All rights are reserved and copyright of this manuscript belongs to the authors. This manuscript has been published without reviewing and editing as received from the authors: posting the manuscript to SCIS 2008 does not prevent future submissions to any journals or conferences with proceedings.

SCIS 2008 The 2008 Symposium on Cryptography and Information Security
Miyazaki, Japan, Jan. 22-25, 2008
The Institute of Electronics, Information and Communication Engineers

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

Framework for making it impossible to write codes vulnerable for injection attacks

渡邉　悠 *　　　松浦　幹太 *
Yu Watanabe　　　Kanta Matsuura

あらまし Web アプリケーションの利用が拡大している昨今、Web アプリケーションの脆弱性の大きな割合を占めるインジェクション系の脆弱性に対する対策技術は非常に重要である。しかし既存の研究は、攻撃もしくは脆弱性の検知を行いシステムのセキュリティを保とうとするものが多く、それゆえ誤検知や検知漏れが発生するという問題抱えている。本論文では、攻撃や脆弱性を検出するのではなく、プログラマに対して安全なコードを記述することを強制することによってシステムの安全性を保つアーキテクチャの考案した。また、この考えを取り入れた SQL インジェクション対策用フレームワークを設計し、実装を行った。

キーワード 脆弱性対策, Web セキュリティ, インジェクション攻撃, SQL インジェクション攻撃, クロスサイトスクリプティング

1 はじめに

1.1 Web セキュリティと脆弱性の現状

近年インターネットが普及・高速化し Web を通じて提供されるサービスが増加するのにともない、Web アプリケーションの脆弱性が大きな問題となっている。そして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をあわせて、Web アプリケーションの脆弱性の半数を超えるといわれるインジェクション系の脆弱性の問題を解決することが特に重要となっている [2]。

1.2 インジェクション系の脆弱性

インジェクション攻撃の具体例として、SQL インジェクションに対する脆弱性を利用して攻撃者に認証を回避されるプログラムの例 [7] を示す。図 1 はユーザの情報が格納されたテーブル Users から、ID とパスワードがともにユーザが入力した値と一致する行の数をカウントし、行数が 0 だった場合は認証を拒否し、そうでない場合は受理するというプログラムの疑似コードである。このプログラムに対してユーザが ID に「alice」、パスワードに「foo」を入力すると生成される SQL の条件部分は「id = 'alice' and password = 'foo'」となり、プログラマが想定した通りの SQL が実行され認証が正しく行われる。しかしユーザが入力したパスワードが「' or 'a' = 'a」などであると生成される SQL の条件部分は「id = 'alice' and password = '' or 'a' = 'a'」という常に成立する論理式になってしまう。そのため SQL を実行した結果得られる値は、Users テーブルの行数となってしまい 0 ではなくなる。その結果、このユーザは alice のパスワードを知らないにも関わらず alice として認証されてしまうことになる。

```
sql = ”select count( * ) from users”
   + ” where id = ‘” + request[”id”]
   + ”’ and password = ‘” + request[”password”] + ”’”;
count = execute(sql);
if ( count == 0) {
    reject
} else {
    accept
}
```

図 1: 認証をバイパスされるプログラム

2 既存技術・研究

インジェクション系の脆弱性は古くから有名なものであり、対策技術・研究もさまざまなものが存在している。これらの対策技術はそれぞれの基本にある考え方から安全なプログラムを記述する技術、攻撃の検出、脆弱なプログラムの検出の３つに分類することができる。

2.1 安全なプログラムを記述する技術

脆弱性が生じるのは、ユーザの入力がプログラム中で適切に処理されていないことが原因である。そのためユーザの入力を適切に処理することがもっとも基本的かつ根本的な対策である。例えば 1.2 の例では、ユーザが

