Web アプリケーションセキュリティの最近の動向

小柳和子*

概 要

Webシステムの脆弱性として クロスサイトスクリプティングとSQLインジェクションの二つが代表として挙げられる. これらの脆弱性の主な対策として現在主に(1) defensive coding, (2) test, (3) static analysis, (4) dynamic monitoring, (5) Browser 側の対策 の 5 種類がある. (1)から(4)は, Webサーバーのアプリケーションプログラムに対する対策である. (1)は新規開発時に主に適用される手法である. (2)(3)(4)はシステムのテスト時や運用時の利用が適している. (5)はユーザ側の Browser での対策である. それぞれの手法の最近の研究動向について述べる. 現状では 100%有効な手法は存在しない.

1 はじめに

Web アプリケーションは, 近年, ますますミッションクリティカルな分野のシステムに利用されている. しかしWebアプリケーションシステムからの情報漏えいやWebコンテンツの書き換えなどの事件は現在でもしばしば発生している.

このような事件を防ぐためにソフトウェア工学の立場からセキュリティに関する要求項目を洗い出し, それらをシステム要件として設計に系統的に組み入れようとするアプローチがなされている. [1][2] これらは大変重要であるが, セキュリティの場合詳細設計以降でどのようにソフトウェアを開発するかにも注意を払う必要がある. ここではコーディング, テスト, 運用において Web のセキュリティに関しての最近の研究について概要を述べる.

Web アプリケーションでは, 新規の開発の際に脆弱性を盛り込まないようにすることだけでなく, 既存のシステム内の脆弱性を検出することも重要である. 現在多くの Web アプリケーションが脆弱性を含んだまま利用されている. また現在発見されていない脆弱性もあり得るので, 運用後の脆弱性の発見・除去は必須の技術である.

2 代表的な Web の脆弱性

Web の脆弱性の代表的なものとして, OWASP†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(SQLI)をあげている. [3] また IPA の情報セキュリティ白書 2009 でもこの二つを最も代表的な脆弱性としている. [4] ここではこの二つの脆弱性を取り上げる. こ

* 情報セキュリティ研究科 教授
† Open Web Application Security Project , http://www.owasp.org/

れらの脆弱性を利用した行為は基本的にユーザの入力によって引き起こされる. ユーザの入力によってアプリケーション開発者の意図しないコマンドが生成され, それにより悪意ある行為が起こる. XSS はユーザ側のブラウザで意図しない JavaScript を起動させる. SQLI は意図しない SQL コマンドが実行される.

2.1 XSS

XSS には三タイプがある.

(1) stored type (second-order XSS)

ユーザの入力がサニタイズされないまま 一旦ファイルやデータベースに蓄積されそれがユーザ要求により検索され結果をレスポンスとして返す場合に発生する.

例) Comment: <? echo $msg; ?>

悪意ある入力 <script>alert('xss');</script>

例題での入力は無害なものであるが, ユーザのブラウザ上で JavaScript が動いてしまう. ここに <script>document.location = 'http://malicious.com/?' + document.cookie </script> と入力した場合は cookie 情報が malicious.com に送られる.

(2) reflected type (first-order XSS)

ユーザの入力がそのままユーザに返される.

例) <? echo $_GET('fname'); ?> Not found.

悪意ある入力 http://xxx.xxx?fname=<script>alert('xss');</script>

本来ユーザが入力したファイル名が存在しない場合のエラーメッセージであるが, ファイル名の代わりに上記入力を入れると, JavaScript が動く.

(3) DOM-based type

DOM を利用したもの

例) var name = document.URL.indexOf("name=") + 5;
document.write("Hello " + name);

悪意ある入力 http://xxx.xxx#name=<script>alert('xss');</script>

2.2 SQLI

ユーザ名と pin＃を入れて SQL 検索をする例を考える.

