が *expr* を改竄することでソート順序の制御以上のことが行えてしまい脆弱性が生じてしまう可能性がある．またソート方法をユーザの入力に従って実行時に制御する必要があるアプリケーションというのは実際には，大量のデータを表示する必要がありその際にどのデータ列でデータをソートするかをユーザが選択可能なアプリケーションに限られるため，実行時に任意の式がソート式として利用できる必要はまったくない．そこでプレースホルダにバインドされるソート式には任意の式に ASC もしくは DESC 付与したものは許さず，列の参照に ASC もしくは DESC を付与した *column_ref* [ASC|DESC] のみを許可するようにする．また長さが 0 のソート式の列がバインドされた場合には ORDER BY 節が除去されたクエリが生成されるように雛型からクエリを生成する規則を定める．たとえば select * from users order by @order という雛型の@order に対して [birthday ASC, name DESC] がバインドされると select * from users order by birthday ASC, name DESC が生成される．また@order に対して空列がバインドされた場合には select * from users が生成される．

© 2009 Information Processing Society of Japan

4.2　安全に SQL クエリを構築する機能

本提案手法ではプログラマは，RDBMS が従来持っている SQL クエリを受け取り評価する機能の代わりに，SQL クエリの雛型と雛型からクエリを作成するための情報を受け取り SQL クエリを作成・評価する機能を利用してプログラムを記述する．そのため本提案手法を導入するには，RDBMS に存在する文字列をクエリとして受け取り評価する機能をすべて洗い出し，それぞれの機能を SQL クエリの雛型と雛型に含まれるプレースホルダにバインドする値の情報を受け取りクエリを作成・評価するものに拡張しなくてはならない．多くの RDBMS はこの拡張の対象となる機能を複数持っているが，こうした機能は大きく 2 つのグループに分類される．1 つは RDBMS の外部から SQL クエリを受け取り評価する機能のグループ，もう 1 つは RDBMS 上に登録されたストアドプロシージャ上で動的に作成された SQL クエリを評価するための機能のグループである．それぞれのグループに属する機能は異なるインタフェースをクライアントに対して提供するため，2 つのグループ間で異なった方法で機能の拡張を行う必要がある．

(1)　外部から SQL クエリを受け取り評価する機能の拡張

RDBMS は決められたプロトコルに従ってクライアントから SQL クエリを受け取り，評価して結果を返す機能を持っている．本提案手法では RDBMS がクライアントとのやりとりに用いるプロトコルを拡張し，クエリの雛型とプレースホルダにバインドする値の情報を受け取りクエリを作成し評価する機能を RDBMS に追加する．ただしクライアントから受け取ったクエリの雛型がアプリケーション実行時にユーザの入力値に基づいて作成されたものであるとクエリの雛型に対するインジェクション攻撃が起こる可能性があるため，何らかの方法でクエリの雛型が開発時に作成された静的なものであることを保証する必要がある．そのために RDBMS がクライアントから SQL クエリの雛型を受け取る際にクエリの雛型に対するメッセージ認証コード（MAC）も一緒に受け取り，MAC を RDBMS が保持する秘密鍵を用いて検証し検証に成功した場合のみクエリの作成・評価を行うようにプロトコルを設計する．また RDBMS にクエリの雛型に対する MAC を計算してクライアントに返す機能を追加し，それをアプリケーション開発時には利用可能でアプリケーション実行時には利用不可能なように設定する．こうすることでアプリケーション開発時に作成された静的な雛型は MAC が取得できるためクエリの雛型として利用できるが，アプリケーション実行時に動的に作成された雛型は MAC が取得できないためクエリの雛型として利用できないこととなる．したがってクエリの雛型が攻撃者に改竄されることによって SQLIA が成功してしまうことはない．

```
CREATE PROCEDURE [EMP].[RetrieveProfile]
@Name varchar(50), @Passwd varchar(50) WITH EXECUTE AS CALLER
AS
BEGIN
    DECLARE @SQL varchar(200);
    SET @SQL= 'select PROFILE from EMPLOYEE where ';
    IF LEN(@Name) > 0 AND LEN(@Passwd) > 0
    BEGIN
        SELECT @SQL = @SQL + 'NAME = ''' + @Name + ''' and ';
        SELECT @SQL = @SQL + 'PASSWD = ''' + @Passwd + '''';
    END
    ELSE
    BEGIN
        SELECT @SQL = @SQL + 'NAME = ''Guest''';
    END
    EXECUTE(@SQL)
END
```

```
CREATE PROCEDURE [EMP].[RetrieveProfile]
@Name varchar(50), @Passwd varchar(50) WITH EXECUTE AS CALLER
AS
BEGIN
    DECLARE @n varchar(50), @p varchar(50);
    IF LEN(@Name) > 0 AND LEN(@Passwd) > 0
    BEGIN
        SET @n = @Name;
        SET @p = @Passwd;
    END
    ELSE
    BEGIN
        SET @n = 'Guest';
        SET @p = NULL;
    END
    SEXECUTE ('select PROFILE from EMPLOYEE where
          NAME = :name and PASSWD = :passwd OPTIONAL')
              USING :name = @n, :passwd = @p;
END
```

図 5　脆弱なストアドプロシージャの例[4]（左）と提案手法による安全な記述例（右）

Fig. 5　Stored procedure vulnerable to SQLIA and secure one written with SEXECUTE.

