修士論文

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

Assured Resistance against SQL-Injection Attacks : A Whitelist-Coding Approach and Implementation

指導教員　　松浦幹太　准教授

東京大学大学院 情報理工学系研究科 電子情報学専攻

76437　　渡邉　悠

平成21年2月4日提出

内容梗概

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，ソフトウェアの安全性がより一層重要となっている昨今において，ソフトウェア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，ソフトウェア脆弱性の中でもデータベースに対する不正操作を引き起こし Web アプリケーションを開発・運用する際に問題となる SQL インジェクション攻撃に対する脆弱性に注目し，アプリケーションの SQL インジェクション攻撃に対する耐性を保証する新しい枠組みとその技術的実現手法の提案を行った．

本論文で提案する新しい耐性保証の手法は，攻撃の検出や脆弱性の検出を用いた既存の SQL インジェクション攻撃対策技術とは異なり，あらかじめ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．それゆえ攻撃の検出や脆弱性の検出を利用した既存の手法に比べ，検出漏れにより脆弱性が放置される危険や誤検知によって通常のユーザ操作が拒否される危険性が原理的に小さい手法である．また提案手法の有効性を実証するために，本提案手法で必要となる拡張をオープンソースのリレーショナルデータベース管理システム PostgreSQL に対して施したデータベースと，拡張されたデータベースを利用するためのデータベース接続ライブラリの拡張を行った．

目次

Chapter 1　序論

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おり，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．

例えば文献[1]によると，2006年の時点で報告される脆弱性の60%以上がWebアプリケーションの脆弱性となっている（[2], 図1.1）．またこの報告によれば，Webアプリケーションの脆弱性のうちクロスサイトスクリプティング（XSS）が38%を・SQLインジェクション（SQLIA）が23%を占めている．この２つの種類の脆弱性は古くから非常に有名なものであり，どちらもユーザからの入力がアプリケーション側で適切に処理されないことによってプログラマの予期せぬHTMLやSQLが生成されてしまうことによって生じる「インジェクション系」の脆弱性である．つまりはWebアプリケーションの脆弱性の実に60%以上がインジェクション系の脆弱性であるということであり，単純に考えればソフトウェアの脆弱性の1/3以上がWebアプリケーションに対するインジェクション系の脆弱性であるということになる．また他の脆弱性調査においてもWebアプリケーションの脆弱性の報告数の1位と2位はやはりクロスサイトスクリプティングとSQLインジェクション攻撃であり，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[3, 4]．そのためインジェクション系の脆弱性を解決することが，世の中のソフトウェアをより安全なものにするために非常に重要であるといえる．

1.2　Webアプリケーションセキュリティ

Webアプリケーションの脆弱性について述べる前に，典型的なWebアプリケーションの構造について簡単に述べる（図1.2）．まず単一のマシン上で動作するアプリケーションとは異なり，Webアプリケーションはアプリケーションを利用するユーザ側とアプリケーションを提供するサーバ側とに分けられる．ユーザサイドではユーザは通常ブラウザを利用してアプリケーションを利用する．またサーバサイドは，ユーザと直接通信を行うWebサーバとデータの管理を行うデータベースによって構成され，アプリケーションの機能を提供するプログラムはWebサーバ上で実行される．そして，ブラウザ上でのユーザの入力や操作に基づいてブラウザからWebサーバに対してリク

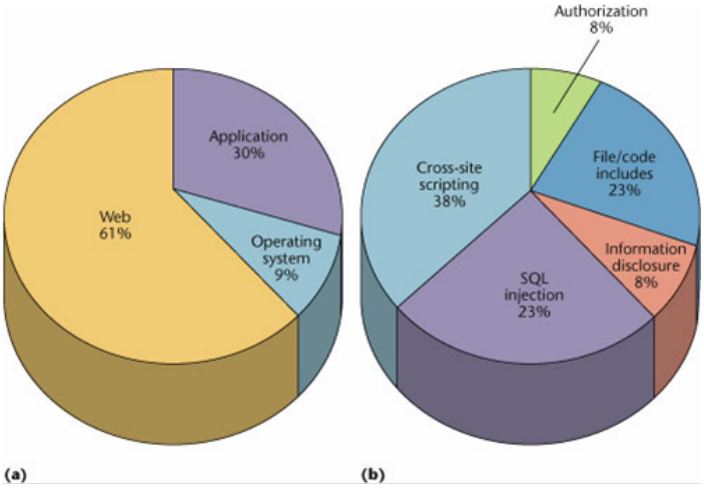

図 1.1: (a) Breakdown of disclosed vulnerabilities by software type in May 2006 (b) current vulnerability types disclosed in Web-based applications. (Source: SecurityFocus.com[2])

エストが送信され，Web サーバは受け取ったリクエストをプログラムに渡し，プログラムがリクエストを処理し処理結果を Web サーバへ伝え，それが Web サーバからブラウザにレスポンスとして返される．その際にデータの取得や更新が必要な場合にはプログラムがデータベースを利用する．

これらのアプリケーションを構成する４つのコンポーネントのうち，ブラウザ・Web サーバ・データベースに関してはアプリケーション毎に用意されることはなく，例えばブラウザであれば Internet Explore や Firefox，Web サーバであれば Apache HTTP Server や Internet Information Services，データベースであれば Oracle DB や SQL Server，PostgreSQL，MySQL などのほんの数種類のプログラムがさまざまなアプリケーションで幅広く利用されている．一方で，アプリケーションの機能自体を提供するプログラムは当然アプリケーション毎に作成されるものであり，さまざまな Web アプリケーションで幅広く利用されることはない．そのためプログラムはブラウザ・Web サーバ・データベースに比べ開発に利用できる資源が非常に限られたものになり，少数の開発者によって短期間で作成される場合が多くなる．

それゆえプログラムの脆弱性に対する検証は不十分なものとなりやすいため，ブラウザ・Web サーバ・データベースに比べてプログラムにはそもそも脆弱性が作り込まれやすく，その脆弱性はシステム管理者の単純な設定ミスによる脆弱性や典型的なプログラムのバグによる脆弱性など広く知られており比較的容易に発見可能な脆弱性であることが多い．一方で，ブラウザ・Web サーバ・データベースは Web アプリケーションに比べて多くの開発者と時間を利用して開発され幅広く利用されているため，脆弱性に対する検証もしっかりしたものとなり脆弱性の数は Web アプリケーションに比べて非常に少ない．しかし脆弱性が存在した場合には影響を受ける範囲が非常に広域に渡るため１つの脆弱性が非常に大きな被害をもたらす可能性がある．そのため，どちらの種類の脆弱性もインターネットを通じてサービスを提供する上では重大な問題と

なる．

ただし，一般に「Web アプリケーションセキュリティ」といった場合にはそれぞれのサービス提供者が Web アプリケーションを開発・運用するにあったって行うべきセキュリティ対策のことを指すため，ブラウザ・Web サーバ・データベースなどの幅広く利用されているコンポーネントに対する脆弱性対策は「Web アプリケーションセキュリティ」には普通含まれない．そのため Web アプリケーションセキュリティを行う上で重要となる脆弱性対策は，基本的に広く知られており比較的容易に発見可能な脆弱性を対象とした対策となる．本論文で取り扱うインジェクション攻撃に対する脆弱性もこのタイプの脆弱性である．

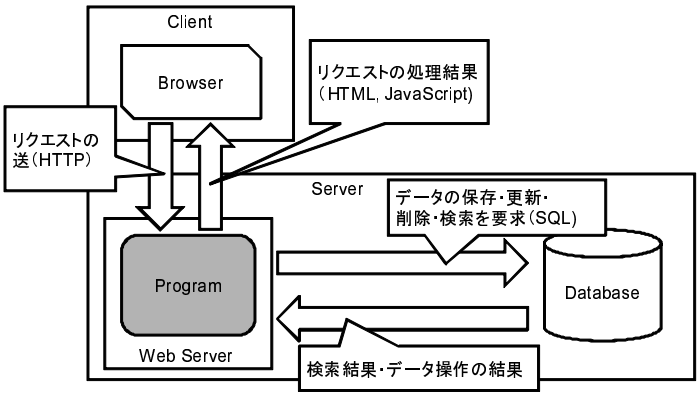

図 1.2: Web アプリケーションの構造

1.3　本論文の構成

本論文は以下の流れで構成される．

本章の後半 SQL インジェクション攻撃について簡単な説明を行い，ユーザ入力値に対する検証と変換による原始的な対策手法について説明を行う．

第 2 章 関連研究 原始的な対策手法よりも洗練された SQL インジェクション対策技術およびに研究について，安全なコーディング技術・攻撃検出技術・脆弱性検出技術に分類した上で紹介を行い，それぞれの対策技術が抱える問題点について述べる．

第 3 章 研究概要 第 3 章で紹介した対策技術および研究が抱える問題点を踏まえた上で本研究のモチベーションについて述べ，本論文で提案する新しい SQL インジェクション対策技術の考え方と提案手法の概要について述べる．

第 4 章 サーバサイドの構成 本論文で提案する SQL インジェクションに対するアプリケーションの耐性保証の枠組みの核となる，データベース（RDBMS）に対する拡張について述べる．

第 5 章 クライアントサイドの構成 本提案手法を導入する際に，Web アプリケーションのプログラム側で必要となる拡張について述べる．

第 6 章 提案手法の導入方法と実装 アプリケーション開発における，設計・開発・検証・運用の各フェーズと本提案手法との関係について簡単に述べ，本提案手法がアプリケーションの開発手法にどのような影響を与えるかについて考察を行う．また本提案手法の有効性を検証するために作成した実装について紹介を行う．

第 7 章 評価 本提案手法を導入した場合に生じる，実行時のパフォーマンス低下と開発時の自由度の低下について評価を行い，これらの問題点が実際にアプリケーションを開発・運用する際には問題にならない可能性が高いことを示す．

第 8 章 結論 本論文の貢献についてまとめ，最後に今後の課題や展望について述べる．

1.4　インジェクション攻撃

1.4.1　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）に対して問い合わ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 の構築が行われている．そして，その SQL 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 を改竄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（SQLIA）に対する脆弱性と言われるものである．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 1.3, [5]）．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を行うものであり，この関数を定義したプログラマは例えば ID に “yunabe”，パスワードに “pass” という文字が入力された場合に

```
 select count(*) from users where id = 'yunabe' and
password = 'pass'
```

という SQL が作成され，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 は

```
 select count(*) from users where id = 'yunabe' and
password = '' or 'a' = 'a'
```

となってしまい，where 節がプログラマの想定に反して常に 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関わらず yunabe として認証されてしまうことになる．

SQLIA の具体的な方法は，この例であげた条件式を常に true になるものに改竄する方法以外にもいくつも存在するが [6]，いずれの攻撃方法もアプリケーション中の

SQL 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 [7].

```
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;
    }
}
```

図 1.3: SQL インジェクション攻撃に対して脆弱なコード

1.4.2　他の言語に対するインジェクション攻撃

SQL 以外にも HTML, JavaScript, XPath, XSLT, XML, OS コマンド, LDAP クエリなどの言語が Web アプリケーションの内部でユーザ入力値に基づいて組み立てられ利用されることがある．そのためこれらの言語の構築処理の不備をついた攻撃も SQLIA と同様に存在する．特にほとんど全ての Web アプリケーションで利用される HTML に対するインジェクション攻撃は SQLIA と同程度かそれ以上に脆弱性が多いと言われるクロスサイトスクリプティング（XSS）に結びつくことになるため Web アプリケーションの安全性を保つ上で大きな問題となる．また攻撃の対象となりうる言語がいくつも存在することは，将来的に例え SQL が利用されなくなったとしても他の言語を対象とした同じような攻撃が猛威をふるう可能性が十分あることを表している．

1.5　原始的な対策

1.5.1　入力値の検証による対策

SQLIA や XSS に対する最も原始的な対策は，ユーザ入力値に対する検証を行い不適切な入力を拒否することである．例えば図 1.3 で例示したプログラムにおいて SQLIA による SQL の改竄が生じたのは，ユーザ入力にシングルクオートが含まれていたためでありユーザの入力値がアルファベットのみであれば脆弱性は生じ得ない．そこでユーザがパスワードおよびユーザ名に入力できる値はアルファベットのみ（正規表現 `^[a-zA-Z]*$`にマッチするもののみ）であると制限してしまい，それ以外の値が含まれている場合にはアクセスを拒否するようにすれば SQL 攻撃に対する脆弱性は生じない．この方針に従って図 1.3 の疑似プログラムを修正すると図 1.4 のようになる．この

プログラムは SQLIA に対して安全である．しかしユーザの入力値に対する検証を行い，ユーザの入力を制限することのみでシステムをインジェクション攻撃から保護しようという対策手法には以下の２つの問題点がある．

```
private Regex allowed = new Regex("^[a-zA-Z]*$");

public bool Authenticate(string id, string password)
{
    if (!allowed.IsMatch(id) || !allowed.IsMatch(password))
    {
        return false;
    }
    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;
    }
}
```