SELECT acct FROM users WHERE login='_____ ' AND pin='_____'

login に"Joe" pin に"123"を入れた場合は以下のように正常な検索になる.

SELECT acct FROM users WHERE login= 'Joe' AND pin='123'

(1) SQLI の例1

login : "admin' -- " pin: 何でも良い

SELECT acct FROM users WHERE login='admin' -- ' AND pin='0'

pin＃を入れなくても admin の検索ができる. -- 以下はコメントとなるため pin#の条

件は無視される.

(2) SQLI の例2

login : " ' OR 1=1 -- "　pin: 何でも良い

SELECT acct FROM users WHERE login=' 'OR 1=1　-- ' AND pin='0'

where 句は, 常に真なため, すべての login が検索される.

(3) SQLI の例3　UNION 利用

login: " 'UNION SELECT cardno from CreditCards where acctNo=11 -- "

SELECT acct FROM users WHERE login=' 'UNION SELECT cardno from CreditCards where acctNo=11-- ' AND pin=

最初の SELCT 文の結果は null となり, 後半はどんな SELECT 文でも可能となる. Union を利用することにより, 後半の SELECT 文の結果が返される.

(4) SQLI の例4　複文利用例

login: "Joe" pin: "0'; drop table users"

SELECT acct FROM users WHERE login='Joe'　AND pin='0' ; drop table　users

";"による複文を許すデータベースの場合, 後半どんな SQL 文でも書ける. この例では drop 文が実行されデータベースが破壊される.

(5) SQLI の例5　エラーメッセージによる情報の流出

上記の例3, 例4を実行するためにはデータベースのテーブル名を知る必要があり, その例である.

login:　なし　pin: "convert(int,(select top 1 name from sysobjects where xtype='u'))"

SELECT acct FROM users WHERE login='' AND pin=convert(int,　(select top 1 name from sysobjects where xtype='u'))

sysobjects テーブルの最初の行の name を検索し, それを数値に変更し pin#とする命令のため, エラーメッセージ"xxx は数値にできない" が返される. (xxxは sysobjects の最初の行, すなわちテーブル名称である) これにより攻撃者はデータベースのテーブル名を得ることができる.

想定外の入力によってプログラムの構文が変化してしまうバグは, Web を利用したシステムに固有のものではなく従来のシステムでも存在していた. しかし Web 以前のシステムでは, それぞれのシステムの利用者は特定された人々であったため, 誰が入力したのか比較的簡単に特定される. このため, このような脆弱性があったとしても運用上はそれほど支障を生じない. それに反してインターネットでは, 入力は世界中のどこからでも, いつでも, 誰でも可能であり, そして入力者の特定は難しいことが多い. このため, この脆弱性を利用してシステムを攻撃する事件がしばしば起こっている.

3　脆弱性の検出手法　または脆弱性を防ぐコーディング

Web アプリケーションプログラム上でのユーザ入力のチェック漏れが情報流出の原因であるので, チェック漏れが起こらないコーディングを強制するライブラリを提供すれば良いという考え方が(1)Defensive Coding である. これは新規にソフトウェアを開発する時は大変有用であるが, 現在すでに運用されている多くのプログラムをそのようなライブラリに書き換えるというのはコストの面であまり現実的ではない.

そこで既存のプログラムから脆弱性を検出しようというのが(2)test と(3)Static Analysis である. この手法の有用性の判断は false negative(偽陰性:プログラム内の脆弱性を見逃すこと)と false positive(偽陽性: 脆弱性ではないのに, 誤って脆弱性と報告されること)の両者をいかに少なくするかである.

これらの手法には限界があるため, あらかじめ Static Analysis を実施しそれと(5)運用時のモニタリングと併用する方法が考えられている. この手法では Static Analysis 単体での利用に比べて脆弱性の検出数は上がるが, 運用時に行うためモニタリングの速度が重要となってくる.