* 東京大学生産技術研究所, 153-8505 東京都目黒区駒場 4-6-1, Institute of Industrial Science, the University of Tokyo, 4-6-1 Komaba, Meguro-ku, Tokyo 153-8505, Japan({yunabe,kanta}@iis.u-tokyo.ac.jp)

パスワードとして入力したはずのシングルクオートが文字列の終端として評価されてしまっていることが認証をバイパスされてしまう原因である。そのため、この場合はユーザの入力に含まれているシングルクオートを SQL の文字列リテラル中でシングルクオートとして評価される文字列[1]に変換してから、図 1 のように文字列の接続で SQL を構築すれば、例で示したような攻撃を回避することができる。こうした文字列処理は一般にエスケープ処理と呼ばれる。しかし同じ言語の中でも、例えば文字列と数字のように、表現したい対象が異なれば行うべきエスケープ処理は異なってくる。またエスケープ処理を行うためには、プログラムのメインロジックとは全く関係のない処理を埋め込む必要があるため、開発効率の低下やエスケープ処理を忘れたことによりシステムに脆弱性が紛れ込むといった事態を招く恐れがある。そのためより効率よく安全にプログラムが作成できる技術が不可欠となる。エスケープ処理を用いずに安全に SQL を構築するための技術に、ADO.NET[1] や JDBC[6] などのデータベース接続ライブラリには標準的に取り入れられている「バインディングメカニズム」というものがある。これは SQL の動的に決定される値の部分に開発時にはスペースホルダと呼ばれる適当な文字列を記述しておき、実行時にスペースホルダに対して値を紐付けることで SQL を作成するという技術である。例えばバインディングメカニズムを利用した、認証用のプログラムは図 2 のようになる。

```
cmd = new SqlCommand();
cmd.sql = "select count( * ) from users"
   + " where id = @id and password = @password";
cmd.parameters.bind("@id", request["id"]);
cmd.parameters.bind("@password", request["password"]);
count = cmd.execute();
if ( count == 0) {
    reject
} else {
    accept
}
```

図 2: バインディングメカニズムを利用した安全な認証プログラム

2.2 攻撃の検出

インジェクション攻撃に対するもう一つの対策は攻撃の発生を検知することであり、検知対象の違いによってさらに攻撃自体を検出する技術と攻撃を受けたことによって生じた異常を検出する技術に分けることが可能である。前者の方法は人手で定義した攻撃のシグネチャを利用して IDS[8] で攻撃を検出する手法が、後者では正常状態を事前に機械学習しておきシステムの状態が正常状態から逸脱することを検出することで攻撃を検知する手法が利用されている [11][4][9]。

[1] シングルクオートを２つ並べたもの

2.3 脆弱なプログラムの検出

インジェクション系攻撃対策のもうひとつの考え方は、プログラムに含まれる脆弱性を検出しようとする手法であり、これらの手法はホワイトボックス的な手法とブラックボックス的な手法に大別できる。ホワイトボックス的な手法としては、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 [3] が非常に有名である。ブラックボックス的な脆弱性検出技術としては、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、インジェクション系の脆弱性を発見するためのツールも数多く存在している。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。

3 動機

インジェクション系の問題の対策技術は以下のような性質を持っていることが望ましいと考えられる。

1. システムのインジェクション系の攻撃に対するセキュリティを確保することはもちろんのこと、インジェクション系の問題によってシステムにバグが埋め込まれていないことを保証することができる。
2. これをプログラマ任せにせずに行うことができる。

この視点からみるとプログラマが利用する 2.1 の安全なプログラムを記述するための技術は 2 番目の点が欠けている。また 2.2 の攻撃検出によってシステムの安全性を保つ技術は、対症療法的な手法であり脆弱性を根本的に解決するものではないため、インジェクション系の問題によって生じるバグに対しては全くの無力である。2.3 の脆弱性検出技術は 1, 2 の両方の点において有効であるが、検出した問題点を修正するには 2.1 の技術を利用する必要がある。