図 1.4: 入力値の検証による SQLIA 脆弱性対策

1.5.2　入力値の検証による対策の問題点

第１に，ユーザの入力として何が許されるかどうかは本来ソフトウェアの実装を行う段階で決まるものではなく，ソフトウェアをデザインする段階で仕様として決定されるものである．そして，その仕様は実装上の都合ではなくソフトウェアのビジネスロジックに基づいて定められるものである．そのため SQLIA を防ぐという実装上の都合のために，実装段階でプログラマが勝手にユーザの入力値を制限することは断じて許されない．上の例のように単にユーザ名とパスワードをアルファベットのみに制限する場合であっても，その制限はシステムの仕様として確実に明記され開発者全体で共有され，システム全体で守られなければならない．さもなければ，例えばパスワード設定を行うページではパスワードに数字が利用できたにも関わらず，パスワード認証を行う画面では数字の入力が許されず正しいパスワードが入力できなくなり，その結果ユーザがシステムにログインできなくなるなどという非常に馬鹿げた事態が発生することとなる．そうした観点から，ユーザの入力値を制限することで SQLIA を防ぐという手法はソフトウェアを実装する際の手段として適切なものとは言い難い．

第２に，どのようにユーザの入力値を制限すれば SQLIA を防げるかはユーザ入力値が SQL 上でどのように利用されるかに依存するという問題がある．例えば，本をタイトルで検索できるアプリケーションを考える．本のタイトルが変数 title に格納されているならば，検索クエリは

```
"SELECT * FROM books WHERE title='" + title + "'";
```

```
public bool validate(string str)
{
    return str != null && str.Contains("'") == false;
}

public Data SearchBook(string title)
{
    if (!validate(title))
    {
        return null;
    }
    string query = "select * from books where title = '" + title + "'";
    using (SqlConnection conn = new SqlConnection(connectionString))
    using (SqlCommand cmd = new SqlCommand(query, conn))
    {
        conn.Open();
        return cmd.ReadData();
    }
    return result;
}

public Data SearchBook(string title, string year)
{
    if (!validate(title) || !validate(year))
    {
        return null;
    }
    string query = "select * from books where title = '"
                + title + "' and year = " + year;
    using (SqlConnection conn = new SqlConnection(connectionString))
    using (SqlCommand cmd = new SqlCommand(query, conn))
    {
        conn.Open();
        return cmd.ReadData();
    }
    return result;
}
```

図 1.5: 不適切な入力値検証による脆弱性

ような文字列の連結で作成することができる．ただしユーザが入力したtitleをそのまま利用してしまうと，SQLIAに対して脆弱になってしまうので入力の検証を行う必要がある．この例ではtitleにシングルクオートが含まれていると想定していたのとは異なる位置で文字列リテラルが終了してしまい問題が発生するので，ユーザの入力した値にシングルクオートが入っていないことを確かめることにする[1]．プログラムは例えば図1.5の2番目の関数のようになる．

次にこのプログラムを拡張して出版年も指定して検索できるようにする．検索クエリは`"SELECT * FROM books WHERE title='" + title + "' AND year = " + year;`のような文字列の連結で構築すればよい．ユーザ入力値yearに対する検証も追加すると，プログラムは図1.5の3番目の関数のように書ける．しかしながら，このプログラムはユーザ入力値の検証を行っているにも関わらずSQLIAに対して脆弱である．例えば攻撃者がyearに

```
2000;DELETE FROM books;
```

という文字列を入力した場合，入力にはシングルクオートは含まれていないので入力値の検証は成功するにも関わらず，生成されるSQLは

```
 SELECT * FROM books WHERE title='Title' AND year =
2000;DELETE FROM books;
```

となり，開発者が予期していないSQL文`DELETE FROM books;`が実行されてしまいSQLIAが成功してしまう．このような脆弱性が生じてしまうのは，yearに入力された文字列はSQLの整数リテラルとして利用されているため，SQLIAを防ぐために必要な入力値検証は入力値が数字のみで構成されているかどうかの検証であったにも関わらず，実際にはシングルクオートが含まれていないかどうかという全く無意味な入力値検証が行われてしまったことに原因がある．もちろん，こうした問題は非常に当たり前なことではある．しかしこの例はただ一つの検証の仕組みを利用してすべてのSQLIAを防ぐことはできないということを示しており，入力値検証のみを利用したSQLIA対策は複雑で開発者によるミスを招き，脆弱性を引き起こす危険性が高い手法であるということができる．

1.5.3　エスケープ処理

もう１つの原始的ではあるが重要な対策手法は，ユーザの入力値に対して適切な変換を施すことである．例えば1.4.1で例示したSQLIAによる認証のバイパスの例では，攻撃者はシングルクオートを入力することでSQLの文字列リテラルを終了させ，その後ろにSQLを直接入力することで，where節の条件式を恒等式にしてしまうことでパスワードを知らないにも関わらずシステムにログインすることが可能となっていた．こうした問題が起こってしまった原因は，ユーザの入力に含まれていたシングルクオートがSQLの文字列リテラルの中ではシングルクオートという文字として扱われず，文字列リテラルの終端であると解釈されてしまったからである．そのため，こうした問

[1]ただしRDBMSがサポートするSQLの実装によってはこれだけでは不十分な場合もある

題を解決するにはシングルクオートをSQLの文字列リテラル上でシングルクオートとして扱われる特殊な文字（列）に変換することが必要になる．

SQLでは（シングルクオートで囲まれた）文字列中に出現する文字列「’’」（シングルクオート２つ）がシングルクオートを表す特殊文字な文字列として扱われため，シングルクオートを正しく処理するためにユーザ入力中の「’」をすべて「’’」に変換する．またMySQLやPostgreSQLなどの一部のSQLの実装では\がエスケープ文字として扱われ\0や\nなどの文字列が特殊な文字として解釈されるので，「\」も文字列リテラル上でバックスラッシュを表す文字列「\\」に変換する必要がある[2]．こうした変換はエスケープ処理と呼ばれる．図1.3で例示したプログラムをエスケープ処理を用いて書き直すと図1.6のようになる．このプログラムはSQLIAに対して安全であり，またユーザ入力値の検証によって脆弱性を防止する図1.4のプログラムのようにユーザが入力できる値を制限する必要もない．

```
private string escape(string str)
{
        return str.Replace("'", "''");
}

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

図 1.6: エスケープ処理による脆弱性対策

1.5.4　エスケープ処理による対策の問題点

脆弱性対策としてのエスケープ処理のもっとも致命的な欠点は，エスケープ処理を行う関数の呼び出しを忘れてしまうとシステムがSQLIAに対して脆弱になってしまう点である．もちろん，エスケープ処理を行わなければ何の対策も施されていないことになるのでこれは至極当然のことではある．しかしながら，そもそも脆弱性やバグというのは開発者の無知のみならず単純なミスからも生じているということを考えれば，こうした点についても脆弱性対策を考える際の問題としてとらえることは重要であるといえる．

エスケープ処理による脆弱性対策の第2の問題はユーザの入力をすべてエスケープ処理すればプログラムが安全になるわけではないという点である．これは1.5.2の2

[2]逆に，\をエスケープ文字として扱わない実装 (SQL Server, Oracle, IBM DB2など) では無意味にユーザの入力が改変されることになるので，変換してはならない．

つ目の問題とほぼ同じ問題である．例えば図 1.5 の本をタイトルと出版年でユーザ入力値 title と year を図 1.6 の関数 escape を利用してエスケープ処理を行っても SQLIA を防ぐことはできない．そのため単に「ユーザの入力をエスケープ処理する」という非常に単純なルールではインジェクション攻撃に対する安全性を保証することはできない．

1.5.5　原始的な対策のまとめ

SQLIA や XSS などのインジェクション攻撃に対する脆弱性が生じてしまうのはユーザ入力値から SQL や HTML を構築する際に行われる文字列処理にバグがあることが原因である．そのため，ユーザ入力値に対する適切な検証と変換を行い SQL や HTML の構築処理が抱えるバグを排除することでインジェクション攻撃に対する脆弱性は防ぐことができる．しかし，ユーザ入力値に対する検証と変換を適切に行えば脆弱性は生じないと単純にいっても，そもそも何が適切な入力値検証であり何が適切な入力値変換であるのかというのは自明な問題ではなく，実際にプログラムを記述するには 1.5.2 と 1.5.4 で述べたように，プログラムの仕様やユーザ入力値の利用のされ方，場合によっては RDBMS の実装に合わせて入力値の検証・変換の方法を検討しなくてはならない．そのため，インジェクション攻撃に対して安全なプログラムを記述するための単純なコーディングルールを作成することができない．

また入力値検証やエスケープ処理のみを利用して安全性を確保する場合，安全なプログラムを記述するためのコーディングルールが単純化できないため，Web アプリケーションの開発者はインジェクション攻撃に対する脆弱性をプログラムに埋め込んでいないかを常に配慮しながらプログラムを記述しなければならない．しかし，そもそも Web アプリケーションの開発者は必ずしも Web アプリケーションセキュリティの専門家ではないため，こうした要求は開発効率の低下や脆弱性発生につながりかねない．そのため単純なユーザ入力値検証・変換による脆弱性対策よりも容易に，インジェクション攻撃に対して安全なプログラムを開発するための手法が必要となる．

なお，ここから先は特に SQLIA に対する対策手法に主眼を置いて話を進める．ただし対策手法のベースにある考え方は，クロスサイトスクリプティングなどの SQL 以外の言語へのインジェクション攻撃に対する対策にも応用可能である．

Chapter 2　関連研究・関連技術

ユーザ入力値から SQL を構築する際の処理に不備があることが SQL インジェクション攻撃に対する脆弱性が生じる原因である．そのため SQLIA に対する脆弱性の解決策はユーザ入力値の適切な検証と適切な変換が全てであり，理論的には 1.5 で述べた原始的な脆弱性対策を正しく行えばあらゆる SQLIA の脆弱性は生じ得ない．しかしながら，現実には 1.1 で述べたように多くの脆弱性が開発時に修正されず放置されている．そのため 1.5 で紹介した原始的な手法よりも進んだ脆弱性対策手法が必要であり，実際に数多く存在する．本章ではそうした技術および研究について言及する．まずはじめに SQLIA に対して安全なプログラムを 1.5 で述べた手法よりもより容易かつ確実に作成するための手法について述べる．その後，プログラム中に SQLIA に対する脆弱性が存在している場合にシステムを攻撃から守るために，攻撃を検出する手法および脆弱性を検出する手法について述べる．

2.1　より容易に安全なプログラムを記述する技術

第 1 章の説明の中で例示したプログラムはいずれも，ユーザの入力値を SQL 中のリテラル（定数）として利用していた．そしてその際に，適切な検証もしくは変換を行うことなく文字列の連結によってユーザの入力を SQL の一部として利用していたため，SQLIA に対する脆弱性が生じてしまっていた．そのためインジェクション攻撃を防ぐためには，ユーザ入力値によって動的に決定される SQL 中のリテラルを 1.5 で紹介したユーザ入力値の検証と変換による手法よりも容易かつ安全に設定できる手法が重要となる．

2.1.1　バインディング機構

アプリケーションを作成する際，プログラマがソケット通信などのローレベルのネットワーク API を利用してデータベースと直接通信することはなく，データベース接続用の API（Java であれば JDBC[8]，.Net Framework であれば ADO.NET[9] など）を通じてデータベースとのやりとりが行われる．こうした API は通常，実行時にユーザの入力に基づいて決定される SQL のリテラルの部分を値が未割り当てであることを表す「プレースホルダ[1]」と呼ばれる記号にしておき，実行時にプレースホルダをリテラルで置き換えるための関数を呼び出して値を設定する機能を提供している．例えば，図 1.5 で例示した本をタイトルと出版年で検索するプログラムを ADO.NET で提供されているこうした機能を利用して書くと図 2.1 のように書くことができる．この例で

[1] パラメータとも呼ばれる

はSQLを記述する際に，ユーザの入力によって決定される検索条件のタイトルの値と出版年の値の部分を@title, @yearというプレースホルダにしておき，実行時に

```
command.Parameters.Add("@title", title);
command.Parameters.Add("@year", year);
```

という処理を行い，パラメータ@title, @yearをユーザが入力した値title, yearで置き換えている．