(2)　RDBMS 内部で作成された SQL クエリを評価する機能の拡張

RDBMS は通常，外部から受け取ったクエリを評価する機能だけでなくストアドプロシージャの中で動的に作成されたクエリを評価する機能を持っている．そしてこの機能も当然 SQLIA に対する脆弱性の源となりうる[4]．そのため本提案手法ではこれらの機能を拡張して，ストアドプロシージャ中でクエリの雛型とプレースホルダにバインドする値からクエリを生成・実行する機能を追加する．拡張されたクエリ評価のための機能はクエリの雛型を文字列リテラルとして受け取り，プレースホルダにバインドされる値をリテラルもしくは変数参照として受け取る．たとえば，文字列として評価される式 *string_expr* を `EXECUTE` *string_expr* というシンタックスで受け取り SQL として評価する `EXECUTE` 文が存在する場合には，`EXECUTE` 文の代わりとして `SEXECUTE` *string_literal* `[USING (`*placeholder* `=` *value*`)+]` というシンタックスでクエリの雛型 *string_literal* とプレースホルダにバインドされる値 *placeholder* = *value* の列を受け取りクエリを生成・評価する `SEXECUTE` 文を導入する．`EXECUTE` 文の代わりに `SEXECUTE` 文を利用すると，文献 4) で例示されている SQLIA に対して脆弱なストアドプロシージャを**図 5** のように SQLIA に対して安全なストアドプロシージャに書き換えることができる．また `SEXECUTE` 文はストアドプロシージャ上の文字列リテラルをクエリの雛型として受け取るため，クエリの雛型が攻撃者によって改竄されている恐れはない．そのため外部からクエリの雛型を受け取る場合には必要な雛型に対する MAC は不要となる．ただしこの拡張機能を RDBMS に実装する際には，クエリの雛型の中で `SEXECUTE` 文のような拡張機能が受け取る雛型にプレースホルダが指定できることがないように気をつけなければならない．さもなければ，実行時に作成した任意の文字列をクエリの雛型として利用で

© 2009 Information Processing Society of Japan

きてしまい，安全性を保証するための枠組みが崩壊してしまう[*1]．こうした問題を避けるためにも 4.1 節の**詳細**のバインディングの項目で述べた「プレースホルダは SQL の変数参照と同じ位置にのみに出現する」という規則は遵守されなければならない．なおこの拡張の対象となる機能は RDBMS によって異なるが，多くの RDBMS では `EXECUTE` 文のほかに `PREPARE` 文などが対象となる．

4.3　ユーザ権限の拡張

ここまで，RDBMS をいかに拡張してクエリを安全に構築する機能を追加するかについて述べた．しかし安全にクエリを構築する機能を RDBMS に組み込んだとしても，開発者がこれらの機能を利用せずに今までどおりクエリ構築のロジックをプログラムの中に自分で作成し，プログラム内で生成されたクエリを RDBMS に実行させてしまうと RDBMS の拡張を行った意味がまったくなくなってしまう．そこで何らかの方法で RDBMS の SQL クエリを直接受け取って実行する機能の利用を禁止し，代わりにクエリを安全に構築する機能を利用しなくてはプログラムが書けないようにしなければならない．そのために RDBMS がサポートするユーザの権限管理の機能を拡張してユーザの権限情報に safemode という情報を追加する．そして safemode が true のユーザアカウントで RDBMS にログインしている場合には

- 外部からクエリを受け取って実行する機能や `EXECUTE` 文などの動的に作成されたクエリを評価する機能の利用
- 4.2 節の (1) で導入したクエリに対する MAC を取得する機能の利用
- 自身のユーザ権限情報を操作して safemode を false にすること

が禁止されるようにする．そして Web アプリケーションのプログラムが RDBMS にログインする際に利用するユーザアカウントの safemode を必ず true に設定する．こうすることで，プログラマは RDBMS が提供する安全にクエリを構築するための機能を利用しなければアプリケーションを開発することが不可能となり，必ず安全にクエリを構築するための機能を利用してアプリケーション開発を行うことになる．また safemode が false のユーザアカウントで RDBMS にログインすればすべての機能を今までどおり利用することができるため RDBMS の利便性が損なわれることはない．

⋆1 たとえば SEXECUTE 'SEXECUTE @query' @query = *query* とすることで EXECUTE 文を用いなくても任意のクエリ *query* が実行できてしまう．