以上の手法はサーバー側のアプリケーションプログラムに着目をしている. しかし XSS の脆弱性は Web アプリケーションプログラムの多くの箇所に潜んでおりサーバー側のチェックだけではすべてを防ぐのは難しい. そこで(6)browser 側にも脆弱の検出の機構を入れる試みがされている.

それぞれの手法が主にどの脆弱性の検出/防御に利用されているかを表1に示す.

表1 脆弱性と各手法の対応

手法	XSS	SQLI
defensive coding	○	○
test	○	○
static analysis	○	–
dynamic monitoring	–	○
browser	○	–

3.1　Defensive coding [5]-[8]

バインドメカニズムは, SQLI を防ぐため各リレーショナルデータベースのライブラリに Java, PHP, Perl などの言語で用意されている. これは実行時に SQL の構文を変更させないための仕組みであり, 古典的な defensive coding の手法である. バインドメカニズムは開発者がその使用方法を誤らない限り有効な方法である. しかし複雑な SQL コマンドー例えばユーザの入力により SQL 構文が変化するーを実行する場合はバインドメカニズムを利用しにくい. そこで, 新しいライブラリ/フレームワークが提案されている. 例えば, Okubo ら[7]は Java 言語で新しいフレームワークを提案し, 上記問題点を回避している.

また, [8]のようなセキュアなプログラミング手法の提唱もこの defensive coding のひとつであると考えられる.

3.2 Test [9]-[12]

Web アプリケーションプログラムのテスト手法には，ブラックボックステストとホワイトボックステストがある． ブラックボックステスト[9]は，プログラムの内容を考慮せずに脆弱性を示す数多くのパターンを入力しテストするものである．現在商用で利用されているペネトレーションテストはこの手法である．これは自動化が出来，多くのアプリケーションプログラムのテストを簡単に出来るという利点はあるが，false negative が多く生じるという問題点もある．

一方，様々なホワイトボックステスト手法が提案されている．Shahriar ら[10]は，HTML および JavaScript の構文から XSS が起こりうるパターンを 11 個定め，それぞれのパターン毎にプログラムの構文を変更させる．プログラム内でその 11 個の構文の出現時に，元のプログラムと変更させたプログラムの両方を用意する．その二種類のプログラムに XSS が起こるパターンを入力させ，出力に構文の変化があるかをチェックする．構文が変化していた場合は，プログラムに XSS の脆弱性がある．この手法の場合，従来検出されにくかった DOM-based XSS の検出が可能である．

Kieyzen ら[11]は，ARDILLA を作成し，SQLI と XSS のホワイトテスト手法を提案している．プログラムを解析し脆弱性の入る可能性のある箇所を検出し，そこに影響を及ぼす入力を見つける．次に，その入力を変化させて実行し脆弱性があるかをテストする．この手法で従来テストが難しかった second-order XSS の検出を行っている．ARDILLA のアーキテクチャを図1に示す．