```
public Data SearchBook(string title, string yearString)
{
    int year = toInteger(yearString);
    string query = "select * from books where title = @title"
                 + " and year = @year";
    using (SqlConnection conn = new SqlConnection(connectionString))
    using (SqlCommand cmd = new SqlCommand(query, conn))
    {
        cmd.Parameters.Add("@title", title);
        cmd.Parameters.Add("@year", year);
        conn.Open();
        return cmd.ReadData();
    }
    return result;
}
```

図 2.1: バインド機構を利用したプログラムの例

ここで重要でなのは，プレースホルダを値で置き換える際に単純な文字列置換が行われているのではなく，追加された値がSQLのリテラルとしてSQL中に挿入されているという点である．例えば関数の引数titleの値が“Effective C++”という文字列であった場合，単に@titleが“Effective C++”で置換されて“title = @title”が“title = Effective C++”になるのではなく，@titleがSQL上での文字列リテラルを表す文字列“’Effective C++’”で置換され，“title = @title”が“title = ’Effective C++’”に変換される．同様に，たとえtitleの値が“Hacker’s Delight”のようにシングルクオートを含んでいる場合でも，適切にエスケープ処理が行われSQL上での文字列リテラル“’Hacker’’s Delight’”に変換された上で“@title”と置き換えられる．そのためこうした機構を利用すれば，SQLIAを引き起こす危険な文字を含む文字列も問題なく処理することができる．そのため，攻撃者は入力値titleを利用してSQLIAを行うことはできない．

また，この処理はプレースホルダにバインドされる値のデータ型合わせて行われるため文字列以外のデータをSQLに埋め込みたい場合はユーザ入力値を文字列のままAPIに渡すのではなく，埋め込みたいデータの型に変換してからAPIに渡すことになる．そのため，ユーザが入力した出版年yearString（文字列型）はyear（整数型）に変換されてからAPIに渡されている．そして整数への変換処理が行われるため，攻撃者がSQLIAを試みようとしてユーザ入力値yearStringに“2000;DELETE FROM books;”のような文字列を入力したとしても，整数への変換処理でエラーが発生しプログラムが中断する．そのため，攻撃者はyearStringを利用してSQLの改竄を行うこともできない．

このように，バインド機構を利用したSQLの構築方法は

1. バインドした値をライブラリがその型に基づいてSQL上での適切な値に自動的に変換してくれるため，文字列だけでなく整数，浮動点小数，日付などの値についても型の違いを考慮することなく全く同じ機構でバインドすることができる．そのため単なる文字列処理でSQLを構築する場合に比べてプログラムの作成が効率的になる．またSQLがプレースホルダを含んだ形で，ソースコード上で分断されずに記述されるためどのようなSQLが生成されるのか把握がしやすく可読性が高くなる．
2. 「SQLは決して文字列の結合で構築せずに，ユーザ入力値に基づいて動的に決まるリテラル部分はプレースホルダにしておき実行時に値をバインドすることでSQLを生成する」というシンプルなコーディング規則に従っていれば，SQLIAに対して安全なプログラムが記述できる．また，リテラルをSQLに埋め込むための処理が入力値の適切な検証・変換処理とまとめられているため，入力値検証用関数の呼び忘れなどの単純ミスによって脆弱性を埋め込んでしまう恐れが小さい．

といった長所を持っている．そのためバインド機構を利用したSQLの構築方法は開発効率・セキュリティの両面で，1.5で紹介したユーザ入力値に対する検証・変換と文字列の連結でSQLを構築する手法に比べて優れている．

2.1.2　研究

安全なプログラムを記述するための手段は学術研究としてもいくつか提案されている．SQL DOM[10]はSQLを文字列の連結で作成するのではなく，構文木を専用のAPIを利用して組み立てることで作成することでSQLIAを防ぐ方法である．この手法も，プログラマが文字列処理を直接行うことなくユーザの入力に基づいてSQLを構築することでSQLIAに対する脆弱性が生じるのを防ごうとするという点は2.1.1で紹介したバインド機構を用いる手法と共通している（図2.2）．ただしこの手法ではプログラム中で利用するSQLを全て手続きによって組み立てなくてはならないため，SQLという「言語」を利用している意義が失われプログラムの記述性および可読性が低下する可能性があり，複数のテーブルを結合するような複雑なSQLを利用する場合に本当に実用に耐えうるのかは疑問である．また文献[11]では，SQL中のリテラル部分以外の構造が動的に変化する場合にもSQLIAに対して安全なプログラムを記述しやすいAPIを開発者に対して提供することで，開発者がバインド機構を利用する場合よりも簡単なルールを守っていれば安全にプログラムを記述することができるようにする手法が提案されている．

文献[12, 13]ではプログラマがSQLIA攻撃を検知し防ぐための工夫をプログラムに施すことでSQLIAを防ぐ手法が提案されている．文献[12]のSQLrandはSQLのキーワード（予約語）の末尾にランダムな整数を付け加えた特殊なSQLを用いることで，攻撃者がSQLのキーワードを挿入することを不可能にしプログラムの安全性を保とうとする手法である．例えばランダムな整数として123を選んだ場合にはSQLは図

```
29  public string GetCustomers( string companyName,
30                              string contactName,
31                              string city,
32                              string region,
33                              string country )
34  {
35    CustomersTblSelectSQLStmt sql =
36      new CustomersTblSelectSQLStmt();
37
38    // add a where condition for CompanyName
39    // if the user specified a CompanyName
40    if ((companyName != null)
41      && (companyName.Length > 0))
42    {
43      sql.AddWhereCondition(
44        new CompanyNameWhereCond(companyName));
45    }
46
47    // similar code would be placed here for each
48    // of the other five possible conditions
49
50    return sql.GetSQL();
51  }
```

図 2.2: SQL DOM による記述例

2.3 のようになる．確かに，このような特殊な SQL を利用してプログラムを構築していれば SQLIA を行うのは難しくなる．ただし SQL の記述性および可読性が低下するため使い勝手が悪いのはもちろんのこと，そもそも SQLIA は，必ずしも SQL のキーワードを利用しなくても起こりうるので脆弱性対策の技術として本当に十分なのかは疑問が残る．文献 [13] で提案されている手法は，SQL を文字列の結合で構築する際に，SQL 上で単一のリテラル（文字列や数字）になるはずの文字列の前後に特殊なマーカをつけ，SQL 実行時にマーカで囲まれた部分が単一のリテラルになっているかどうかをチェックし，単一のリテラルになっていない場合は SQLIA が起こったとみなして実行をキャンセルすることで SQLIA を防ぐという手法である（図 2.4）．ただし，こうした SQLIA を検知し防止するためにこうした特殊な手法を用いたとしても，結局ユーザの入力を正しく処理するためには 1.5.3 で述べたようにユーザの入力に対して何らかの適切な変換を施す必要があり，そうした処理を効率的に行うには 2.1.1 のバインド機構のような技術が結局のところ必要となる．

```
select123 gender, avg123 (age)
   from123 cs101.students
      where123 dept = %d
   group123 by123 gender
```

図 2.3: SQLrand による記述例

2.1.3　まとめ

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

Traditional Method

```
Connection c = DriverManager.getConnection(strDBconn);
Statement s = conn.createStatement();
String q = ''SELECT * FROM reports WHERE id='' + id;
ResultSet RS = s.execute(q);
...
RS.close();
conn.close();
```

Guarded Method

```
Connection c = SafeDriverManager.getConnection(strDBconn);
Statement s = conn.createStatement();
String q = SQLGuard.init() + ''SELECT * FROM reports WHERE id='' + SQLGuard.wrap(id);
ResultSet RS = s.execute(q);
...
RS.close();
conn.close();
```

図 2.4: 文献 [13] の提案手法による記述例

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，もしくは全く利用しなかった場合，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全く機能しない．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とすることは，アプリケーションの安全性を完全にプログラマ任せにしてしまうということを意味しており大いに問題がある．

2.2　攻撃の検出

2.2.1　侵入検知システムによる保護

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挙げられる．こうしたシステムを利用してインジェクション攻撃を防ぐ場合，あらかじめ攻撃とみなすべきメッセージのパターンをシグネチャとして定義しておき，運用時にWeb アプリケーションに送られてくるメッセージを１つ１つシグネチャとマッチするかどうかを調べ，シグネチャとマッチするものを攻撃とみなしてブロックする手法が一般的となる．例えば広く利用されているIDS の１つであるSnort[14] を利用してSQLIA を引き起こすことがある，“’”，“--”，“#” を攻撃とみなし排除するためのシグネチャは/(\%27)|(\’)|(\-\-)|(\%23)|(#)/ix のようになる[15].

ただし，Web アプリケーションの手前にフィルタを設けて攻撃を検出・排除するこうした手法は，プログラム自体をブラックボックスとみなしプログラムの外部にフィルタを配置している点は異なるものの，基本的な考え方は1.5.1 で紹介したユーザ入力値検証によるSQLIA 対策と同じである．そのため1.5.2 で述べたのと同様の問題，すなわち

- どのような入力を攻撃とみなし，拒否して構わないかはシステムの仕様によって依存する
- どのような入力が攻撃となるかは各々の入力値がプログラム中でどのように利用されるかに依存する

という 2 つの問題が存在する．そのため，誤検出・検出漏れを全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い．また，あらゆる Web アプリケーションを保護することができるシグネチャを作成することは不可能であり Web アプリケーションの特性に合わせてシグネチャを作成しなくてはならない．

さらに，上で示した例で単に “’”，“--”，“#” を含むメッセージを検出するためだけのシグネチャが/(\%27)|(\’)|(\-\-)|(\%23)|(#)/ix という複雑なものになることが示すように，何を攻撃としてみなすべきかが明確になったとしてもそれに対応するシグネチャを正確に記述することもそれほど容易な作業ではない．それゆえ，こうしたシステムのみを利用して SQLIA から Web アプリケーションを完璧に防御するのは容易ではない．

2.2.2　機械学習を利用した攻撃検出

シグネチャベースで攻撃を検出し Web アプリケーションを SQLIA から保護する手法の問題点は，攻撃を正確に検出するには Web アプリケーション毎にシグネチャを人手で定義しなければならないところにある．そのため，アプリケーションを事前に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作成することでこうした問題点を解決しようという研究が存在する．

攻撃検出を行うためのルールを自動生成する手法の 1 つに，機械学習を利用して攻撃検出のルールを作成する手法が存在する [16, 17]．これら手法では，まずアプリケーションが正常に実行されている際にアプリケーションからデータベースに対して発行される SQL を観察し，正常時に利用される SQL のパターン（モデル）をあらかじめ機械学習する．そして，実際に運用する際にアプリケーションからデータベースに発行される SQL が学習しておいた正常な SQL のパターンと一致するかどうかを調べ，一致しない場合には SQLIA が発生したとみなし SQL の実行を拒否することでアプリケーションを SQLIA から保護する．ただし機械学習を用いたこうした手法は，検知精度が学習に利用されるデータセットの質に左右されやすく，適切な学習データセットを利用しないと高い確率で検出漏れ・誤検知を引き起こしてしまう．

2.2.3　プログラムの静的解析を併用した攻撃検出

アプリケーションを解析することで攻撃検出のルールを自動生成する手法のもう 1 つのグループは，プログラムの静的解析を利用する手法である．こうした手法の研究としては文献 [18, 19] が存在する．

AMNESIA[18] はまずプログラムに対する静的解析を行い，プログラム中のデータベースを利用している各部分で実行される SQL を表現するモデルを有限状態オートマ

トンとして作成する．次にプログラム中のSQL実行部分を書き換え，SQLを実行する前に各SQLが静的解析によって得られた有限状態オートマトンで受理されるかどうかを調べ，受理される場合のみ実行を行うようにプログラムを変更する．例えば図2.5のSQLIAに対して脆弱なプログラムの場合，静的解析によって下から3行目のexecuteで実行されるSQLのモデルとして図2.6の有限状態オートマトンが作成される．そしてプログラムはexecuteでSQLを実行する前にexecuteに渡されるSQLが図2.6の有限状態オートマトンで受理されるかどうかをチェックするように書き換えられる．この有限状態オートマトンは，通常のユーザ入力によって得られるSELECT info FROM userTable WHERE login = 'doe' AND pass = 'xyz'のようなSQLは受理するが，例えばSELECT info FROM userTable WHERE login = '' OR 1 = 1 --' AND pass = ''のようなSQLIAによって改竄されたSQLは受理しないため，改竄されたSQLの実行がブロックされることになり，SQLIAからアプリケーションを保護することが可能となる．

