Realization of Effective XSS Attack Countermeasure based on the Combination of Examination-based Whitelist and CSP

情報処理学会論文誌 Vol.61 No.9 1374–1387 (Sep. 2020)

テストベースホワイトリストとCSPの組合せによる効果的なXSS対策の実現

井上 佳祐[1] 本多 俊貴[1] 向山 浩平[1] 大木 哲史[1] 堀川 博史[2] 西垣 正勝[1,a)]

受付日 2019年11月26日, 採録日 2020年6月1日

概要：Software as a Service（SaaS）などのクラウドサービスの普及にともない，Web アプリケーションに対する攻撃が急増している．本論文ではクロスサイトスクリプティング（以下，XSS）に焦点を当て，その対策を検討する．現在，XSS の効果的な対策として Content Security Policy（以下，CSP）が普及しつつある．CSP を利用して，インラインスクリプトに対してはその動作を禁止し，外部スクリプトに対してはそのコード署名を検証することで，開発者が意図したスクリプトのみを動作させることが可能となる．しかし，現在の Web サービスにおいてインラインスクリプトを利用していない Web サイトは数少ないため，CSP のみでは十分な XSS 対策を実現し得ない．そこで本論文では，CSP とホワイトリストを併用することによって，効果的な XSS 対策を達成する．提案方式は，外部スクリプトに対しては，CSP のコード署名によって開発者の意図したスクリプトのみの実行を許可する．提案方式は，コード署名による対策が難しく，サニタイジングの不備の影響を大きく受けるインラインスクリプトに対しては，テストベースホワイトリストを用いた XSS 対策を提案する．テストベースホワイトリスト方式では，開発プロセスの最終段階で行われるソフトウェアテストを通じ，各 Web アプリケーションの仕様に合致するホワイトリストが自動生成される．

キーワード：クロスサイトスクリプティング，Content Security Policy，ホワイトリスト，自動生成，テストケース

Realization of Effective XSS Attack Countermeasure based on the Combination of Examination-based Whitelist and CSP

Keisuke Inoue[1] Toshiki Honda[1] Kohei Mukaiyama[1] Tetsushi Ohki[1] Hiroshi Horikawa[2] Masakatsu Nishigaki[1,a)]

Received: November 26, 2019, Accepted: June 1, 2020

Abstract: Owing to the widespread usage of cloud services (such as “software as a service”), attacks targeting web applications are rapidly increasing. In this study, we focus on Cross Site Scripting (XSS) attacks and consider the corresponding countermeasures. Currently, Content Security Policy (CSP) is considered as an effective countermeasure for addressing XSS attacks. CSP can prevent XSS attacks by prohibiting script action of inline scripts and verifying external scripts using their code signatures. However, a large number of websites do use inline scripting in their web services, and they are therefore not protected by solely utilizing CSP. Hence, our objective is to develop effective XSS countermeasures by combining whitelists with CSP. The proposed method employs CSP for protecting external scripts by incorporating the associated code signatures. Additionally, the proposed method incorporates whitelists as countermeasures for addressing inline scripts that are difficult to manage. Through the software testing performed during the final stage of the development process, it is possible to automatically generate a whitelist that matches the specifications of each web application.

Keywords: Cross Site Scripting, Content Security Policy, whitelist, automatic generation, test cases

1. はじめに

コンピュータの性能向上とネットワークの広帯域化にともない，Web 上のコンテンツは従来の静的なコンテンツのみの Web ページから，動的なコンテンツを扱う Web ア

[1] 静岡大学
Shizuoka University, Hamamatsu, Shizuoka 432–8011, Japan
[2] 三菱電機インフォメーションネットワーク株式会社
Mitsubishi Electric Information Network Corporation, Minato, Tokyo 108–0023, Japan
[a)] nisigaki@inf.shizuoka.ac.jp

プリケーションへと移行している．Software as a Service（SaaS）などのクラウドサービスの普及により，Web サービスはリッチかつ柔軟なサービスへと進化した．今や，Web サイトの大部分は，JavaScript ライブラリやコンテンツ管理システム（CMS），Web 開発用のプラットフォームなどを使用し，クライアント側とサーバ側の両方のプログラムを利用して動的にコンテンツが生成されている．その結果，従来の（Web アプリケーションではないタイプの）アプリケーションで発生するような脆弱性が Web サービスにおいても発生するようになり，Web アプリケーションに対するサイバー攻撃の急増を招いた．

クロスサイトスクリプティング（XSS）攻撃も，Web アプリケーションに対するサイバー攻撃の 1 つである．XSS は，Web サイト側の「ユーザから受け取る入力値が適切なものかどうかをチェックする機能」や「有害な入力値を無害化する機能」の欠陥によって生まれる Web アプリケーションの脆弱性である．攻撃者は，この脆弱性を悪用することで，ユーザの Web ブラウザ上で任意のスクリプトを実行することが可能となる．SQL インジェクションなどの他の Web アプリケーションの脆弱性は，サービス提供者側の Web サーバに影響を与えるのに対し，XSS はユーザの PC に直接影響を与える．したがって，XSS に対する効果的な追加対策を行うことが急務となっている．