5. クライアントサイドの構成

既存の手法ではプログラムが RDBMS を利用する場合には JDBC や ADO.NET などのデータベース接続ライブラリを通じて RDBMS に SQL クエリを受け渡すが，本提案手法ではプログラムが SQL クエリを利用する際には実行したい SQL クエリの雛型と雛型に対する MAC そして雛型から SQL クエリを生成するための情報を受け渡さなければならない．そのため既存のデータベース接続ライブラリを拡張する必要がある．また本提案手法ではアプリケーション中で利用される SQL クエリのすべての雛型に対して，アプリケーション実行前に MAC を計算しておかなければならないためコーディング方法についても拡張が必要となる．これらの拡張の詳細については本論文では省略するが，7 章で紹介する．NET Framework 用のライブラリ実装を利用した際の開発・運用方法の概観は

(1) バインド機構を用いる場合と同様に，ユーザ入力に基づいて決定される部分をプレースホルダとした SQL の雛型と，ユーザ入力値の情報を別々にデータベース接続ライブラリに渡すようにコーディングを行う（**図 6**），

(2) ただし本提案手法では SQL の雛型は静的なものでなくてはならないため，SQL の雛型の文字列は `static` な定数文字列として定義する．また MAC を算出するためには，どの文字列が SQL の雛型であるかが明示されている必要があるため属性（Java であればアノテーション）を利用して定数文字列が SQL の雛型であることを明示する（図 6），

(3) 運用者は MAC 計算用ツールにプログラムの中間コードを渡し SQL の雛型に対する MAC 値の取得を行う．このツールは中間コードからリフレクションを利用してプログラム中で利用される SQL の雛型を抽出しデータベース上でそれぞれの雛型に対する MAC の計算を行い，SQL の雛型とその MAC のペアのリストをファイルとして出力する，

(4) そして，MAC 計算用ツールによって出力されたファイルのパスが接続文字列としてデータベース接続ライブラリに渡されるように Web アプリケーションの設定を行いアプリケーションを実行する．実行時にはデータベース接続ライブラリがそのファイルから SQL の雛型に対応する MAC を読み込み，RDBMS に SQL の雛型と MAC を送信し SQL の実行を行う，

のようになる．バインド機構を利用したコーディングの例（図 1 右）と図 6 を比較すると分かるように，本提案手法ではバインド機構を利用したコーディングの方法とよく似た方法

© 2009 Information Processing Society of Japan

```
// 属性を利用してどの文字列がSQLの雛型として利用されるかを明示
[BuiltinSql]
public const string query = "select count(*) from users where id = @id"
                          + " and password = @password";

public bool Authenticate(string id, string password)
{
    // RDBMSに接続するための情報を設定ファイルから読み込む
    // この設定情報にMACが格納されたファイルのパスも含まれる
    string connectionString = LoadConfiguration("connection_string");
    // 設定情報から接続オブジェクトを作成
    // MACの情報が接続オブジェクトに結び付けられる
    using (SqlConnection conn = new SqlConnection(connectionString))
    {
        // RDBMSへの接続の開始
        conn.Open();
        // 接続オブジェクトからコマンドオブジェクトを作成
        // コマンドオブジェクトにSQLの雛型を渡す
        using (SqlCommand cmd = new SqlCommand(query, conn))
        {
            // 未定値のパラメータへ値を追加
            cmd.Parameters.Add("@id", id);
            cmd.Parameters.Add("@password", password);
            // SQLの実行
            // 接続オブジェクトに結び付けられたMACが内部で利用される
            int result = (int)cmd.ExecuteScalar();
            return result != 0;
        }
    }
}
```

図 6　本提案手法におけるコーディングの例
Fig. 6　Example of code developed in our proposal.

でプログラムを記述することが可能である．

6. アプリケーション開発への導入

本章ではアプリケーション開発を大きく設計・開発・検証・運用の 4 つのフェーズに分けた際の各フェーズと本提案手法との関係について述べる．まず本提案手法ではアプリケーションを設計する段階で，運用時には 4 章で述べた安全にクエリ構築を行うための機能が実装された RDBMS を利用することと SQL を利用する際には RDBMS が提供する安全にクエリ構築を行うための機能を利用するようにコーディングを行うことを決定し，仕様として設計書に明記する．そして開発の段階では，設計書に従い，SQL を使用する際には RDBMS が提供するクエリ構築機能を利用するようにプログラムを作成する．もしも開発者が設計書の内容に完璧に従っていて，作成されたプログラムが RDBMS が提供するクエリ構築機能を確実に利用していればこの時点で作成されたプログラムは SQLIA に対して安全なものとなる．しかし実際には開発者が設計書をきちんと読んでいなかったり何らかのミスを犯すことにより，RDBMS が提供するクエリ構築機能を正しく利用せずに SQLIA に対して脆弱なプログラムを記述してしまっている可能性がある．