```
public ResultSet getUserInfo(String login, String password) {
    Connection conn = DriverManager.getConnection("MyDB");
    Statement stmt = conn.createStatement();
    String queryString = "";
    queryString = "SELECT info FROM userTable WHERE ";
    if ((! login.equals("")) && (! password.equals(""))) {
        queryString += "login='" + login + "' AND pass='" + password + "'";
    }
    else {
        queryString+="login='guest'";
    }
    ResultSet tempSet = stmt.execute(queryString);
    return tempSet;
}
```

図 2.5: AMNESIAの利用例

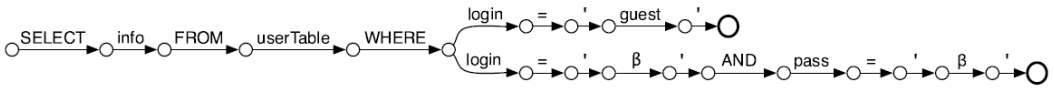

図 2.6: 図2.5のプログラムに対応するSQLクエリモデル

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するため，プログラム毎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える場合には，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，アプリケーションの構造によってはプログラムの静的解析が正確に行えず攻撃の誤検出や検出漏れが生じる危険性がある [6, 11].

2.2.4　動的なデータフロー追跡による攻撃検出

文献 [20] では，外部から入力されるユーザ入力値の前後に “マーカー” として機能する特殊な文字列を挿入し，SQL実行時にそのマーカーを利用してSQLに含まれるユー

ザ入力値の追跡を行うことで，ユーザ入力値によってSQLの改竄が行われていないかを検証する手法が提案されている．この手法は，アプリケーションが正常に実行されている場合にはユーザ入力値に基づく情報はSQL中のリテラルの内部に閉じ込められるはずであるという考えに基づいており，ユーザ入力値がSQL上で単なる文字列として扱われる場合にはうまく機能すると考えられる．しかし現実には，ユーザ入力値は整数型や浮動小数点型に型変換されて利用される場合や，ユーザ入力値の長さに意味がある場合，ユーザ入力値が分割される場合などがあり，様々な利用のされ方をするため実用的なアプリケーションでは “マーカー” の挿入が障害を引き起こす可能性が高い．

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一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上で非常に重要な技術である．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．

2.3.1　実際に攻撃を行い脆弱性を発見する手法

ブラックボックス的な脆弱性検出技術としては，Webアプリケーションのさまざまな種類の脆弱性に対して検査ツールが存在しており，SQLIAに対する脆弱性を発見するためのツールも存在している [21, 22]．こうしたツールはシステムに対して実際に攻撃を行ったり，攻撃に類するリクエストを送りつけた際にシステムに対して異常が発生しないかを検査し脆弱性を発見する．ただしブラックボックス的な手法であるためブラックボックステストと同様，検査の網羅性を保証することは難しく，脆弱性の見落としが生じる危険性がある．

2.3.2　データフロー解析による脆弱性の発見

プログラムを解析することで脆弱性を発見するホワイトボックス的な脆弱性発見手法としては，実行時に動的にデータの流れを解析し危険な処理を検出する汚染検出モード [23] が非常に有名である．汚染検出モードでは，プログラムの外部から与えられたユーザ入力は “汚染されている” とみなされ，汚染されている値に基づいて計算されるに値にも “汚染” が伝播する．同時に，SQLの実行のような危険を伴う処理を行う関数の引数として汚染された値が利用されることが禁止され，明示的に汚染を解除する関数を呼び出すか，入力値をフィルタリングしない限りはユーザ入力値に基づく情報が危険を伴う関数の引数として利用されることが防止される．この機能が実装されて

いる言語としてはPerl[24]やRuby[25]が存在する．ただし汚染検出モードは単にユーザ入力が何のフィルタリング処理もされないままSQLの一部として利用されるような危険な処理を検出するだけであり，SQLIAに対する脆弱性そのものを検出するわけではないため，例えば入力値の検証処理に不備があるために脆弱性が生じてしまった場合などは脆弱性を検出することはできない．また同様の汚染検出による脆弱性の発見を静的解析によって行う手法としてPQL[26]という解析ツールを利用した研究[27]なども存在する．

Chapter 3　研究概要

3.1　モチベーション

第 2 章で述べた技術はいずれも SQL インジェクション攻撃対策として有益な技術である．しかしながら，2.1 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 2.3 で述べた攻撃検出技術や 2.2 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，ここまでで述べてきたように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技術が本質的に抱えてい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全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．

3.2　概要

多くの Web アプリケーションにおいて，プログラムの内部で 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非常に間違いを誘発しやすい処理である上に，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．それにも関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 1.4.1 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

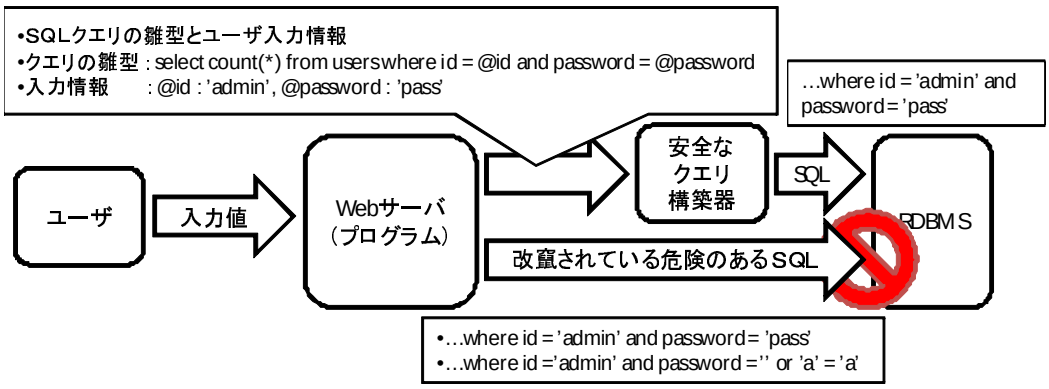

図 3.1: 本提案手法における Web アプリケーションの構造

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 3.1)．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 の雛型と，雛型から SQL 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 を作成，それを SQL の評価器に受け渡し実行する機能を RDBMS に追加する（図 3.1 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 で実行させることがないように，SQL 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 3.1 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 1.4.1 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得なくなる．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，それに伴って 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新しい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のみを利用してプログラムを記述すること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

- SQL の雛型に何を利用するのか．どういった処理で SQL の雛型から実際に実行する SQL を生成するのか．
- SQL 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ざんされてしまうと，いかに SQL の雛型から SQL 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 SQL 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，攻撃者による改ざん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．そこで第4章ではこうした課題が存在することを踏まえた上で，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第5章で言及する．

Chapter 4　サーバサイド（RDBMS）の構成

4.1　SQLの雛型とSQLの構築処理

本提案手法ではSQLを安全に構築する手段を利用することでSQLインジェクション攻撃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を安全に構築する処理をどのように設計するかが非常に重要となる．もしもSQLの雛型から実際に利用したいSQLを作成する処理の自由度が大きすぎると，攻撃者にSQLに対する任意の改竄を許す可能性を与えてしまうことになる．一方でSQL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SQL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SQL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．

4.1.1　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の頭文字を取って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4.1）．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，実際に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・検索条件の繰り返しのみである．このようにWebアプリケーションで処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・条件の繰り返し・ソート順序に限定することができるため，SQLの雛型ではこれらの要素のみを未定にしておき，SQL構築処理でユーザ入力値に基づく情報からこれらの要素を決定できるようにSQLを安全に構築する機能の設計を行う．次に上の議論を踏まえた上でSQL構築処理の具体的な構成方法について述べる．

4.1.2　詳細

SQL を拡張して上で述べた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 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 の雛型を構文木に変換する．そして雛型から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 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 4.1）．

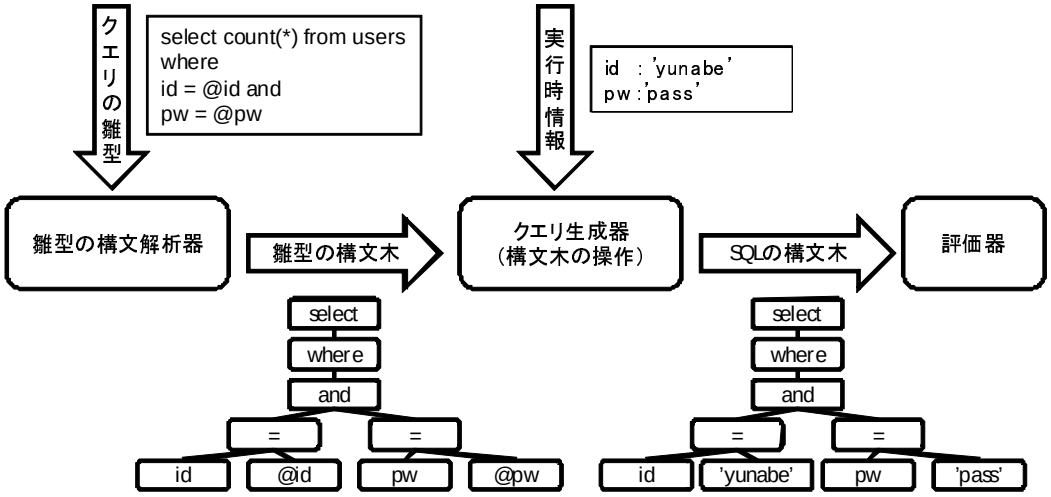

図 4.1: SQL の雛型から SQL を生成する処理の概要

バインディング

すでに 3.2 の図 3.1 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 2.1.1 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，例えば SQL の文法上文字列リテラルの

表 4.1: 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

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

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．そのため例えば`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ない．

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の雛型からSQL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SQL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，SQL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*では無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`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`で修飾するできるようにし，`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[1]．図4.2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を作成する場合の例をあげる．このWebアプリケーションでは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

[1] このようにすると`OPTIONAL`修飾されたプレースホルダにNULLをバインドすることができなくなるが，そもそもSQLは3値論理と呼ばれる論理システムを採用しており[28, 29]，条件式中にNULLをリテラルとして含むSQLをアプリケーション中で利用する必要はほとんど無いため問題にならない．逆に`OPTIONAL`で修飾していないプレースホルダに誤ってNULLをバインドしてしまうことを防止するため，6.2で紹介する本提案手法の実装では明示的にキーワード`NULL`で修飾されたプレースホルダ以外にNULLがバインドされることを禁止している．

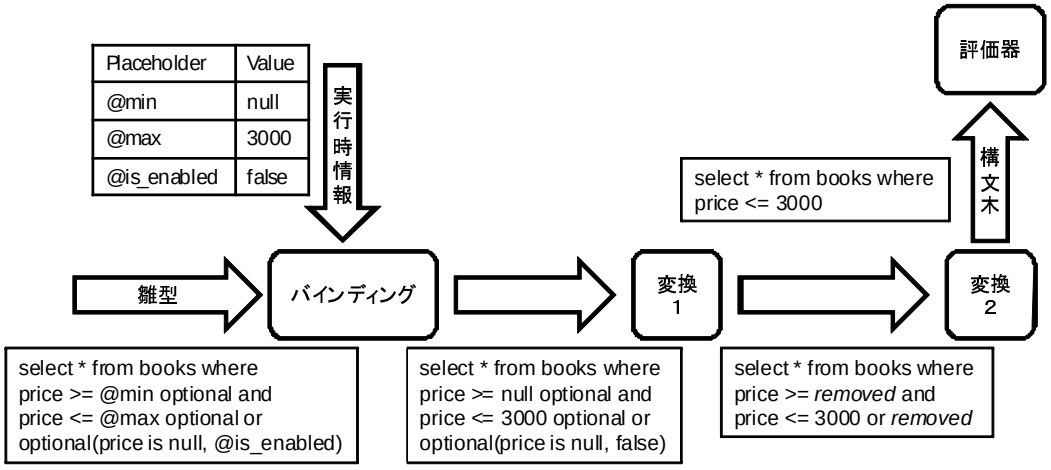

図 4.2: 実行時に有無が決定される条件式を含む構文木の変換処理

どうかを選択することが可能である．そのため図 4.2 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 SQL の雛型が利用される．そして例えばユーザが価格が 3000 円以下の本を検索した際には，価格の上限が 3000 なので`@max` に 3000 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 “`price >= 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 3000 円以下の本を検索する SQL “`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，条件式を真とみなすべきか偽とみなすべきか判断することができないのでエラーとなり SQL の生成に失敗する．

条件の繰り返し

条件の繰り返しをサポートするために，SQL 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，SQL 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．例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 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 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，任意の式の配列をバインドすることはできない．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．

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 SQL 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 SQL 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，攻撃者による改ざんが起こりうる *expr* によって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全くない．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` を付与したもの *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 SQL が生成されるように雛型から SQL を生成する規則を定める．例えば `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．

4.2　安全に SQL を構築する機能

本提案手法ではプログラマは，RDBMS が持っている SQL を直接受け取り評価する機能の代わりに SQL の雛型と雛型から SQL を作成するための情報を受け取り SQL を作成・評価する機能を利用してプログラムを記述する．そのために RDBMS に存在する文字列を SQL 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 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 SQL を SQLIA に対して安全な方法で作成・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 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 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ーフェイ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．

4.2.1　外部から SQL 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 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，SQL 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 SQL 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 SQL の雛型がアプリケーション実行前にあらかじめ作成されたものではなく，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 SQL 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 SQL の雛型が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 の雛型を受け取る際に，その雛型に対するメッセージ認証コード（MAC）も一緒に受け取けとり，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 SQL の作成・評価の処理を行うようにプロトコルを設計する．また RDBMS に SQL 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に対しては MAC が取得できるため SQL 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に対しては MAC が取得できないため SQL の雛型として利用できないこととなる．従って SQL 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．

4.2.2　RDBMS 内部で作成された SQL を評価する機能の拡張

RDBMS は通常，外部から受け取った SQL 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 SQL 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 [7]．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 SQL の雛型とプレースホルダにバインドする値から SQL を生成・実行する機能を追加する．拡張された SQL 評価のための機能は SQL 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．例えば，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 SQL 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 SQL 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 [7] 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は 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（図 4.3）．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 SQL の雛型として受け取るため，SQL 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 SQL 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，SQL の雛型の中で `SEXECUTE` 文のような拡張機能が受け

```
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 4.3: 脆弱なストアドプロシージャの例 [7]（左）と提案手法による安全な記述例（右）

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 SQL の雛型として利用できてしまい，安全性を保証するための枠組みが崩壊してしまう[2]．こうした問題を避けるためにも 4.1.2 バインディングで述べた「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 文の他に 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 SQL を安全に構築するする機能を RDBMS に追加するかについて述べた．しかし安全に SQL 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通り SQL 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 SQL を RDBMS に実行させてしまうと RDBMS の拡張を行った意味が全くなくなってしまう．そこで何らかの方法で RDBMS の SQL 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 SQL 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 が有効な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 SQL 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 SQL を評価する機能の利用
- 4.2 の (1) で導入した SQL に対する MAC を取得する機能の利用