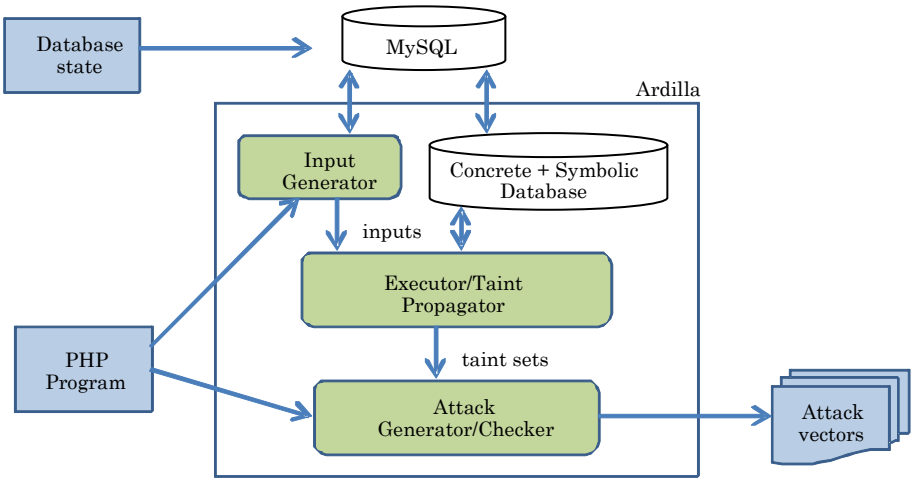


図1 ARDILLA のアーキテクチャ（Kieyzun 他著[11], p.202 Figure 2 より転載）

3.3 Static Analysis [13]

当初，XSS を防ぐためには，ユーザ入力から”<script>”や”javascript”のような文字列をフィルタリング（サニタイジング）できれば良いと考えられていた．しかし，2005 年に MySpace Worm(別名 Samy Worm)事件‡が発生した．これは XSS の脆弱性を利用しているが[14]，見掛け上，上記のような文字列は使わずエンコードした文字列や文字列の連

‡ Kotadia, M. : Samy opens new front in worm war (2005)
http://news.cnet.com/Samy-worm-opens-new-front-in-malware-war/2100-7349_3-5897099.html

結を巧みに利用し，フィルタリングをすり抜けている．この事件により，従来の入力文字列のフィルタリングだけでは不十分であることが広く認知され，プログラムの静的解析が考案された．

Wasserman ら[13]は，フィルタリング(サニタイジング)を行っているモジュールを解析し形式言語に変換し，入力データが十分にチェックされているかの検証を行った．彼らは合わせて Browser を分析し，タグの内側で JavaScript が起動され得る箇所を調査し，起動されるコマンドを形式言語に変換した．ユーザの入力データはすべて untrusted と仮定し，JavaScript が起動され得る箇所で入力データ/入力データの伝播したものが JavaScript を起動しているかのチェックを行った．

Static Analysis 手法の場合，一般に false positive が多くなってしまう傾向がある．

また，上記 MySpace Worm のような事件を防ぐために OWASP は現在 AntiSamy Project[15]を実施中である．これは Web サーバーを運用している組織に対して入力をフィルタリングするライブラリを提供しようとするものである．各組織は自分自身のフィルタリングポリシーを記述でき，このライブラリと併用することによって各組織のポリシーに基づいたフィルタリングが実行可能である．これは手法としては Static Analysis ではなく，defense coding の分類に入る．

3.4 Dynamic Monitoring （Static Analysis と組み合わせて）[15]-[21]

実行時にユーザの入力を何らかの方法でマークをし，それが データベース側に渡る直前に正しいかどうかをチェックする．このチェックを有効にするため，あらかじめプログラムに何らかの解析を行っておくのが普通である．この Dynamic Monitoring 手法は一般に SQLI を防ぐ手法として用いられる． SQLI はチェックする箇所が SQL データベースに制御を渡す部分に特定出来る．しかし一方 XSS はチェックすべき場所―JavaScript が起動され得る箇所―がプログラム中に多く多様な形で存在するため，Dynamic Monitoring の利用は難しい．また Dynamic Monitoring は実行時に行われるため、実行速度を落とさないという課題がある。

Halfond ら[19]は AMNESIA というツールを実装した．まずあらかじめ Web アプリケーションプログラム内の SQL 文を実行する箇所を探す．次にその構文を解析しておく．実行時，(ユーザ入力により完成された)SQL 文をデータベースに渡す直前に解析し，元の構文と同じかどうかを確認する．同じ場合のみ正しい入力である．これを図2，図3に示す．図 2 は SQL 構文を解析したものである．図3(a)は，図 2 の構文解析とマッチングする．しかし，図 3(b)は，本来 AND キーワードであるべきところに OR キーワードがあり正しくない．また，AMNESIA のアーキテクチャを図4に示す．これはバインドメカニズムの利用の如何にかかわらず実行できる．しかし，実行時にユーザ入力により SQL 構文が変化してしまう場合は利用できない．