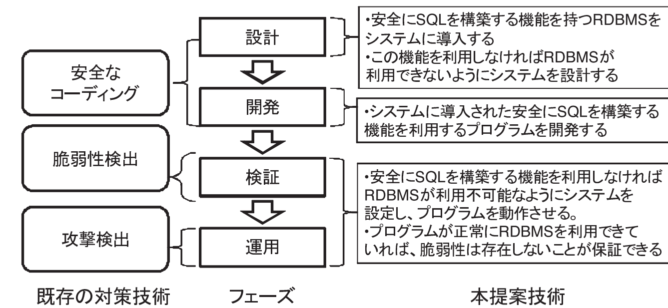

図 7　アプリケーション開発と対策技術の関係
Fig. 7　Relationship between application development and countermeasures.

そこでプログラムを検証する際に，4.3 節で述べた safemode が true のユーザアカウントを必ず利用してプログラムが RDBMS にログインする設定になっていることを確認したうえでアプリケーションが正常に動作するかを確認する．もしも開発者が設計書の内容に従わず，プログラムに独自の SQL クエリ構築処理を実装してその処理によって生成された SQL クエリを RDBMS に実行させようとしている場合には RDBMS によって SQL クエリの実行が拒絶される．そのため，RDBMS による拒絶が生じていないことを確かめることで検証を行った部分についてはプログラムは必ず RDBMS が提供する安全なクエリ構築機能を適切に利用していることが確かめられる．また安全なクエリ構築機能を適切に利用していればプログラムは SQLIA に対して安全なものとなるため，結果として動作の検証を行った部分についてはプログラムが SQLIA に対して安全であることを保証することができる．また運用時にも検証時と同様に，プログラムが RDBMS にログインする際のユーザアカウントの safemode が true になっていることを確認したうえでアプリケーションを実行することで，たとえ動作の検証に漏れがあり見落とされている独自のクエリ構築処理が存在したとしても，検証時と同様に RDBMS がその部分で動的に作成されたクエリの実行を拒絶するため，独自のクエリ構築処理が SQLIA に対する脆弱性を招くことはありえず，アプリケーションが SQLIA に対して安全であることを保証することができる．

以上で述べた開発のそれぞれのフェーズと提案技術との関係およびそれぞれのフェーズと既存の SQLIA 対策技術との関係を表したものが**図 7** である．この図に表れているように

© 2009 Information Processing Society of Japan

本提案手法はアプリケーション開発の設計から運用まで全体にわたる技術であり，その他の技術がアプリケーション開発の一部のフェーズでしか利用されないのと対照的である．

7. 実　　装

本提案手法の実装例として，本提案手法で必要となる拡張を PostgreSQL[13] に対して施したものを作成した．またクライアントサイドのデータベース接続ライブラリの実装例として，.NET Framework 用の PostgreSQL 接続ライブラリ Npgsql[14] を拡張した接続ライブラリを作成した．この実装は文献 15) から入手可能である．

8. 評　　価

8.1 パフォーマンス

本提案手法では既存の方法よりも複雑な方法で SQL を構築し RDBMS 上で実行するため実行速度の劣化が発生する．そこで 7 章で述べた PostgreSQL を拡張して提案手法を実装した RDBMS と Npgsql を拡張したデータベース接続ライブラリを利用して簡単な実行速度の測定を行い，本提案手法の導入がアプリケーションに対してどの程度の性能劣化を引き起こすのかについて考察を行った．まずはじめに SQL 中のリテラルを実行時に制御する場合の例として，`select a from t where a = @a` という SQL の`@a` の部分を実行時に決定し RDBMS に接続しデータを取得するという処理を

- 提案手法を利用せずに SQL をプログラム側で組み立てる，
- 提案手法が提供する安全に SQL を構築する機能は利用するが，SQL の雛型に対する MAC の検証処理は行わない，
- 提案手法が提供する安全に SQL を構築する機能は利用し，SQL の雛型に対する MAC の検証処理も行う，

という 3 通りの方法で実装し，実行時間の測定を行った．次に本提案手法が提供するより複雑な SQL 構築機能を利用した場合の例として 4.1 節で提案した Optional Condition の機能を利用した場合の実行時間の測定を行った．具体的には SQL の雛型 `select a from t where a = @a optional and b = @b optional and c = @c optional and d = @d optional and e = @e optional` に対して`@a` にのみ非 `NULL` をバインドし`@b`〜`@e` には `NULL` をバインドすることで SQL の雛型から `b = @b optional and c = @c optional and d = @d optional and e = @e optional` の部分を除外し `select a from t where a = @a` という形の SQL を生成・実行する処理の実行時間を測定した．

表 2　実行速度の比較
Table 2　Comparison of performances.