また脆弱性を検出するにせよ攻撃を検出するにせよ、検出に人が作ったルールを利用する場合にせよ機械学習によって得られたルールを利用する場合にせよ、何かを検出する技術には常に誤検知の問題と検知漏れの問題がつきまとう。実際こうした技術研究の多くが、いかに誤検知率・検出漏れ率を小さくするかに重点を置いているが、検出技術を持ってシステムの安全性・正しさを 100%保証することはどれほど技術が進んでも不可能である。

そこで本研究では単一の技術を持って

1. システムの安全性および正しさを
2. プログラマ任せにすることなく
3. 本質的に不確実さをともなう検出技術を用いることなく

保証することのできるアーキテクチャの提案を行う。ただしモデルチェッキング [5] に代表されるように、プログラマによって自由に作成されたプログラムに対してその正しさを検証し証明することは非常に難しく、少なくとも現在の技術では、現実のプログラムを検証することは

不可能である。そのため本研究ではプログラマに対して安全にプログラムを記述するためのツールの利用を強制することでシステムの正しさを保証するアーキテクチャを提案する。以下ではまず 4 でインジェクション系の問題を解決するアーキテクチャの提案を行い、5 でこのアーキテクチャを利用した SQL インジェクション対策フレームワークの設計と実装について述べ、最後に 6 で本手法がもたらすメリットと、本手法が抱える問題点について述べる。

4 アーキテクチャ

4.1 プログラムを出力するプログラムの構造に対する考察

Web アプリケーションでは Java, C#, PHP などで記述されたプログラムによって SQL や HTML などのプログラムが生成され、それがデータベースやブラウザによって評価・実行される。以下では混乱を避けるために、SQL, HTML など Web サーバ上のプログラムで動的に作成されるプログラムを「プログラム」と呼び、Java, C#, PHP などで記述された、「プログラム」を作成するためのプログラムを「メタプログラム」と呼ぶことにする。この用語を用いると、プログラムを作成するプログラムの開発と実行の基本構造は以下のように見ることができる。

1. プログラムを生成するメタプログラムがプログラマによって開発時に作成され
2. 運用時に、メタプログラムが実行されプログラムが生成される。

そしてこれは視点を変えると

1. 生成されるプログラムの集合が開発時にメタプログラムを利用して定義され
2. メタプログラム実行時にその中からプログラムがひとつ選ばれ、出力される

というように考えることもできる。例えば 1.2 の例は

1. 図 1 の SQL を生成するメタプログラムがプログラマによって記述され
2. 実行時に、クライアントの入力から特定の ID とパスワードを持ったデータの個数を数える SQL が生成されデータベースで実行される

というものであったが、視点を変えると

1. 生成される SQL の集合は「“select count(*) from where id = ? and password = ?” の?に具体的な値を入れたもの集合」であると、開発時にプログラマが定義し
2. 実行時にクライアントからの入力を利用して、定義された集合の中から例えば「select count(*) from where id = 'alice' and password = 'foo'」という要素を選び出し、出力している

と見ることが可能である。そしてこうした視点から見るとインジェクション系の問題というのは、「集合の定義」と「要素の選択」に利用しているツールが不適切であるために生成されるプログラムの集合が正しく定義できず、意図しているものとは異なる集合が定義されてしまい、その結果開発時に想定していたのとは異なるプログラムが選択・生成されてしまう問題であると考えることができる。例えば 1.2 の例ではプログラマは図 1 のメタプログラムによって、where 節が「id = ? and password = ?」という形の SQL の集合を定義し、実行時にその中から適切な SQL が選択され実行されることを期待しているが、実際には図 1 のメタプログラムでは where 節の形が「id = ? and password = ?」とは異なる SQL が生成可能であり、それが認証をバイパスされるという脆弱性につながっている。また 2.1 で紹介したバインディングメカニズムを使ったメタプログラム (図 2) は、スペースホルダを利用して「select count(*) from users where id = @id and password = @password」と記述することで生成される SQL の集合をプログラマが定義し、実行時にスペースホルダに値をバインディングすることで集合から SQL を選択しているとみることができる。そしてバインディングメカニズムを利用した場合には「プログラムの集合の定義」と「集合からのプログラムの選択」が適切に行えるためインジェクション系の問題が起こらないと解釈することができる。