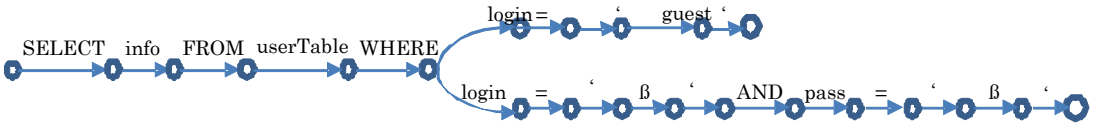
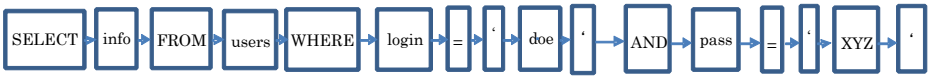
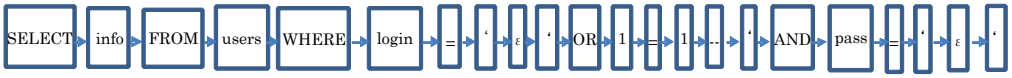


図2 SQL 構文解析モデル(Halfond 他著[19], p.794 Figure 2 より転載)



(a) SELECT info FROM users WHERE login = 'doe' AND pass = 'xyz'



(b) SELECT info FROM users WHERE login = '' OR 1=1 – 'AND pass = ''

図3　実行時の解析例（Halfond 他著[19], p.797 Figure 4 より転載）

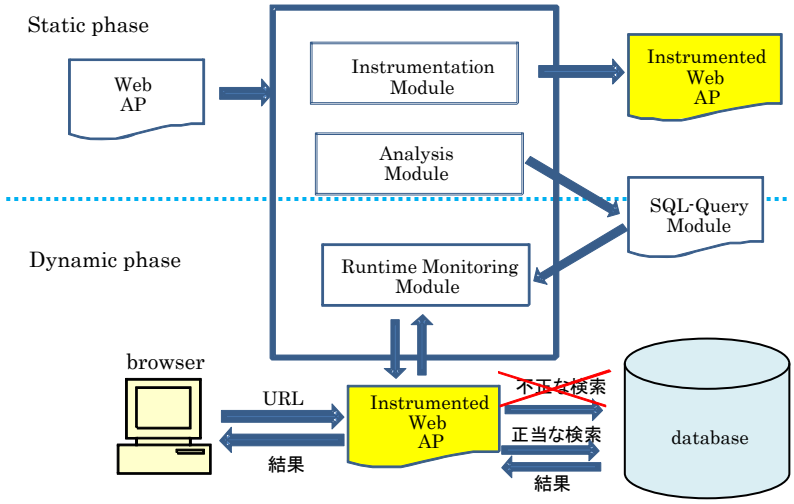


図4　AMNESIA のアーキテクチャ(Halfond 他著[19], p.797 Figure 5 より転載)

Halfond ら[20]は，さらに上記の手法を WASP という動的モニタリングシステムに拡張した．WASP は Java で書かれたサーバプログラムを対象としている．このシステムの概念は以下のものである．

(1) positive tainting

通常の taint propagation は，ユーザの入力を untrusted としてマークし，それが実行中どのように伝播していくかを追跡する．しかし，このシステムでは逆にプログラム内であらかじめ定義された SQL 文のキーワードを trusted としてマークする．

(2) 正確な taint propagation

マーキングはストリングレベルでなく文字レベルで行う．これを行うためにストリングクラスを拡張しメタストリングクラスを作成し，その中にマークを入れる．SQL 文を構成するストリングインスタンスが生成されるとき，このメタストリングクラスを用いてマーク済みのストリングを生成する．

(3) Syntax-aware evaluation

データベースに SQL 文が渡る直前に SQL 文内のストリングをチェックする. SQL 文を構成するキーワードがすべてマークされていたら正しい入力である. これを図5, 図6に示す. 図中で長方形で囲まれた文字列が SQL 文のキーワードである. マークされたキーワードには下線が引かれている. 図5ではすべてのキーワードがマーク済みであるため正しい入力である. しかし 図6では最後尾の “ ‘ “, “AND”, “=”がマークされていないため不正な入力である.