[2]例えば SEXECUTE ’SEXECUTE @query’ @query = *query* とすることで SQLIA に対して耐性があるはずの SEXECUTE 文だけを用いて任意の SQL（つまり改竄を受けている可能性のある）*query* が実行できてしまう．

- 自身のユーザ権限情報を操作して SAFEMODE を無効にすること

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 を必ず有効にするようにする．こうすることで，プログラマは RDBMS が提供する安全に SQL 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 SQL 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 が無効なユーザアカウントで RDBMS にログインすれば全ての機能を今まで通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．

4.4　SQL インジェクション攻撃に対する耐性保証

4.4.1　耐性保証の概要

本提案手法では，4.1.2 で述べた

- SQL 中のリテラルの決定
- 実行時に有無が決定される条件の制御
- 条件の繰り返し
- ソート順序の制御

という Web アプリケーションを開発するのに必要な最低限の機能のみを利用して SQL の雛型から SQL を安全に構築することができる SQL 生成器をアプリケーションの外部に用意し，その安全な SQL 生成器をプログラマが正しく利用していることを保証することで Web アプリケーションの SQLIA に対する耐性の保証を行う．そしてその耐性保証のために，4.2 や 4.3 で述べた様々な機能や制約が RDBMS に追加される．この節では，Web アプリケーションの SQLIA に対する耐性がどのように保証されるのかについてまとめ，上述した機能および制約がどのように役立つのかを整理する．

まず我々が本提案手法を利用して保証したい，Web アプリケーションの SQLIA に対する安全性は端的に言うと次の命題で表すことができる．

命題 4.1. *Web アプリケーション実行時に RDBMS で実行される SQL は SQL インジェクション攻撃による改竄を受けていない．*

我々の提案手法の枠組みにおいては，この命題は次の 1 つの仮定と 1 つの補題によって保証されることになる．

仮定 4.1. *Web アプリケーションは必ず特殊権限 SAFEMODE が有効なユーザアカウントを利用して RDBMS にログインしている．*

補題 4.1. *SAFEMODE が有効なユーザで RDBMS にログインしている際に実行される SQL は SQL インジェクション攻撃による改竄を受けていない．*

そして，この補題 4.1 は次の 1 つの仮定と 2 つの補題が成立すれば成立する

仮定 4.2. *SAFEMODE* が有効なユーザで *RDBMS* にログインしている際に実行される *SQL* は *SAFEMODE* が無効なユーザが *RDBMS* に登録した *SQL*[3] もしくは安全な *SQL* 生成器によって *SQL* の雛型から作成された *SQL* のみである．

補題 4.2. *SAFEMODE* が無効なユーザが *RDBMS* に登録した *SQL* は *SQL* インジェクション攻撃による改竄を受けていない．

補題 4.3. *RDBMS* に *SAFEMODE* が有効なユーザでログインしている際に，安全な *SQL* 生成器によって *SQL* の雛型から作成された *SQL* は *SQL* インジェクション攻撃による改竄を受けていない．

まず補題 4.2 が成立するには以下の 2 つの仮定が成立すればよい

仮定 4.3. *SAFEMODE* が無効なユーザによって *RDBMS* に登録された *SQL* は静的[4] である．

仮定 4.4. 静的な *SQL* は *SQL* インジェクション攻撃による改竄を受けていない．

また補題 4.3 が成立するには，以下の 1 つの仮定と 1 つの補題が成立すればよい．

仮定 4.5. 静的な *SQL* の雛型から安全な *SQL* 生成器を利用して作成される *SQL* は *SQL* インジェクション攻撃による改竄を受けていない．

補題 4.4. *RDBMS* に *SAFEMODE* が有効なユーザでログインしている際に，安全な *SQL* 生成器に渡される *SQL* の雛型は必ず静的なものとなる．

以上より，仮定 4.1〜4.5 と補題 4.4 が成立すれば，命題 4.1 が成立し，Web アプリケーションの SQLIA に対する安全性が保証されるといえる．そこで

仮定 4.6. *RDBMS* に *SAFEMODE* が有効なユーザでログインしている際に，安全な *SQL* 生成器に渡される *SQL* の雛型は，有効なメッセージ認証コードとともに外部から渡された *SQL* の雛型（*4.2.1* 参照）もしくは *RDBMS* 内部で実行される *SQL* に含まれる *SQL* の雛型（*4.2.2* 参照）のいずれかである．

という仮定が成立すると仮定すると，この仮定 4.6 と仮定 4.2 より，SAFEMODE が有効なユーザで RDBMS にログインしている際に安全な SQL 生成器に渡される SQL の雛型は，

- 外部から有効なメッセージ認証コードとともに渡された SQL の雛型
- SAFEMODE が無効なユーザによって登録された SQL に含まれる SQL の雛型

[3] SAFEMODE が無効なユーザによって，定義されたストアードプロシージャ

[4] アプリケーション実行前にあらかじめ定義されており，アプリケーション実行の影響を受けないという意味

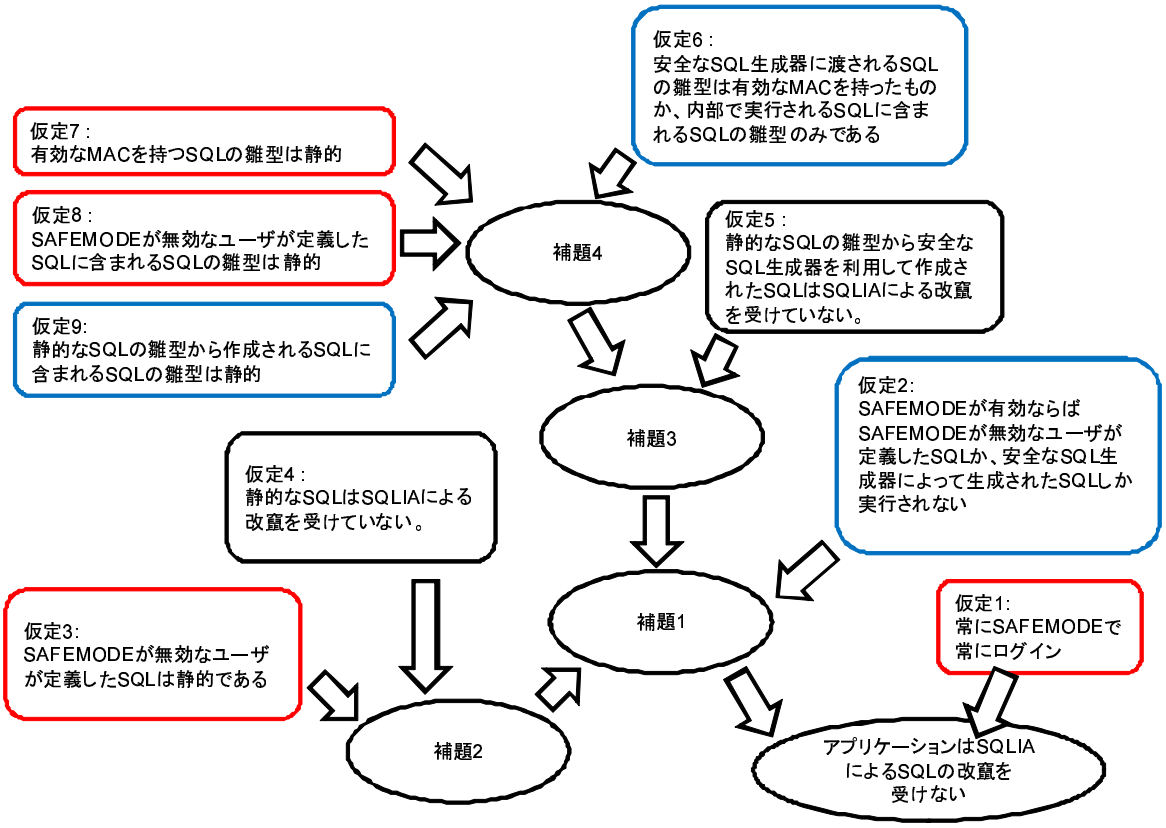

図 4.4: SQL インジェクション攻撃の耐性保証の流れと必要な仮定

- 上記の 2 つに対して，安全な SQL 生成器による SQL の生成と生成された SQL の評価という処理が有限回数行われた結果，安全な SQL 生成器に対して渡された SQL の雛型

のいずれかとなる．そのため補題 4.4 は

仮定 4.7. 有効なメッセージ認証コードが付与された *SQL* の雛型は静的である．

仮定 4.8. *SAFEMODE* が無効なユーザによってデータベース上に登録された *SQL* を評価した際に利用される，*SQL* の雛型は静的である．

仮定 4.9. 静的な *SQL* の雛型から安全な *SQL* 構築器によって生成される *SQL* に含まれる *SQL* の雛型は全て静的である（*4.2.2* 参照）．

という 3 つの仮定が成立すれば成立することとなり，本提案手法を利用して Web アプリケーションの SQLIA に対する安全性が保証されることになる．以上の議論を図として簡単にまとめると図 4.4 のようになる．

4.4.2　各仮定の保証

図 4.4 において青枠の四角で表した仮定 4.2, 仮定 4.6, 仮定 4.9 の 3 つの仮定は本提案手法において行われる RDBMS に対する拡張が，本章で設計した通りに実装されてい

れば満たされる仮定であり，赤枠の四角で表した，仮定 4.1, 仮定 4.3, 仮定 4.7, 仮定 4.8 の 4 つの仮定は Web アプリケーションの運用者が成立することを保証すべき仮定である．また，黒枠の四角で表した仮定 4.4 と仮定 4.5 は 4.1 で設計した SQL の雛型から SQL を安全に構築するための機能と SQLIA がみたすべき仮定である．そのため，Web アプリケーションの運用者が本提案手法の枠組みを利用して Web アプリケーションの SQLIA に対する耐性保証を行いたい場合には，まず本章で行った設計に基づいて拡張が施された RDBMS を Web アプリケーションが利用する RDBMS として用意して，仮定 4.2, 仮定 4.6, 仮定 4.9 と仮定 4.4 と仮定 4.5 が満たされると仮定した上で，仮定 4.1, 仮定 4.3, 仮定 4.7, 仮定 4.8 が成立するように実行環境の設定を行うことになる．

Web アプリケーション運用者が保証すべきこれらの 4 つの仮定のうち仮定 4.3, 仮定 4.8 は「SAFEMODE が無効なユーザで RDBMS にログインしている際に RDBMS 上で行われる操作は，Web アプリケーション上でユーザが行う操作の影響を全く受けない」ということが仮定できれば成立するため，運用者はユーザから操作されるアプリケーションやサービスが RDBMS を利用する際には，必ず SAFEMODE が有効になっているユーザアカウントで RDBMS にログインするように設定を行えばよい．なお 4.3 で述べた，SAFEMODE が有効なユーザが自身のユーザ権限情報を操作して SAFEMODE を無効にすることを禁止するという性質によって，運用者はより確実に SAFEMODE が有効なユーザアカントが必ず利用されていることを保証できるように設計が行われている．また 4.3 で述べたように「SAFEMODE が有効なユーザは MAC の計算が行えない」ように設計が行われているため，MAC を取得するためには SAFEMODE が無効なユーザで RDBMS にログインする必要がある．そのため,「SAFEMODE が無効なユーザで RDBMS にログインしている際に RDBMS 上で行われる操作は，Web アプリケーション上でユーザが行う操作の影響を全く受けない」ということが仮定できれば仮定 4.7 も成立することになる．以上より Web アプリケーションの運用者は

- Web アプリケーションが RDBMS にログインする際には，SAFEMODE が有効になっているユーザアカントが確実に利用される
- SAFEMODE が無効なユーザで RDBMS にログインしている際に RDBMS 上で行われる操作は，Web アプリケーション上でユーザが行う操作の影響を全く受けない

という容易に達成可能な 2 つの仮定[5]が成立していることを保証すれば，Web アプリケーションの SQLIA に対する安全性を保証できることになる．

残された問題は本当に仮定 4.2, 仮定 4.6, 仮定 4.9 と仮定 4.4 と仮定 4.5 が成立するかどうかである．まず仮定 4.2, 仮定 4.6, 仮定 4.9 は RDBMS の拡張が設計通りに正しく実装されていると仮定してしまって良いかということであるので，その実装がある程度広く利用され検証されていれば仮定 4.2, 仮定 4.6, 仮定 4.9 は成立するとしてしまっても現実的には構わない．問題は仮定 4.4 と仮定 4.5 が成立するとみなしてしまって本当に構わないのかである．これについては第 8 章の考察 8.1 で詳しく述べること

[5] 2 番目の仮定が 1 番目の仮定を包含しているので厳密には 2 番目の仮定だけでよい．

にし，ここでは例えば [6] でまとめられているような典型的な SQLIA に関しては仮定 4.4 と仮定 4.5 が成立し，典型的な SQLIA に対するアプリケーションの安全性を保証することができるため，本提案手法を導入することにより SQLIA に対する高いレベル耐性保証が行えると述べるにとどめる．