4.2 インジェクション系の問題の回避方法

このように視点を変えるとインジェクション系の問題というのは、プログラマが不適切なツールを利用して生成されるプログラム集合を定義しているために生じる問題であると考えることができる。そのため、インジェクション系の問題は

1. 「集合の定義」と「要素の選択」がプログラマの意図通りに行うことができるツールを用意し、
2. そのツールの利用を何らかの方法でプログラマに強制する

ことによって回避することができるといえる。

はじめに特定のツールの利用を強制するためのアーキテクチャについて考える。特定のツールの利用を強制するアーキテクチャを設計するにあたりまず「中継者」を作成する。中継者はプログラマによって作成されるメタプログラムとは独立したコンポーネントであり、プログラム生成者から「作成するプログラムの集合」と「集合からプログラムを選択するための情報」を受け取りプログラムを生成し、生成したプログラムをプログラム実行者に受け渡す役割と、プログラムの実行結果を実行者から作成者に対して中継する役割を担う。そして、プログラムを生成するコンポーネント（e.g. Web サーバ）と生成されたプログラムを実行するコンポーネント (e.g. データベース) の間に「中継者」を設置し、この中継者を通してしかプログラム生成側からプログラム実行側へプログラムを受け渡すことができないようにシステムを構築する。このようにシステムを構築すれば、プログラマは中継者が提供する適切なツールを利用しなければプログラムを作成することができなくなり、脆弱性やバグを

埋め込む恐れの多い手法を用いてプログラムを作成することが不可能になる。図 3 は Web サーバとデータベースの間に「中継者」を設置し、SQL インジェクション系の問題を回避するためのアーキテクチャの例である。

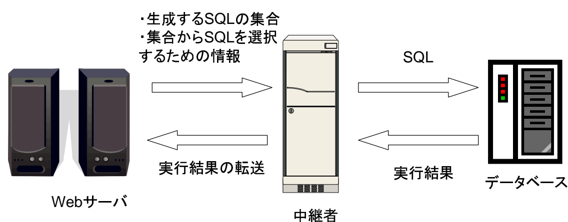

図 3: 基本構造

4.3　中継者に対する要求

4.2 で述べたようにメタプログラムとプログラム実行者の間に中継者を設置すれば、メタプログラムに対して中継者が提供しているプログラムを作成するためのツールの利用を強制することができる。そのためインジェクション系の問題を解決する上で残された課題は、いかにして中継者の提供する「集合の定義」と「要素の選択」のためのツールを設計するかである。この部分を具体的にどのように設計するかは対象とする言語（SQL, HTML, etc...）に依存するため、設計は言語ごとに行わなければならない。ただし対象とする言語が何であったとしても、中継者は以下の要求を満たすように設計されなければならない。

1. プログラマは作成したいプログラムの集合を意図した通りに定義できなくてはならない。
2. プログラマは作成したいプログラムの集合を不足なく、効率的に定義することができなくてはならない。
3. 集合の定義は静的に定義されたものでなくてはならない。
4. 実行時に行われる、要素の集合からの選択はインジェクション攻撃を受けることのないように、できる限りシンプルでなくてはならない。
5. プログラマは作成したいプログラムの集合を不必要に大きくすることなく、できるだけ精確に定義できなくてはならない。またプログラマは必要以上に大きな集合を定義することができてはならない。

以下で各性質に対して簡単な説明を行う。

要求 1　プログラマがプログラムの生成を行うメタプログラムを、意図したプログラムを正しく表現できない手段を利用して作成することがインジェクション系の問題の原因である。そのため、この性質はインジェクション系の問題を引き起こさないためのフレームワークとして自然に要求されるものである。