	提案手法の利用なし	MAC の検証なし	MAC の検証あり
リテラルの制御	4.5 ms	4.7 ms	5.9 ms
Optional Condition	-	4.7 ms	6.3 ms

また既存の手法で条件式の有無の制御を行う場合は特に特殊な処理は必要とならず，単に `select a from t where a = @a` という形の SQL を実行する場合とパフォーマンス上の差異は存在しないため，本提案手法を利用して実装した場合の実行時間についてのみ測定を行った．その結果が**表 2** である．なお実験には Intel Core 2 Duo T7300 を搭載したコンピュータにインストールされた Windows Vista 上に Virtual PC 2007 を利用して構築した仮想マシン環境を利用した．

まず表 2 の「リテラルの制御」の「提案手法の利用なし」と「提案手法（MAC の検証なし）」の値を比較すると分かるように，本提案手法で導入する安全な SQL 構築機能を利用して RDBMS 上で SQL の生成を行う場合の実行時間の増加は 5%程度であり，SQL の構築処理をプログラム上ではなく RDBMS 上で行うことによる実行性能への影響は限定的であることが分かる．一方で「提案手法の利用なし」と「提案手法（MAC の検証あり）」の値を比較すると 30%以上の実行時間の増加が生じてしまっており，SQL の雛型が静的であることを保証するために行っている MAC 検証処理が大きなパフォーマンスの低下を招いていることが分かる．また「リテラルの制御」と「Optional Condition」を比較すると分かるように RDBMS 上で行われる SQL の構築処理が複雑化しても MAC の検証を行わない場合はパフォーマンスの低下は小さいが，SQL の雛型が複雑化すると雛型が長くなり MAC の検証にかかる時間が増大するため，MAC の検証を行う場合の実行時間が既存の手法に比べて 40%近く増加してしまう．

もちろんアプリケーションの実行時間は RDBMS の処理時間だけで決まるわけではないので，この実行時間の増大がそのままアプリケーション全体の実行時間の増大に直結するとは限らないが，アプリケーションによっては本提案手法の導入が実行時間の増大を招く危険があり，パフォーマンス改善のための実装上の工夫を行うべきであること，またその際には MAC の検証にかかっている処理時間をいかに削減するか[⋆1]が重要となることをこの実験は

⋆1 繰り返し利用される SQL の雛型に対する MAC の検証を 1 回しか行わないようにすることなどで，パフォーマンスを大きく改善できる余地がある．

© 2009 Information Processing Society of Japan

表 3　アプリケーションで利用される SQL クエリの性質
Table 3　Characteristics of SQL queries in web applications.

	Total	Bind	Optional	Repeat	Order	Other
Online Bookstore	97	81	11	0	8	0
Bug Tracking System	49	44	2	0	4	0
Employee Directory	29	25	3	0	3	0
Events	36	30	4	0	3	0
Classifieds	51	38	13	0	5	0
Online Portal	87	67	16	0	11	0

示している．

8.2　クエリ生成機能の妥当性

本提案手法では実行時の SQL クエリ構築処理において，プレースホルダへの値のバインド・条件の有無の制御・条件の繰返し・ソート順序の制御の 4 つの処理以外は行われないことを保証することで SQLIA に対するシステムの安全性を保証する．そのため SQL クエリを構築する際にこれらの 4 つの処理以外が必要なアプリケーションは本提案手法の安全性保証の枠組みの中では開発することが不可能となる．そのため SQL に対して実行時に行えることをプレースホルダへの値のバインド・条件の有無の制御・条件の繰返し・ソート順序の制御の 4 つのみに制限してしまうことが本当に妥当であるのか，つまりこの制限が本当に Web アプリケーション開発の妨げになることがないのかを検証しなくてはならない．そこで文献 10)，11) で性能評価に利用されているアプリケーション[16] から Online Bookstore, Bug Tracking System，Employee Directory，Events，Classifieds，Online Portal の 6 つのアプリケーションを選び，それぞれのアプリケーションが実行時に構築する SQL クエリの性質について調査を行い，これらのアプリケーションが本提案手法の枠組みの中で実際に開発することが可能かどうかを検証した．

表 3 がその結果である．Total が各アプリケーションを本提案手法を用いて実装した場合に必要となる SQL の雛型の総数，Bind がそれらのうちバインド機構のみで記述することができる SQL の雛型の個数，Optional が実行時に有無が決定される条件式を検索条件に含む SQL の雛型の個数，Repeat が条件の繰返しを検索条件に含む SQL の雛型の個数，Order が実行時にデータのソート順序が決定される SQL の雛型の個数，Other がその他の動的な要素を含むため本提案手法が適用できない SQL の雛型の個数である．ただし Optional と Repeat の両方に含まれる SQL の雛型が存在するため Bind，Optional，Repeat，Order, Other の総和は Total にはならない．表 3 ですべてのアプリケーションの Other が 0 となっていることが示すように，今回検証を行ったアプリケーションには値のバインド・条件の有無・条件の繰返し・ソート順序の制御以外の操作が SQL クエリ構築時に必要となるアプリケーションは存在しなかった．そのためここであげたプログラムに対しては本提案手法が適用可能であることが確認できた．当然，この結果だけをもって本提案手法があらゆる Web アプリケーションに対して適応可能であるということはできず，実際には本提案手法が適用可能であるかをより多くのアプリケーションに対して検証していく必要はある．ただしこの検証結果は本提案手法が実用的なアプリケーションに対して適応可能であるということを，つまり本提案手法の有効性を，他の研究[10),11)] と同程度のレベルで示している．