Chapter 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を受け渡すが，本提案手法ではプログラムがSQLを利用する際には実行したいSQLの雛型と雛型に対するMACそして雛型からSQLを生成するための情報をRDBMSに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すべてのSQL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．そして，本章で取り扱うクライアントサイドの拡張は，本提案手法が提供するSQLインジェクション攻撃に対する耐性保証によって得られる安全性自体には直接関係はないが，開発者からみた実用面での利便性はクライアントサイドにおける拡張の設計次第で大きく左右されることになるため，クライアントサイドの拡張をいかに行うかについての述べることも重要な点である．

なおクライアントサイドの拡張の設計を述べるにあたっては，既存のクライアントサイドの構成とそれをいかに拡張するかについて具体的に述べる必要があるため，本章では特に.NET Frameworkで利用されるADO.NETを利用してRDBMSを利用するアプリケーションを作成する場合に絞って話を進める．ただし本章で述べる内容の多くの部分は，Javaで利用されるJDBCやPythonで利用されるPython Database API[30]などの他のデータベース接続ライブラリを用いてRDBMSを利用するアプリケーションを構成する場合にも適用可能である．

5.1　クライアントサイドの既存の構造

データベース接続ライブラリを通じてRDBMSを利用する場合，まずRDBMSに接続するために必要なサーバ名・ユーザ名・認証情報・データベース名などの設定情報を含んだ「接続文字列」と呼ばれる文字列を設定ファイルから読み込み，読み込んだ設定情報から接続オブジェクトを作成し，接続オブジェクトを通じてRDBMSへの接続を開始する．次に接続オブジェクトからコマンドオブジェクトを作成し，作成されたコマンドオブジェクトにSQLを設定する．そしてコマンドオブジェクトが提供するSQL実行関数を呼び出すことでSQLの実行を行う．また，その際にSQL中の一部のリテラルを未定にしておき，未定な部分に割り当てる値をパラメータとしてSQLとは別にコマンドオブジェクトに渡し，ライブラリの内部でSQLの組み立てを行いSQLを実行することも可能である．例えば1.4.1で説明に利用した，ユーザ名とパスワードが一致するユーザ情報の数を数えることで認証を行うプログラムをADO.NETのライブラリを利用して作成すると図5.1の疑似コードのようになる．また図5.1のプログラムを実行した際に，ライブラリの内部で行われる処理の流れを簡単に図で表すと図5.2の

ようになる[1].

```
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み
    string connectionString = LoadConfiguration("connection_string");
    // 一部のリテラルが未定なパラメータとなっているSQL
    string query = "select count(*) from users where id = @id"
                 + " and password = @password";
    // 設定情報から接続オブジェクトを作成①
    using (SqlConnection conn = new SqlConnection(connectionString))
    {
        // RDBMSへの接続の開始②
        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③
        // コマンドオブジェクトにSQLを渡す④
        using (SqlCommand cmd = new SqlCommand(query, conn))
        {
            // 未定値のパラメータへ値を追加⑤
            cmd.Parameters.Add("@id", id);
            cmd.Parameters.Add("@password", password);
            // SQLの実行⑥
            int result = (int)cmd.ExecuteScalar();
            return result != 0;
        }
    }
}
```

図 5.1: ADO.NET のライブラリを利用したプログラムの例

5.2　提案手法におけるクライアントサイド拡張

5.2.1　拡張の概要

第 4 章で述べたように本提案手法では Web アプリケーションの SQLIA に対する耐性を保証するために，RDBMS は外部から直接渡された SQL の実行は行わず，代わりに外部から SQL の雛型とその雛型が改竄を受けていないことを示すための MAC，そして SQL の雛型から SQL を生成するためのユーザ入力値に基づく情報を受け取り，SQL を安全に生成する機能を利用して SQL を作成し，その SQL を実行するように拡張されている．そこでクライアントサイドのデータベース接続ライブラリも図 5.2 のように実行したい SQL を RDBMS に直接送るのではなく，実行したい SQL の生成に必要な SQL の雛型とその雛型に対する MAC，そして実行したい SQL を雛型から生成するための情報を RDBMS に送り RDBMS 側で SQL の生成を行わせ SQL を実行するように拡張しなくてはならない．この拡張を単純にデータベース接続ライブラリに対して施すと，クライアントサイドで行われる処理の流れは図 5.3 のようになる．また 4.1.2

[1]JDBC の PreparedStatement は本来 SQL を安全に組み立てるために用意された機能ではないため，内部で行われる処理の様子がこの図とは異なるがそれについては割愛する．

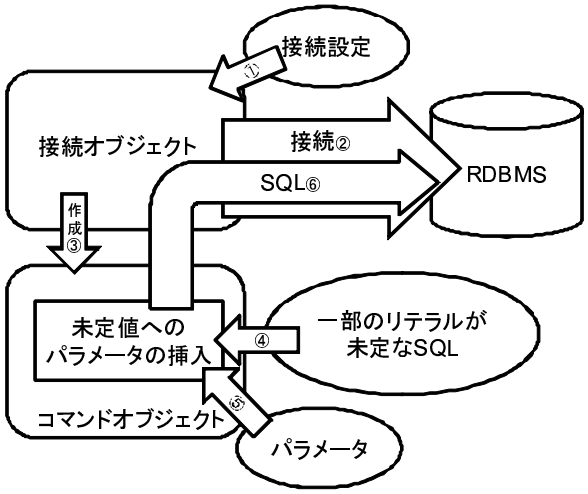

図 5.2: クライアントサイドにおける SQL 実行の流れ

で述べたように本提案手法において，SQL の雛型はユーザ入力情報に基づいて実行時に決定される部分がプレースホルダで代用されている SQL であり，SQL の雛型から SQL を生成するために用いるユーザ情報は SQL の雛型に含まれる各プレースホルダに対して割り当てられる値として与えられる．そのため図 5.3 においてコマンドオブジェクトが受け取る「SQL の雛型」・「ユーザ入力情報」は図 5.2 においてコマンドオブジェクトが受け取る「一部のリテラルが未定な SQL」・「パラメータ」と非常に類似したものとなり，コマンドオブジェクトは開発者に対して提供するインターフェイスを拡張することなく「SQL の雛型」・「ユーザ入力情報」を受け取ることができる．またライブラリの内部で行われる処理については全てライブラリの内部に隠蔽されることになるため，開発者からみえるデータベース接続ライブラリの拡張はコマンドオブジェクトが SQL の雛型に対する MAC を受け取る部分だけとなる．以上の点を踏まえて図 5.3 の構造をしたデータベース接続ライブラリを利用して，図 5.1 のプログラムを書くと図 5.4 のようになる．

5.2.2　MAC の取扱い

ここまでの拡張をデータベース接続ライブラリに施せば，開発者はアプリケーションと RDBMS 間で行われる処理が既存の方法から拡張されていること意識することなく，図 5.4 のようにプログラムを記述することで本提案手法の耐性保証の枠組みの中で開発を行うことが可能となる．しかし 4.2.1 で述べたように，本提案手法で利用される MAC は各 SQL の雛型に対して RDBMS が保持している秘密鍵を用いて作成されるものであり，アプリケーションの運用者によって管理されるべき情報であるため MAC は図 5.4 のようにプログラムの内部に埋め込まれるべきものではない．またデータベース接続ライブラリ自身が SQL の実行を行う必要がある場合があるため，データベース接続ライブラリ自身が利用する SQL の雛型に対する MAC を何らかの方法で設定する必要があるが，図 5.3 の構造ではそれが行えないという問題も存在する．

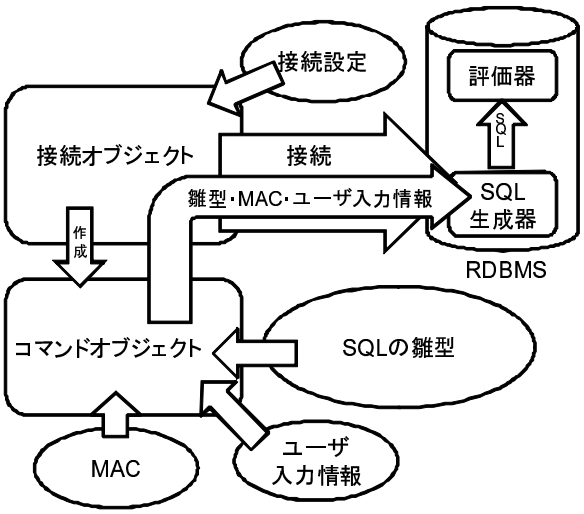

図 5.3: 単純な拡張を行った場合の実行の流れ

```
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み
    string connectionString = LoadConfiguration("connection_string");
    // 一部のリテラルが未定なパラメータとなっているSQL
    string query = "select count(*) from users where id = @id"
                 + " and password = @password";
    // 設定情報から接続オブジェクトを作成①
    using (SqlConnection conn = new SqlConnection(connectionString))
    {
        // RDBMSへの接続の開始②
        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③
        // コマンドオブジェクトにSQLを渡す④
        using (SqlCommand cmd = new SqlCommand(query, conn))
        {
            // SQLの雛型に対応するMACの指定
            cmd.Mac = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
            // 未定値のパラメータへ値を追加⑤
            cmd.Parameters.Add("@id", id);
            cmd.Parameters.Add("@password", password);
            // SQLの実行⑥
            int result = (int)cmd.ExecuteScalar();
            return result != 0;
        }
    }
}
```

図 5.4: 単純な拡張を行った場合のプログラムの記述例

そのため，データベース接続ライブラリによるMACの取り扱い方について工夫を行い，図5.3の単にコマンドオブジェクトにSQLの雛型とそれに対応するMACを渡すという構造を修正する必要がある．第4章で述べたMACに関する性質と既存の開発手法との互換性を考慮に入れると，MACおよびSQLの雛型の取り扱いに関して要求される性質は以下のようになる．

1. 開発者によって作成されたプログラムを受け取ったWebアプリケーション運用者は，Webアプリケーションを実行する前にアプリケーションの内部で利用される全てのSQLの雛型に対してMACをあらかじめ取得する必要がある．そのためWebアプリケーション運用者はアプリケーションの内部で利用されているSQLの雛型の一覧が容易に取得できなくてはならない．

2. 開発者は，SQLの雛型に対応するMACを実行時にアプリケーションの外部から読み込むようにプログラムを記述することになるが，このMACの読み込みのための処理はできる限り既存のデータベース接続ライブラリのインターフェイスを変更しない形で行えるようにする．つまり開発者は既存のデータベースを利用している場合とできる限り同じ方法でプログラムを書けるようにする．

3. 同様にデータベース接続ライブラリが内部で利用するSQLの雛型に対するMACを外部から読み込む処理も，容易に行えるように設計を行う．

こうした要求に沿う，データベース接続ライブラリの設計方法はいろいろと考えられるが我々の提案手法では以下のような設計を行った．

属性によるSQLの雛型の明示

まずアプリケーション中で利用されるSQLの雛型の一覧をアプリケーション運用者が容易に入手できるように，開発者はアプリケーション中で利用する全てのSQLの雛型を，SQLの雛型であることを表す特殊な情報を付加した定数文字列としてソースコード中で宣言するようにする．この際必要となるプログラム中の要素に対する情報の付加は，C#では「属性」を用いることで，Javaでは「アノテーション」を用いることで行うことができる．例えば文字列がSQLの雛型であることを表す属性として`[BuiltinSql]`という属性を導入したとすると，開発者はプログラムを図5.5のように書くことになる．