要求 2　ここで提案しているアーキテクチャではプログラマは中継者の提供しているツールを利用してプログラムを生成することを強制される。そのため、このツールはプログラマが実行時に生成したいと考えているプログラムの集合を不足なく、しかも効率的に記述することが可能でなくてはならない。さもなければ、このアーキテクチャは開発の足かせとなり場合によっては開発を不可能なものにしてしまう。

要求 3　要求 3, 5, 4 はこのツールが不適切な利用のされ方をして、インジェクション系の問題をシステムに埋め込まれるのを防ぐために必要となる性質の要求である。要求 3 であるが、もしもこの性質が満たされず生成されるプログラムの集合を動的に定義することができるとすると、今度はプログラムの集合の定義に対してインジェクションが行われることになってしまい中継者というコンポーネントをシステムに追加しシステムを複雑化させただけで、何の安全性の向上にもつながらないことになってしまう。

要求 4　上で述べたように、集合の定義は静的に行われるべきものであり何らかの手段を持って静的に定義されていることを確かめられれば集合の定義に対するインジェクション系の問題は回避することができる。しかし集合からプログラムを選択する部分は、外部からの入力に基づいて動的に行われるものであるため、この部分にインジェクション系の問題が埋め込まれる潜在的な危険性が存在する。そのため集合からプログラムを選択する部分はなるべくシンプルにし、外部からの入力が問題を引き起こす余地ができる限り小さくなるように設計しなくてはならない。

要求 5　上述したように、集合から要素を選択する部分には潜在的に問題が埋め込まれる余地があるため、静的に定義されたプログラムの集合に不必要なプログラムが含まれることは望ましくない。そのためプログラマが生成したいと考えているプログラムの集合を、集合を不必要に大きくすることなく定義できることが望ましい。また大きすぎる集合の定義が可能であるとプログラマがプログラムの集合として「すべてのプログラム」を指定し、実行時にその中からひとつプログラムを選ぶということが可能になる。こうなってしまうと、プログラムを選択する部分の複雑さが生成対象であるプログラムの複雑さと全く変わらなくなり、結果的にプログラムを選択する部分に対してインジェクション系の問題が埋め込まれることになる。これではやはりアーキテクチャが複雑化しただけで本質的には何の解決にもなっていないため、大きすぎる集合の定義は不可能でなければならない。

4.4　集合が静的に定義されていることを保証する枠組み

要求 1, 2, 4, 5 を満たすためにフレームワークをどのように設計するかはフレームワークが出力対象とするプログラミング言語に依存するが、要求 3 を実現する方法のみは生成対象である言語とは独立した話であるため具体的な実現方法について簡単な考察を行う。

まずもっとも単純な実現方法は、中継者にプログラムの集合を登録するためのインターフェイスを設けておき、プログラマがそれを利用してメタプログラムが利用する

プログラムの集合を事前登録し、実行時に、事前登録されたプログラムの集合とメタプログラムから送られたプログラムを選択するための情報とを利用してプログラムを作成するという方法である（図 4）。ただし集合を事前登録するための機能は実行時には利用できてはならないため、この方法ではメタプログラムをアップデートするたびに中継者の設定を変更し集合の登録の機能を利用可能にしたり利用不可能にしたりする必要があり、管理の手間の増加が問題になる。