また攻撃の検出や脆弱性の検出といった既存の対策手法では，攻撃のパターンやプログラムの構造がそれぞれの手法が想定している範囲から逸脱した場合，攻撃の検出漏れや脆弱性の検出漏れが生じてしまいシステムに存在する脆弱性が放置されることになる．そのため対策技術が有効に機能していないことは実際に攻撃を受けてしまうか脆弱性が他の対策技術によって発見されるまで顕在化することがない．一方で本提案手法ではプログラムの構造が我々が想定している範囲を逸脱した場合，つまりプレースホルダへの値のバインド・条件の有無の制御・条件の繰返し・ソート順序の制御以外の処理が必要になった場合には，プログラムが作成できないという問題が発生するため，本提案手法が有効に機能しないことが開発時に顕在化する．そのため既存の対策技術のように対策技術が有効に機能していないことに気がつかずにシステムに存在する脆弱性が放置されるという事態を招くことはない．このように本提案手法は対象とするアプリケーションが想定する範囲から逸脱している場合でもシステムの脆弱性が放置されるような事態を招くことはなくアプリケーションの開発を中断させるという点で，既存の対策技術に比べてフェイルセーフであるといえる．

また SQL クエリを構築するにあたって，値のバインド・条件の有無・条件の繰返し・ソート順序の制御以外の操作が必要となる部分がアプリケーション内部に存在したとしても，実際にはその部分については本提案手法を利用せずに通常の方法で SQL クエリの構築を行い通常の方法で RDBMS に SQL クエリを渡すように記述しておき，アプリケーションの検証および運用時には，それ以外の部分は必ず safemode が true のアカウントで RDBMS に接続していることを慎重に確認したうえで，本提案手法の枠組みの中では記述できない処理の部分のみ例外的に safemode が false のアカウントで RDBMS に接続することを許すことで，アプリケーションの開発を行うことはできる．この場合，本提案手法を用いずに開発した部分については本提案手法による安全性の保証が及ばなくなってしまうが，表 3 の結果が示すようにこうした処理が必要となる部分は仮に存在したとしてもアプリケーション全体

© 2009 Information Processing Society of Japan

のごく一部に限られるので，その部分については特に慎重にプログラムを記述し，人手や他の脆弱性検出技術を利用して重点的に安全性の検証を行うことで SQLIA に対する脆弱性を排除することは十分可能である．このように本提案手法の枠組みの中では記述できない部分がアプリケーション中に存在したとしても，本提案手法がまったく無意味になることも本提案手法によってアプリケーション開発自体が不可能になることもない．そのため本提案手法の導入によって SQL の構築方法に制限が生じることを必要以上に懸念する必要はない．

9. 考　　察

9.1 脆弱性検出技術との比較

脆弱性検出技術と本提案手法はともにプログラムに脆弱性が存在しないことをプログラマ任せにせずに保証することを目的とした技術であり，それぞれをアプリケーション開発に導入することによってもたらされるメリットは同じである．しかし脆弱性が存在しないことを保証するためのアプローチは 2 つの間で大きく異なっており，脆弱性検出技術が任意のプログラムに対して検証を行い特定の脆弱性が存在しないことを保証するのに対し，本提案手法では攻撃に対して安全なプログラムを書くための方法を提供しておき，提供された安全な開発方法を利用していないプログラムは脆弱性の有無にかかわらずすべて危険なもの見なすことでアプリケーションの安全性を保証している．そしてこのアプローチの違いが安全性の保証の確実さとそれぞれの技術を利用して安全性を保証するのに必要なコストに大きな影響を与える．

まず安全性の保証の確実さについては本提案手法のほうが脆弱性検出技術よりも高い．なぜなら脆弱性検出技術で必要となる任意のプログラムを解析してプログラムを脆弱な部分と安全な部分とに分類する処理というのは単純なものではなく高度で複雑なプログラム解析技術をもってしてもまったく誤分類のない完璧な分類を行うのは困難であるが，一方で本提案手法で必要となる，特定の安全な方法を利用して書かれているプログラムのみを許容しその他のプログラムは脆弱性があるかどうかにかかわらずすべて拒絶するという処理は，許容されるものと拒絶されるものとの境界が明確であるため簡単な仕組みで確実に実現することが可能であるからである．また本提案手法では安全性の保証の基盤となる部分は Web アプリケーションのプログラムではなく RDBMS 側にあるため，RDBMS の内部で実行されるストアドプロシージャ上で生じる SQLIA についても包括的に取り扱うことが可能であり，そうした面でも安全性の保証の確実さが既存の脆弱性検出技術に比べて高いといえる．