SQLの雛型に対するMACの取得

図5.5のように開発者が属性やアノテーションを利用してSQLの雛型として利用される文字列を明示しておくと，中間コードからリフレクションを用いて属性`[BuiltinSql]`が付加されている文字列を抽出することでアプリケーション中で利用されるSQLの雛型の一覧が容易に取得できることになる．そこでアプリケーション中で利用されるSQLの雛型に対するMACを取得するためのプログラムとして，

```
// 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み
    // 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 5.5: 適切な拡張を行った接続ライブラリを利用したプログラム

- 中間コードを受け取りリフレクションを利用してその中に含まれる SQL の雛型を抽出を行う
- RDBMS に SAFEMODE が無効な（つまり MAC の取得が許された）ユーザアカウントでログインし，中間コードから抽出した各SQL の雛型に対応する MAC と，データベース接続ライブラリが内部で利用する SQL の雛型に対応する MAC の取得を行う．
- SQL の雛型と MAC のペアのリストをファイルに出力する．

という動作をするプログラムを作成する．そして，アプリケーション運用者はこのプログラムを利用してアプリケーション中で利用される SQL の雛型に対する MAC の取得を行い，アプリケーションの運用を開始する前に SQL の雛型と MAC の組の列が格納されたファイルを作成しておくようにする．

MAC の読み込み

以上のように拡張を行うことでアプリケーション中で利用される SQL の雛型とその MAC の組のリストが作成できるようになるため，後は SQL 実行時に利用する MAC を読み込む仕組みを開発者が効率よく実装できるようにすればよい．この点については MAC の設定の手間を最小化するということや，接続ライブラリがバックグラウンドで行う処理も MAC を必要とする場合があることを配慮して，まず接続文字列とし

て接続オブジェクトに渡す設定情報を拡張して接続オブジェクトに MAC が格納されているファイルのパスを渡すようにし，接続オブジェクトが MAC が格納されているファイルを読み込み，SQL 実行時にコマンドオブジェクトが外部から渡された SQL の雛型に対応する MAC を接続オブジェクトが保持している MAC の情報から選択し，それを RDBMS の SQL 生成器に送信するように設計を行った（図 5.6）.

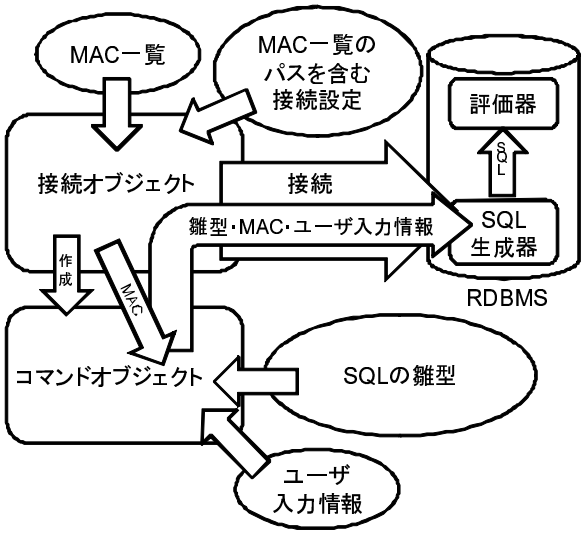

図 5.6: MAC の設定を考慮した構成における実行の流れ

5.2.3　まとめ

以上のように拡張を行ったデータベース接続ライブラリを用いると開発者は図 5.5 のようにプログラムを記述することで本提案手法が提供する SQLIA に対する耐性保証の枠組みの中で開発を行うことができるようになる．また図 5.5 のプログラムと図 5.1 のプログラムを比較すれば分かるように，本提案手法で行ったクライアントサイドの拡張は既存のデータベース接続のインターフェイスをできる限り変更することがないように工夫されている．そのため既存のデータベース接続ライブラリを用いて記述されているプログラムは，大きな書き換えをすることなく本提案手法が提供する SQLIA に対する耐性保証の枠組みの中で動作するプログラムへ書き換えることが可能となる．また，既存のデータベース接続ライブラリの利用方法を理解している開発者は新たなライブラリの使い方を学ぶことなく本提案手法の枠組みの中で開発が行えることになる．

Chapter 6　提案手法の導入方法と実装

6.1　アプリケーション開発への導入

本節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 が提供する安全な SQL 構築機能を用いるように開発者は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 5 章で述べた拡張されたデータベース接続ライブラリを通じて RDBMS が提供する SQL 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 が提供する安全な SQL 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 に対する耐性を持つことになる．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 SQL 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．

そこでプログラムの動作を検証する際に，4.3 で述べた特殊権限 SAFEMODE が有効な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上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 構築処理を実装してその処理によって生成された SQL を RDBMS に実行させようとしている場合には RDBMS によって SQL 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 SQL 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 SQL 構築機能を適切に利用していればプログラムは SQLIA に対して耐性を持つことになるため，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の SQLIA に対する耐性を保証することができることになる．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 が有効になっていることを確認した上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 SQL 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 SQL の実行を拒絶するため，開発者が独自に作成した SQL 構築処理によって SQLIA に対する脆弱性が生じることはありえず，アプリケーションが SQLIA に対して耐性を持つことが保証できる．

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 6.1 である．この図に表れているように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

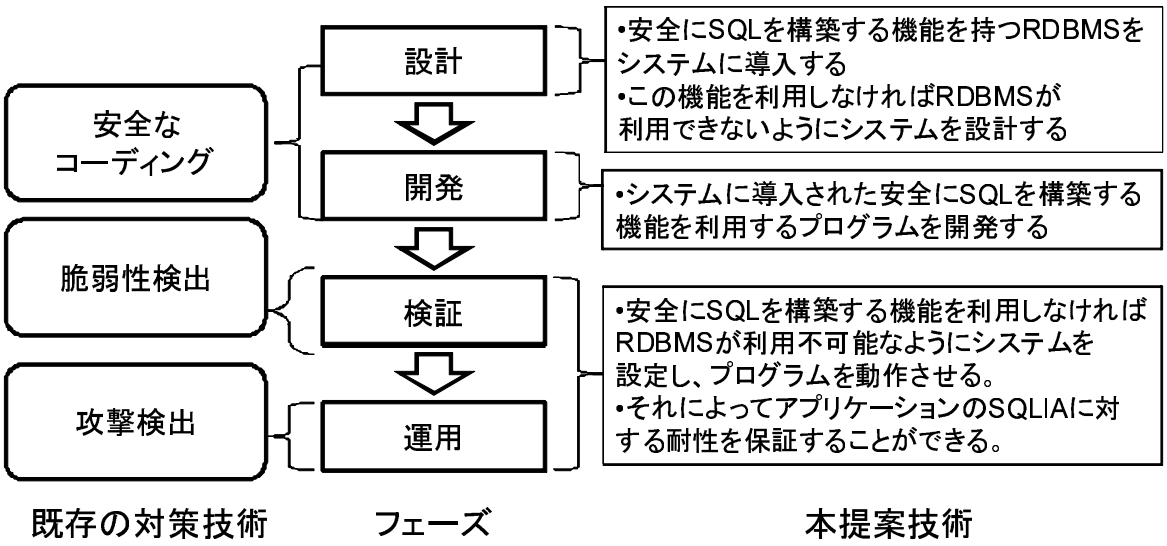

図 6.1: アプリケーション開発と対策技術の関係

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．

6.2　実装

本提案手法の実装として，第 4 章で述べた拡張をオープンソースのリレーショナルデータベース管理システム PostgreSQL[31] に対して施したものを作成した．また第 5 章で述べた拡張を施した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[32] を拡張したデータベース接続ライブラリを作成した．なお実装の内部で利用する MAC の計算アルゴリズムには HMAC-SHA1[33] を採用している．

PostgreSQL に対する拡張は主に外部との通信や SQL の構文解析などを担当するフロントエンド部分に対して行われており，SQL の実行やデータベースに格納されたデータの読み書きなどを担当するバックエンド部分に対する拡張はほとんど行われていないため PostgreSQL が内部で利用するデータフォーマットなどの互換性は保持されている．そのため PostgreSQL で運用している既存のデータベースが存在する場合にも，PostgreSQL をバージョンアップするのと同等の手順で導入することができる．また拡張が行われたフロントエンド部分に関しても，4.3 で述べたように SAFEMODE が無効なユーザアカウントで RDBMS にログインしている場合には既存の RDBMS で利用できる全ての機能が同じように利用できるように設計されているため，拡張された PostgreSQL は既存の PostgreSQL と全く同じように運用管理を行うことができる．また Npgsql を拡張したデータベース接続ライブラリも，接続文字列として接続オブジェクトに渡される設定情報を通じて，本提案手法向けのデータベース接続ライブラリとしての動作（図 5.6）を行うか既存のデータベース接続ライブラリとしての動作

（図 5.2）を行うかを切り替えられるように実装されているため，既存の Npgsql として利用することも可能である．

なお，これらの実装はオープンソースとして BSD ライセンスの下で公開を行っている [34]．また第三者が実際に実装を入手して本提案手法の有効性を検証できるように，[34] ではソースコードだけでなくインストール手順や基本的な利用方法をまとめた文書の公開も行っている．

Chapter 7　評価

7.1　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 を構築を行い RDBMS 上で実行するため実行速度の劣化が発生する．ただしその実行速度劣化はアプリケーション全体の大きな性能劣化を引き起こすほど深刻なものではないことを示すために，6.2 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実行速度の簡単な評価を行った．

本提案手法によって性能劣化が起こるのは RDBMS のフロントエンド部分の処理であるため，その部分の影響が実験結果に最大限あらわれるようにバックエンドの処理が軽量な SQL を選び，その実行時間を図ることによって実行速度の実験を行った．具体的にはあらかじめデータベース上に整数型の列を 1 つだけ持つテーブルを作成しそのテーブルに 1 行だけデータを格納しておき，クライアントが RDBMS に接続してから SELECT 文でデータを取得し RDBMS との接続を切断するまでにかかる時間の評価を行った．その結果が表 7.1 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．

表 7.1 が示すように本提案手法を導入することによる実行時間の増加は 30%程度で済んでおり，Web アプリケーションの実行を阻害するような著しい実行速度の低下は生じないことがわかる．さらに実際の Web アプリケーションにおいて必要となる処理は，RDBMS におけるバックエンドの処理・データを RDBMS からプログラムに送信する処理・Web ページを構築する処理・セッションの管理処理などの非常に多岐に渡るため，RDBMS のフロントエンド部における処理は Web アプリケーションで行われる処理の一部分に過ぎない．そのため，RDBMS のフロントエンド部の処理が Web アプリケーションのボトルネックとなっている場合を除き，本提案手法が Web アプリケーションの実行速度に与える影響は軽微である．

7.2　SQL 生成機能の妥当性

本提案手法では実行時の SQL 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・条件の繰り返し・ソート順序の制御の 4 つの処理以外は行われな

表 7.1: 実行速度の比較

	提案手法の利用なし	提案手法の利用あり
実行時間	3.6ms	4.7ms

いことを保証することでSQLIAに対するシステムの耐性を保証する．そのためSQL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・条件の繰り返し・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 [18, 19] で性能評価に利用されているアプリケーション [35] 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7.2がその結果である．Totalが各アプリケーションが利用するSQLの種類の総数，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の個数，Repeatが条件の繰り返しをともなうSQL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の個数である．ただしOptionalとRepeatの両方に含まれるSQLが存在するためBind，Optional，Repeat，Order，Otherの総和はTotalにはならない．表7.2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・条件の繰り返し・ソート順序の制御以外の操作がSQL構築時に必要となるものは存在しなかった．そのためここで挙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，この結果だけを持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 [18, 19] 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

表 7.2: アプリケーションで利用されるSQLの性質

	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

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・条件の繰り返し・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という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・条件の繰り返し・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の構築を行い通常の方法でRDBMSにSQL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が有効なアカウントでRDBMSに接続していることを慎重に確認した上で，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が無効なアカウントで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7.2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の極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，本提案手法が全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過度に懸念する必要はない．

Chapter 8　結論

本論文では，安全なコーディング方法をあらかじめ用意しておき，その安全なコーディング方法が利用することを開発者に対して技術的に強制付けることでアプリケーションの脆弱性を防止するという考え方に基づいたSQLインジェクション攻撃対策技術の提案を行った．この手法は，SQLの構築処理とその構築処理で利用されるSQLの雛型のみをプログラムの外部に出し，プログラムのその他の部分についてはブラックボックスとして扱うため既存の脆弱性・攻撃検出技術の一部が用いる高度なプログラム解析技術が不要であり，脆弱性を防止するための仕組みを非常に単純にすることが可能である．また既存の対策技術ではプログラムをブラックボックスとして扱う場合，アプリケーション運用者が攻撃検出ルールの定義などの複雑な設定作業を行わなければならないが，本提案手法では運用者によるそうした複雑な設定作業は必要ない．そのため本提案手法は既存のSQLIA対策技術に比べより容易かつ確実に脆弱性が防止できる技術となっている．

また本研究では，提案技術を実際に設計・実装するにあたって必要となるRDBMSおよびデータベース接続ライブラリの拡張について技術課題の洗い出しとその解決方法についての分析を行い，拡張方法を設計し，実際にPostgreSQLに対して拡張を施すことで実装を行った．さらに，本提案手法を導入することによって生じる制約が，実際にWebアプリケーションを開発する際の妨げにならないことを実験・評価を通じて確かめ，本提案手法が現実のWebアプリケーション開発に問題なく導入できることを確認した．最後に，本論文で述べた対策技術の更なる発展のために本章の残りの部分で提案手法の限界や展望について述べて結びとする．

8.1　本提案手法による耐性保証の限界

4.4で述べたように，本提案手法をシステムに導入することによってアプリケーション運用者は「SAFEMODEが無効なユーザがRDBMS上で行う操作は，アプリケーション上でユーザが行う操作の影響を全く受けない」という容易に達成可能な条件が成立するように運用環境を設定することによってアプリケーションのSQLインジェクション攻撃に対する耐性が保証できる．そしてこの耐性保証を支えるのが

- 静的なSQLはSQLインジェクション攻撃による改竄を受けていない．
- 静的なSQLの雛型から安全なSQL生成器を利用して作成されるSQLはSQLインジェクション攻撃による改竄を受けていない．

という2つの仮定であり，本論文で提案しているSQLIAに対する耐性保証の枠組みの有効性はこの2つの仮定が成立するか否かに依存することになる．しかしながら，実

のところ「SQL インジェクション攻撃による SQL の改竄」が一体何を指すのかについての厳密な定義が存在しないため，自明に思える「静的な SQL は SQL インジェクション攻撃による改竄を受けていない」という前者の仮定でさえ，そもそもこの仮定が何を定義しているのかがはっきりとせず，成立するのか成立しないのかについて厳密には何も言うことができない．

もしも「SQL インジェクション攻撃」をプログラム内部で行われる単純な文字列処理による SQL の構築処理の不備をついて SQL の改竄を行う攻撃であると定義するのであれば,「SQL インジェクション攻撃」によって静的な SQL に対して SQL の改竄が起こることも，静的な SQL の雛型から 4.1 で設計した安全な SQL 構築器によって生成された SQL に対して改竄が起こることもない．そのため上述の 2 つの仮定が成立し,「SQL インジェクション攻撃」による SQL の改竄が発生しないことを本提案手法の枠組みによって保証することができる．そして [6] でまとめられているような典型的な SQLIA というのはいずれも文字列処理による SQL の構築処理の不備をつく攻撃であるため，こうした典型的な SQLIA に対する安全性は本提案手法の枠組みで保証することができる．そのため，現実に問題になるような SQLIA の多くに対しては本提案手法を用いることでアプリケーションの安全性が保証できる．

一方で，上記の 2 つの仮定に合致しない攻撃に関しては本提案手法の枠組みの中では安全性を保証することができないため，そうした攻撃も「SQL インジェクション攻撃」として定義するのであれば，本提案手法による安全性保証に穴があることになり「SQL インジェクション攻撃」に対する脆弱性が放置される危険性がある．例えば図 8.1 のようなユーザ認証のプログラムが存在したとする．このプログラムでは静的に宣言された SQL の雛型 `normalQuery` と `backdoorQuery` から作成された SQL の実行のみが行われるため，本提案手法の枠組みの中では SQLIA に対する脆弱性は存在しないとみなされ正常に実行することができる．しかしながら，図 8.1 のコードには第 1 章で例示した図 1.3 のユーザ認証用コードと同様に，攻撃者がパスワードとして “`’ or ’a’ = ’a`” を入力することで内部で実行される SQL の条件式を “`id = ’...’ and password = ’...’ or ’a’ = ’a’`” という形に改竄し認証を迂回することができてしまう「脆弱性」が存在する．そうした意味では，図 8.1 のプログラムにも図 1.3 と同様に SQLIA に対する脆弱性が存在していると考えることも可能であり，その場合本提案手法による SQLIA に対する安全性保証には穴があるということになる．もちろん，図 8.1 のようなコードが実際の開発時に作成されることはほとんど考えられないが，この例は

- 開発者が故意に危険な SQL をプログラム内に組み込んだ場合に生じる脆弱性
- どの SQL（の雛型）を利用するかを選択するロジックに不備がある場合に生じる脆弱性

の少なくとも 2 つについては本提案手法は有効に機能しないことを示している．また本提案手法では，SQL の雛型から SQL を作成する際には「値のバインド」・「条件の有無」・「条件の繰り返し」・「ソート順序の制御」の 4 つの処理以外は行えないようになっているが，攻撃者がこの 4 つの処理のみを用いて開発者の意図せぬ SQL を作成し

てしまい脆弱性が発生する可能性も否定できない．そして，こうした脆弱性もSQLIAに対する脆弱性であると定義するのであれば，本提案手法を導入することで保証できることはあくまで，上述の2つの仮定を満たすSQLIAの部分集合に対する脆弱性がアプリケーションに存在せず，アプリケーションはSQLIAに対して一定レベルの耐性があるということだけとなる．