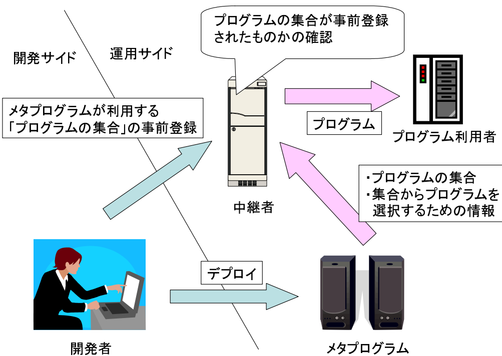

図 4: 集合を事前登録する方法

この設定の煩雑さは、集合を登録するための機能を中継者から分離し運用環境からアクセスできない位置に設置することで緩和することができる。集合を登録するための機能を中継者から分離するために、まず「署名者」というコンポーネンをアーキテクチャに追加する。そして、この署名者を運用環境からは物理的にアクセスできない位置に設置し、署名者と中継者それぞれに秘密鍵と公開鍵（もしくは両方に共通鍵）を配布する。このようにすれば、プログラマはメタプログラムが利用するプログラムの集合の定義に対して署名（もしくは MAC）を署名者に付与してもらいそれを運用環境にデプロイし、実行時にメタプログラムが中継者に対して集合の定義と署名（もしくは MAC）を送信し中継者がそれを検証することで、集合の定義が静的に作成されたものであることが保証できる（図 5）。

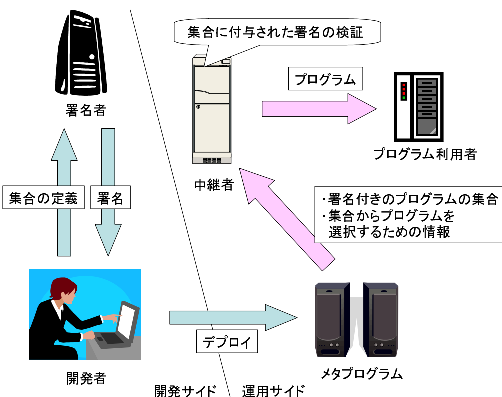

図 5: 中継者と集合の登録の分離

署名の検証コストをなるべく小さくすることを考慮した実際のプロトコルは次のようになる。

署名の付与 プログラマは署名者に集合の定義を送信し、署名を取得する。また衝突が起こらない程度に十分広い空間からユニークな ID を適当に選択し、(ID, 集合の定義, 署名) というタプルをファイルに保存する。そしてこれをメタプログラムに配布する。

署名の検証・プログラムの作成 メタプログラムはプログラマから受け取ったファイルを読み、中継者に対して集合の ID と集合からプログラムを選択するための情報を送る。受け取った ID に対応する集合の定義を中継者が知らない場合、中継者はその旨をメタプログラムに通知しメタプログラムから集合の定義とそれに対する署名を受け取り、署名の検証を行い ID と集合のペアを保存する。そして中継者は、メタプログラムから送られてきた ID に対応する集合の定義と集合からプログラムを選択するための情報を利用してプログラムを作成しプログラム実行者にプログラムを渡す。

5 SQL インジェクション対策フレームワークの設計

残された問題は、対象とする言語に依存する 4.3 の要求 1, 2, 4, 5 を満たすように、いかにしてプログラムの集合の定義と集合からのプログラムの選択のためのツールを設計するかである。特に要求 1, 2 で求められる性質と要求 4, 5 で求められる性質は相反するものであるため、このトレードオフをいかにして解決するかが問題となる。以下では SQL を対象とする、プログラムの集合の定義とプログラムの選択を行うためのフレームワークの設計について述べる。

5.1 動的に決定される要素の限定

SQL にはさまざまな機能が用意されているが、本提案手法ではその中でも Web アプリケーションで頻繁に利用される CRUD 処理と呼ばれる「作成（Create）」「読み出し（Read）」「更新（Update）」「削除（Delete）」の 4 つの主要な機能についてのみ考える。この CRUD 処理はそれぞれ SQL の「insert 文」,「select 文」,「update 文」,「delete 文」を利用して行われる。これらのにも、行の選択・テーブルの選択・検索条件・テーブルの結合など多くの要素が存在しているが、こうした要素すべてが動的に決定されているわけではない。Web アプリケーションなどにおいて実際に、動的に決定される SQL の要素は