他方で本提案手法の性質上，本提案手法で既存の Web アプリケーションの安全性を保証する場合にはプログラムの書き換えが不可欠となる．そして元の Web アプリケーションに脆弱性が存在しない場合，このプログラムの書き換えにかかるコストは結果的にはまったくの無駄である．一方で脆弱性検出技術を利用して Web アプリケーションの安全性を保証する場合には脆弱性検出用のツールを利用して Web アプリケーションの検証を行うだけでよいため，本提案手法で必要となるプログラムの書き換えに比べて小さな手間で安全性が保証できる．そのため既存のアプリケーションを対象とする場合には，本提案手法を用いるよりも脆弱性検出技術を利用したほうが小さなコストでアプリケーションの SQLIA に対する安全性の保証が行える．

9.2 データベースの拡張を利用しない手法との比較

本提案手法ではプログラムが安全な SQL 構築手法を利用していることを確実に保証するために RDBMS に対する拡張を行う．これは 2.2，2.3 節で紹介した攻撃・脆弱性検出技術がプログラムに対して何らかの解析を行うことで安全性の保証を行う手法であり RDBMS に対する拡張をいっさい必要としないのと対照的であり，安全性の保証のための仕組みを RDBMS に対して施す点が本提案手法の大きな特徴となっている．またそれは本提案手法の利点とも欠点ともなる．

まず利点の 1 つは，安全性の保証のための技術が開発に用いられるプログラミング言語に依存しないという点である．そのため本提案手法は Java や C#だけでなく C/C++や Python，Ruby などのスクリプト言語を用いて開発を行う場合であっても問題なく適用することができる[*1]．逆に 2.2，2.3 節で紹介した安全性保証の技術の多くは，ソースコード解析技術などのプログラミング言語の性質に強く依存した技術に基づいており，たとえば C++，Java，Python などの性質が大きく異なる言語のすべてに 1 つの対策技術を適用することは困難である．また本提案手法は RDBMS に対する拡張を行うため，その他の対策技術で無視されているストアドプロシージャ中で生じる SQLIA も包括的に取り扱うことができるという利点がある．

一方で本提案手法は安全性の保証のための拡張を RDBMS に対して行う必要があるため，同一のプログラミング言語を利用している場合であっても利用する RDBMS が異なる場合，それぞれの RDBMS ごとに安全性の保証のための拡張を用意しなくてはならない．また SQL Sever や Oracle DB のような非オープンソースの RDBMS を利用している場合には，開発元が本提案手法に対応しない限り本提案手法を利用することは不可能であるという欠

⋆1 RDBMS に対する拡張に合わせて各言語ごとのクライアント接続ライブラリを拡張する必要はある．

© 2009 Information Processing Society of Japan

点がある．そのため本提案手法の導入することが RDBMS の選択肢の幅を狭めることにつながってしまう危険性がある．一方で 2.2，2.3 節で紹介した技術は RDBMS に対する拡張を必要としないため本提案手法に比べ RDBMS への依存度が低く，どの RDBMS を利用するかという決定に影響を及ぼすことがない．そうした点で RDBMS に対する拡張を必要としない既存の対策技術のほうが本提案性に比べ利便性が高いといえる．

10. 展　　望

本提案手法の大きな欠点は，9.1 節で述べたように既存のアプリケーションに導入する際にはプログラムの書き換えという処理が不可欠となり，これを人手で行うと大きなコストがかかってしまう点にある．そのためこのプログラム書き換えコストを減少させることが本提案手法の使いやすさを向上させるうえで重要であるが，これに関しては文献 10) で利用されているプログラムの各部分でどのような SQL クエリが生成されるかを分析する静的解析技術などを応用することで，プログラムの書き換えを自動的に行うことが可能ではないかと考えられる．もしも文献 10) のような静的解析を用いた SQLIA 対策技術が有効に機能する部分については自動書き換えが行えたとするならば，人手による書き換えはプログラムの一部の静的解析が正確に行うことができない部分についてのみでよくなり，プログラムの書き換えコストを大きく削減することが可能になるのではないかと考えられる．