```
[BuiltinSql]
private const string normalQuery = "select count(*) from users "
                                 + "where id = @id and password = @password";
[BuiltinSql]
private const string backdoorQuery = normalQuery + " or 'a' = 'a'";

public bool Authenticate(string id, string password)
{
    string query;
    if(password = "' or 'a' = 'a")
    {
        query = backdoorQuery
    }
    else
    {
        query = normalQuery;
    }
    using (SqlConnection conn = new SqlConnection(connectionString))
    using (SqlCommand cmd = new SqlCommand(query, conn))
    {
        conn.Open();
        cmd.Parameters.Add("@id", id);
        cmd.Parameters.Add("@password", password);
        int result = (int)cmd.ExecuteScalar();
        return result != 0;
    }
}
```

図 8.1: バックドア付きの認証プログラム

こうした問題は，SQLの構築処理とその構築処理で利用されるSQLの雛型のみをプログラムの外部に出し，その他の部分についてはブラックボックスとして扱う本提案手法の限界を超えているため，こうした問題を防止するには本提案手法とは異なる考え方に基づく技術や手法を導入する必要がある．ただし第2章で挙げた関連研究・技術は，既存のSQLIAを招きにくいコーディング手法・研究は本提案手法で行われるコーディング方法の拡張と衝突するため併用が難しく，また既存の脆弱性検出・攻撃検出技術の多くは上述したようなSQLIAと定義すべきがどうか曖昧な攻撃を対象としていないため本提案手法と併用しても脆弱性防止効果はあまり期待できない[1]．そのため，こうした本提案手法では扱いきれない問題を解決するには既存のSQLインジェクション攻撃対策技術とは異なる技術・研究との併用が必要になる．特に，上で述べたような開発者の悪意による脆弱性や利用するSQLの選択ロジックに不備があることによって生じる脆弱性などはプログラムだけをみてもそれが脆弱性なのか仕様通りの振る舞いなのか区別が困難であるため，こうした脆弱性を防止するためにはプログラ

[1]ただしWAFやIDSを利用したシグネチャベースの攻撃検出技術はシステムの脆弱性を探索している攻撃者を早期に発見するのには有益である

ムとは別に仕様を記述しておき，プログラムがその仕様を満たしているかどうかを検証する必要がある．そのためこうした脆弱性を技術的に防止するには形式仕様記述とそれを利用した形式的検証技術が必要となると考えられる．

8.2　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脆弱性が存在するかしないか分からないプログラムはすべて危険なものみなすことでアプリケーションに脆弱性が存在しないことを保証する．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持ってしても全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関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非常に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全くの無駄である．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．

8.3　展望

本提案手法の大きな欠点は，8.2 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，これを人手で行っていると大きなコストの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上で重要であるが，これに関しては文献 [18] で利用されているプログラムの各部分でどのような SQL が生成されるかを分析する技術などのプログラムの静的解析技術を用いることで，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 [18] 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良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 SQLIA に注目していたが，本提案手法の根本にある安全に 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言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．

さらに，開発者に対して脆弱性を招くようなコーディング方法[2]を許可しないことによって強制的に脆弱性がないプログラムを記述させるという本提案手法の考え方を他の典型的な Web アプリケーションの脆弱性 [3] に対して適用した技術を研究・開発することも可能であると考えられる．そして，そうした技術を 1 つのアプリケーションフレームワークとしてパッケージ化し典型的な脆弱性をもつプログラムを開発者が作成してしまわないように最大限考慮された Web アプリケーションフレームワークを提案することで，基本的に開発効率のみが重視されセキュリティが考慮されていないことの多い Web アプリケーションフレームワークの現状に一石を投じることができれば非常に面白い．

[2]本提案手法の場合は，プログラム内部で独自の文字列処理に基づいてユーザ入力値から SQL を生成するプログラムを記述すること

謝辞

本研究にあたりご指導を頂き，また学会参加など多数の各種活動の機会を与えて頂いた東京大学情報理工学系研究科 松浦幹太准教授 に心から感謝申し上げます．そして，学生の研究活動が円滑に行えるように尽力してくださっている秘書の仲野さんと技術職員の細井琢朗さんにも深く感謝致します．

また修士課程の 2 年間を通じて，松浦研究室の先輩・後輩としてお世話になった楊鵬さん，Jacob Schuldt さん，Vadim Zendejas さん，Phan Thi Lan Anh さん，北田亘さん，松田隆宏さん，中井泰雅君，施屹君には研究だけでなく日常の会話や議論を通じて，暗号学などの自分の研究とは異なる視点からの考え方を学ばせていただき，ソフトウェアエンジニアとしての視点に偏り狭くなりがちな自分の研究者としての視点を広げていただきましたことに特に感謝申し上げます．また定期的に松浦研究室のミーティングに参加して下さった NHK 放送技術研究所の小川一人さん，警察庁情報技術解析課の岡田智明さん，産業技術総合研究所の田沼均さんには研究内容に関して様々な指摘や議論して下さったことに感謝申し上げます．

参考文献

[1] Mike Andrews. The state of web security. *IEEE Security and Privacy*, Vol. 4, No. 4, pp. 14–15, JULY/AUGUST 2006.

[2] `http://www.securityfocus.com/`.

[3] OWASP Top Ten Project. OWASP Top 10 2007. `http://www.owasp.org/index.php/Top_10_2007`.

[4] 情報処理推進機構. 情報セキュリティ白書, 2008.

[5] SPI Labs. SQL injection are your web applications vulnerable? In *SPI Dynamics Whitepaper*, 2006.

[6] W. G. Halfond, J. Viegas, and A. Orso. A classification of sql-injection attacks and countermeasures. In *In Proc. of the Intern. Symposium on Secure Software Engineering*, 3 2006.

[7] Ke Wei, M. Muthuprasanna, and Suraj Kothari. Preventing sql injection attacks in stored procedures. In *ASWEC '06: Proceedings of the Australian Software Engineering Conference*, pp. 191–198, Washington, DC, USA, 2006. IEEE Computer Society.

[8] JDBC. `http://java.sun.com/javase/technologies/database/`.

[9] ADO.NET. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/adonetanchor.asp`.

[10] Russell A. McClure and Ingolf H. Krüger. Sql dom: compile time checking of dynamic sql statements. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pp. 88–96, New York, NY, USA, 2005. ACM Press.

[11] 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.

[12] Stephen W. Boyd and Angelos D. Keromytis. Sqlrand: Preventing sql injection attacks. In *In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pp. 292–302, 2004.

[13] Gregory Buehrer, Bruce W. Weide, and Paolo 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*, pp. 106–113, New York, NY, USA, 2005. ACM Press.

[14] SNORT. Snort.org. `http://www.snort.org/`.

[15] Nilesh Burghate K. K. Mookhey. Detection of sql injection and cross-site scripting attacks. `http://www.securityfocus.com/infocus/1768`.

[16] Elisa Bertino, Ashish Kamra, and James P. Early. Profiling database application to detect sql injection attacks. In *IPCCC*, pp. 449–458, 2007.

[17] Fredrik Valeur, Darren Mutz, and Giovanni Vigna. A learning-based approach to the detection of sql attacks. In *In DIMVA (2005*, pp. 123–140, 2005.

[18] William G. J. Halfond and Alessandro Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 174–183, New York, NY, USA, 2005. ACM.

[19] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. Candid: preventing sql injection attacks using dynamic candidate evaluations. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pp. 12–24, New York, NY, USA, 2007. ACM.

[20] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 372–382, New York, NY, USA, 2006. ACM.

[21] National Institute of Standards and Technology. Web application vulnerability scanners. `https://samate.nist.gov/index.php/Web_Application_Vulnerability_Scanners`.

[22] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web application security assessment by fault injection and behavior monitoring. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pp. 148–159, New York, NY, USA, 2003. ACM.

[23] Gunther Birznieks. Cgi/perl taint mode faq. `http://gunther.web66.com/FAQS/taintmode.html`.

[24] Perl. `http://www.perl.org/`.

[25] Ruby. A programmer's best friend. http://www.ruby-lang.org/.

[26] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pp. 365–383, New York, NY, USA, 2005. ACM.

[27] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pp. 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[28] C. J. Date. *Database in Depth: Relational Theory for Practitioners.* O'Reilly Media, Inc., 2005.

[29] Claude Rubinson. Nulls, three-valued logic, and ambiguity in sql: critiquing date's critique. *SIGMOD Rec.*, Vol. 36, No. 4, pp. 13–17, 2007.

[30] Python Database API Specification v2.0. http://www.python.org/dev/peps/pep-0249/.

[31] PostgreSQL. The world's most advanced open source database. http://www.postgresql.org/.

[32] Npgsql. .Net Data Provider for Postgresql. http://npgsql.projects.postgresql.org/.

[33] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. pp. 1–15. Springer-Verlag, 1996.

[34] http://www.logos.ic.i.u-tokyo.ac.jp/~yunabe/svsql/.

[35] GotoCode. Open Source Web Applications with Source Code in ASP, JSP, PHP, Perl, ColdFusion, ASP.NET / C#. http://www.gotocode.com.

発表文献

和文論文誌

i 渡邉悠, 松浦幹太. “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”, 「社会を活性化するコンピュータセキュリティ技術」特集 情報処理学会論文誌, (投稿中).

査読無し国内会議投稿論文

ii 渡邉悠, 松浦幹太. “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”, 2009 年 暗号と情報セキュリティシンポジウム (SCIS 2009) 予稿集 CDROM, 1E1-2. 滋賀, 1 月・2009 年.

iii 渡邉悠, 松浦幹太. “インジェクション系脆弱性を持つコードの記述が不可能なフレームワーク”, 2008 年 暗号と情報セキュリティシンポジウム (SCIS 2008) 予稿集 CDROM, 1C1-2. 宮崎, 1 月・2008 年.

iv 渡邉悠, 松浦幹太. “SQL の条件節が動的に構成されることを考慮したデータベース接続 API の設計”, Computer Security Symposium 2007 (CSS 2007). 奈良, 10-11 月・2007 年.

デモンストレーション

v 渡邉悠, 松浦幹太. “SQL インジェクション攻撃を防止するためのプログラミング方法とデータベース拡張”, Computer Security Symposium 2007 (CSS 2007). 沖縄, 10 月・2008 年.