1. セルに代入する値や検索条件などに使われる「式」
2. select 文で出力結果をソートするための「ソート式」

ぐらいである。そこで本研究では動的に決定されるのはこの「式」と「ソート式」だけであると仮定して、フレームワークの設計を行った。

5.2 式の動的な構築

4.1 の終わりに述べたように、バインディングメカニズムを利用すれば、SQL 中の式に現れる値が動的に決定

する場合について、集合の定義と集合からのプログラムの選択が効率的に行うことができる。しかしバインディングメカニズムを利用した方法では、値だけではなく式の構造が動的に決定されるような場合には対応することができない。そこで本フレームワークでは我々が提案したバインディングメカニズムを拡張し、動的に有無が決定される条件および条件の繰り返しを効率的に取り扱うことができる手法 [10] を利用して、集合の定義とプログラムの選択を行う。例えば顧客情報の格納されたテーブル customers からユーザ名・誕生日の範囲（何日から何日までの間）を検索条件として情報を取得する場合を考える。ここで各条件は省略することも可能であるとする。この場合 [10] の手法を用いると、SQL の集合の定義は「select * from customers where name = @name and birthday >= @from and birthday <= @to」と書くことができる。そして例えば、name が “Alice” で誕生日が 2042/01/01 までのユーザに関する情報を取得する SQL を生成したい場合、@name に “Alice” を、@to に “2042/01/01” を、そして使用しない条件@from にはその条件を利用しないことを表す特殊値をバインドすることで、「select * from customers where name = “Alice” and birthday <= “2042/01/01”」という SQL を得ることができる。より詳細については [10] や実装のドキュメントを参照されたい。

5.3 実装

以上で述べた SQL インジェクション対策用のフレームワークを.NET Framework のデータベース接続ライブラリである ADO.NET のライブラリとして実装を行った。なお本研究では実装の都合上、中継者はデータベースから独立したプログラムとして実装を行ったが、原理的には中継者をデータベースのフロントエンドとして実装することも可能である。

6 長所と短所

6.1 長所

最後に本研究で提案したフレームワークの長所と短所について考察する。長所は 3 で述べたとおり、インジェクション系の問題によってシステムのセキュリティおよび正しさが脅かされていないことを

- プログラマ任せにせずに
- 検出技術のように本質的に不確実さを持つ技術を利用することなく

保証することができる点である。そのため誤検知や検知漏れに悩まされることなくシステムの安全性を確保できる。

6.2 短所

本提案手法のもっとも致命的な短所は、本手法を既存のシステムに導入するためには確実にソースコードの書き換えが必要になるということである。そのため既存のアプリケーションに本手法を導入する際にはソースの修正が必要になるのはもちろんのこと、新しいアプリケーションを作成する場合であっても、既存の開発補助ツールを利用する場合にはそのツールを本手法に合わせて修正する必要がある。

7 おわりに

本稿ではまずインジェクション系の脆弱性を回避するための既存技術について紹介を行い、技術間の比較を行った。次に検出技術などの不確実さを本質的に抱える技術を利用することなく、システムがインジェクション系の問題を持たないことを保証するための新しいアーキテクチャを提案した。そして、このアーキテクチャを基本とする SQL 対策用のフレームワークを設計し実装を行い、最後に本稿で提案したフレームワークの長所と短所について考察を行った。

参考文献

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

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

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

[4] 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.

[5] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, Vol. 23, No. 5, pp. 279–295, 1997.

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

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

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

[9] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2007.

[10] 渡邉 悠 松浦幹太. SQL の条件説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. In *Computer Security Symposium*, pp. 571–576, 奈良, 2007.

[11] 川中真耶. 異常木検知による javascript injection 検知. In *Computer Security Symposium*, pp. 417–422, 奈良, 2007.

[12] 金子勇. Winny の技術. アスキー, 10 2005.