図5 ランタイムモニタの解析結果例1(正しい入力)
(Halfond 他著[21], p.76 Figure 8 より転載)

図6 ランタイムモニタの解析結果例2(不正な入力)
(Halfond 他著[21], p.76 Figure 9 より転載)

このような概念で実装することにより, 本来のプログラムの変更を最小限度にし, また実行時の性能も確保した.

3.5 Browser 側の対策[22]-[25]

SQLI の発生する箇所を Web アプリケーションプログラムの中で特定するのは比較的簡単である. しかし, XSS の発生する箇所を Web アプリケーションプログラムの中で特定するのは難しい. JavaScript は HTML コンテンツの多くの場所で起動可能のためである. そのため JavaScript の起動をサーバー側ではなく Browser 側で制御しようとする試みがいくつか提案されている.

Iha ら[23]の手法は厳密には Browser だけではないが, この分類に入れる. あらかじめサーバプログラムは, 生成するコンテンツをユーザ入力値によって生成された部分と, それ以外の部分とを分けておく. 前者は http ヘッダの一部として, 後者は通常のデータとしてサーバーからレスポンスとしてブラウザに返すようにする. Browser の中の JavaScript を起動する直前でこれらをバインドすることにより, 意図しない JavaScript の起動を回避できる. この手法を実現するためには, サーバー側のアプリケーションプログラムも一部変更する必要がある.

Ofuonye ら[24]は, サーバーからレスポンスとして送られてくる HTML コンテンツ内の JavaScript のコードをそれぞれのポリシーに基づき同等のより安全なコードに書き換える手法を提案している. 書き換えるオブジェクトは window, document, image,

XMLHttpRequest の 4 種類である.

4 今後の Web セキュリティ

現在までの Web アプリケーションセキュリティの手法について述べてきた. 100%有効な手法は存在しないため, 多種類の手法を併用して Web アプリケーションプログラムを開発し運用する必要がある.

最近はクロスサイドドメイン通信を利用したマッシュアップが利用され始めた. ユーザの利便性を高め, クライアントをよりリッチにしたいという要求からでたものであろう. これは現在の実装手法は JavaScript を利用したものや html タグの src 属性を利用したものが多い. これらは現存する SQLI や XSS の脆弱性をそのまま持っており, プログラムが複雑になるため従来の手法で脆弱性を見つけるのがますます困難になると思われる. また W3C は HTML5[26]で クロスドメイン通信を許す仕組みを検討中であり, それが実装されたときのセキュリティに関しては今後検討が必要であると思われる.

参考文献

[1] 吉岡信和, Bashar Nuseibeh:セキュリティ要求工学の概要と展望:吉岡信和, 情報処理学会誌, Vol.50, No.3, pp.187-192 (2009)

[2] Common Criteria; http://www.commoncriteriaportal.org/

[3] OWASP Foundation , Top 10 Web application vulnerabilities for 2007, http://http://www.owasp.org/index.php/Top_10_2007

[4] 10 大脅威攻撃手法の『多様化』が進む, 情報セキュリティ白書 2009 第 2 部, http://www.ipa.go.jp/security/vuln/documents/10threats2009.pdf

[5] Cook, W.R. and Rai, S.: Safe query objects: statically typed objects as remotely executable queries, Proc. 27th International Conference Software Engineering, pp.97-106 (2005)