また本論文では特に SQL インジェクション攻撃に注目していたが，安全に SQL や HTML などのプログラムを構築することができる手段を提供しておき，その利用をプログラム開発者に強制付けることでアプリケーションのインジェクション攻撃に対する安全性を保証するという本提案手法の根本にある考え方自体は SQL と何の関係もない．そのため Web アプリケーションで利用されることの多い HTML，XPath，XSLT，XML，OS コマンド，LDAP クエリなどの SQL 以外の言語に対して本提案手法の考え方を適用することで，それぞれの言語へのインジェクション攻撃に対する脆弱なプログラムの記述を禁止しアプリケーションの安全性を保証するフレームワークを構築することが可能であると考えられる．特に HTML に対するインジェクション攻撃を防ぐことは SQLIA 以上に脆弱性が多いといわれるクロスサイトスクリプティングを防ぐことになるため，本提案手法の考え方を HTML に対して適用して HTML へのインジェクション攻撃に対して脆弱なプログラムの開発を防止するフレームワークを構築することができれば，大きな意味がある．

参　考　文　献

1) OWASP Top Ten Project: OWASP Top 10 2007. http://www.owasp.org/index.php/Top_10_2007
2) 情報処理推進機構：情報セキュリティ白書 (2008).
3) Halfond, W.G., Viegas, J. and Orso, A.: A Classification of SQL-Injection Attacks and Countermeasures, *Proc. Intern. Symposium on Secure Software Engineering* (2006).
4) Wei, K., Muthuprasanna, M. and Kothari, S.: Preventing SQL Injection Attacks in Stored Procedures, *ASWEC '06: Proc. Australian Software Engineering Conference*, Washington, DC, USA, pp.191–198, IEEE Computer Society (2006).
5) McClure, R.A. and Krüger, I.H.: SQL DOM: Compile time checking of dynamic SQL statements, *ICSE '05: Proc. 27th International Conference on Software Engineering*, New York, NY, USA, pp.88–96, ACM Press (2005).
6) Boyd, S.W. and Keromytis, A.D.: Sqlrand: Preventing Sql Injection Attacks, *Proc. 2nd Applied Cryptography and Network Security* (*ACNS*) Conference, pp.292–302 (2004).
7) Buehrer, G., Weide, B.W. and Sivilotti, P.A.G.: Using parse tree validation to prevent SQL injection attacks, *SEM '05: Proc. 5th International Workshop on Software Engineering and Middleware*, New York, NY, USA, pp.106–113, ACM Press (2005).
8) 大久保隆夫，田中英彦：インジェクション系攻撃防止ライブラリの評価，情報処理学会研究報告 CSEC［コンピュータセキュリティ］，Vol.2007, No.71, pp.177–184 (2007).
9) Su, Z. and Wassermann, G.: The essence of command injection attacks in web applications, *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, pp.372–382, ACM (2006).
10) Halfond, W.G.J. and Orso, A.: AMNESIA: Analysis and monitoring for NEutralizing SQL-injection attacks, *ASE '05: Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA, pp.174–183, ACM (2005).
11) Bandhakavi, S., Bisht, P., Madhusudan, P. and Venkatakrishnan, V.N.: CANDID: preventing sql injection attacks using dynamic candidate evaluations, *CCS '07: Proc. 14th ACM Conference on Computer and Communications Security*, New York, NY, USA, pp.12–24, ACM (2007).
12) 情報処理推進機構：セキュアプログラミング講座 Perl の Taint モード. http://www.ipa.go.jp/security/awareness/vendor/programming/a04_03.html
13) PostgreSQL: The world's most advanced open source database. http://www.postgresql.org/

© 2009 Information Processing Society of Japan

14) Npgsql: .Net Data Provider for Postgresql. http://npgsql.projects.postgresql.org/
15) Svsql: Svsql documentation. http://www.logos.ic.i.u-tokyo.ac.jp/~yunabe/svsql/
16) GotoCode: Open Source Web Applications with Source Code in ASP, JSP, PHP, Perl, ColdFusion, ASP.NET / C#. http://www.gotocode.com

(平成 20 年 12 月 1 日受付)
(平成 21 年 6 月 4 日採録)

渡邉　悠

昭和 59 年生まれ．平成 21 年 3 月東京大学大学院情報理工学系研究科電子情報学専攻修士課程修了．並列分散計算やソフトウェア開発，コンピュータセキュリティ等に関する研究開発に興味を持つ．

松浦　幹太（正会員）

昭和 44 年生まれ．平成 9 年 3 月東京大学大学院工学系研究科電子工学専攻博士課程修了．博士（工学）．平成 9 年 4 月東京大学生産技術研究所助手等を経て，平成 14 年 4 月同助教授（平成 19 年 4 月より准教授）．平成 12 年 3 月から平成 13 年 3 月までケンブリッジ大学客員研究員．情報セキュリティの研究に従事．ACM シニア会員（2009 年），電子情報通信学会シニア会員（2009 年）．IACR，IEEE，社会・経済システム学会，情報理論とその応用学会各会員．日本セキュリティマネジメント学会非常任理事（2004 年から 2008 年まで）．同常任理事（2008 年から現在）．

　© 2009 Information Processing Society of Japan