[6] Okubo, T. and Tanaka, H.: Secure Software Development through Coding Conventions and Frameworks, Proc. 2nd International Conference on Availability, Reliability and Security, pp. 1042 – 1051 (2007)

[7] 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, Proc. Symposium on Cryptography and Information Security (2009)

[8] セキュアプログラミング講座: 情報処理推進機構: http://www.ipa.go.jp/security/awareness/vendor/programmingv2/index.html

[9] Fonseca, J: Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks, Proc. 13th IEEE International Symposium on Pacific Rim Dependable Computing, pp.365-372(2007)

[10] Shahriar, H. and Zulkernine, M.: MUTEC: Mutation-based testing of Cross Site Scripting: Proc. 5th International Workshop on Software Engineering for Secure Systems, pp.47 - 53(2009)

[11] Kieyzun, A., Guo, P.J., Jayaraman, K. and Ernst, M.D.: Automatic creation of SQL Injection and cross-site scripting attacks, Proc. 31st International Conference on Software Engineering, pp.199 – 209 (2009)

[12] Shin, Y., Williams, L and Xie, T.: SQLUnitGen: Test Case Generation for SQL Injection Detection, Raleigh Technical report, NCSU CSC TR 2006-21,(2006)

[13] Wassermann, G. and Zhendong Su: Static detection of cross-site scripting vulnerabilities, Proc. 30th International Conference on Software Engineering, pp.171 - 180 (2008)

[14] Technical explanation of the MySpace worm： http://namb.la/popular/tech.html
[15] OWASP Foundation: OWASP AntiSamy Project. http://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project
[16] Wassermann, G. and Zhendong S.: Static detection of cross-site scripting vulnerabilities, Proc. 30th International Conference on Software Engineering, pp.171 - 180 (2008)
[17] Di Lucca, G.A., Fasolino, A.R., Mastoianni, and M., Tramontana. P.: Identifying cross site scripting vulnerabilities in Web applications: Proc. 6th IEEE International Workshop on Web Site Evolution, pp. 71 – 80 (2004)
[18] Buehrer, G.T., Weide, B.W. and Sivilotti, A.G.: Using Parse Tree Validation to Prevent SQL Injection Attacks, Proc. 5th International Workshop on Software Engineering and Middleware, pp.106-113 (2005)
[19] Halfond, W.G.J. and Orso, A.: Preventing SQL injection attacks using AMNESIA, Proc. 28th International Conference on Software Engineering, pp. 795 – 798 (2006)
[20] Halfond, W.G.J. and Orso, A: AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks, Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (2005)
[21] Halfond, W.G.J. and Orso, A.: WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation, IEEE Transactions on Software Engineering, Vol. 34, No. 1, pp. 65-81 (2008)
[22] 小菅裕史,花岡美幸, 河野健二 S: QL インジェクション攻撃の脆弱性の効果的な自動検出手法, 情報処理学会研究報告, CSEC 2008(45), pp.103-108
[23] Hallaraker, O., Vigna, G : Detecting Malicious JavaScript Code in Mozilla, Proc. 10th Engineering of Complex Computer Systems, pp. 85 – 94 (2005)
[24] Iha, G. and Doi, H.: An Implementation of the Binding Mechanism in the Web Browser for Preventing XSS Attacks: Introducing the Bind-Value Headers: Proc. 2009 International Conference on Availability, Reliability and Security, pp.966 – 971(2009)
[25] Ofuonye, E. and Miller, J.: Resolving JavaScript Vulnerabilities in the Browser Runtime, Proc. 19th International Symposium on Software Reliability Engineering pp. 57 – 66 (2008)
[26] Tiwari, S., Bansal, R. and Bansal, D.: Optimized client side solution for cross site scripting, Proc. 16th IEEE International Conference on Networks, pp.1 – 4 (2008)
[27] HTML 5 differences from HTML 4, W3C Working Draft 25 August 2009; http://www.w3.org/TR/2009/WD-html5-diff-20090825/