この脆弱性が発生する主な原因は，Web アプリケーションの開発時に実装すべき XSS 対策が不完全なことにある．この観点から，サニタイジングによって Web アプリケーションに対する入力値の無害化 [1] を徹底することができれば，XSS を防止できることが分かる．しかし，対策漏れを完全に防ぐことは難しく，ゼロデイ攻撃に対しては後追いの対策とならざるを得ない．したがって，サニタイジングによって無害化を図るという方法は，十分かつ容易な XSS 対策となり得ていないという現状にある．

XSS の実態は，攻撃者による Web アプリケーションへの「開発者の意図しないスクリプト」の注入である．この観点から，ポリシに基づくホワイトリスト型対策を採用し，開発者の意図したスクリプト（すなわち事前にポリシに定義されたスクリプト）のみを使用可能とする方法が有効であることが分かる．その具体的な仕組みが Content Security Policy（CSP）であり，すでに多くの Web サーバや Web ブラウザに実装されている [2]．しかし，CSP が正しく機能するのは，Web アプリケーションにおいて動作する外部スクリプト[*1]に限定されるという点が課題として残る．外部スクリプトは，スクリプト本体は静的なプログラムコードとなるため，スクリプトプログラムに対するコード署名を検証することによって，当該スクリプトの真正性を検査可能である．これに対し，インラインスクリプトを含むような動的な Web アプリケーションの場合は，ユーザからの入力やデータベース中の登録内容に応じて HTML ソースコードが動的に変化するため，コード署名による真正性検査を適用することができない．

すなわち，CSP による XSS 対策を完全に行うには，default-src [3] あるいは script-src [4] の宣言記述によってインラインスクリプトを禁止したうえで，Subresource Integrity（SRI）の機能 [5] を利用して外部スクリプトをそのコード署名によって検証するという運用を行う必要がある．しかし，インラインスクリプトは汎用性に富むプログラム要素であり，現在の Web サービスにおいてインラインスクリプトを利用していない Web アプリケーションは皆無といっても過言ではない．したがって，サニタイジング処理の不備により意図しないスクリプトがインラインで注入される恐れが依然として残っているという現状にある．

この問題を解決するために，本論文では，インラインスクリプトに対するホワイトリスト型対策の実現方式について検討を行っていく．外部スクリプトに対するホワイトリスト型対策である CSP と，本論文で論ずるインラインスクリプトに対するホワイトリスト型対策を併用することによって，XSS を効果的に防ぐことが可能となる．

インラインスクリプトに対するホワイトリスト対策を実現するにあたっては，「開発者が意図したスクリプトであるか否か」を判定するためのポリシをいかに策定するのかが課題となる．そこで本論文では，テストベースのアプローチによってホワイトリストを作成する方法を提案する．具体的には，ホワイトリストの作成を，Web アプリケーション開発の最終段階として実施されるソフトウェアテストと結合する．テスト工程で確認された動作をホワイトリストとして定義することにより，Web アプリケーション利用時に想定外の入力（未テストの入力）によるインラインスクリプトが実行されることを禁止する．各 Web アプリケーションの仕様に基づいて行われるソフトウェアテストを通じて，仕様を満たす（仕様と一致する）ホワイトリストが自動的に生成される．

なお本論文は，文献 [6], [7], [8] を基に，研究内容の精緻化と実装および評価を行ったものである．

2. XSS の攻撃メカニズムと既存対策

2.1 XSS 攻撃

XSS は，「攻撃者」，「被害者」，XSS 脆弱性のある「標的 Web アプリケーション」の 3 者が関与して成り立つ．XSS の脆弱性には，Reflected XSS，Stored XSS，DOM Based XSS の 3 種類が存在する．本節では，このうち，Reflected XSS を例にとり，そのメカニズムを説明する．

攻撃者は，標的 Web アプリケーションに対して有害なリクエストを発生されるためのリンク（XSS 脆弱性を突く

[*1] Web アプリケーションとは独立したコードであり，Web アプリケーションの実行時に Web アプリケーションによって読み込まれるスクリプト

パラメータが埋め込まれた標的 Web アプリケーションの URL）を E メールに記載するなど，何らかの方法で被害者に送信する．被害者は，そのリンクにアクセスすることにより，攻撃者の指定した有害なリクエストを被害者の Web ブラウザを通して標的 Web アプリケーションに送信する．標的アプリケーションは，有害なスクリプトを含んだ Web コンテンツを被害者の Web ブラウザに対してレスポンスする．被害者のブラウザはレスポンスに含まれた有害なスクリプトを実行してしまう．

XSS 脆弱性が悪用され，Web ブラウザ上で任意のスクリプトが実行されてしまうと，標的 Web アプリケーションのセッション ID などが奪取され，その Web アプリケーションに対して攻撃者による被害者へのなりすましを許す恐れや，攻撃者が偽のログインページを表示させて被害者の認証情報を盗む恐れ，また，攻撃者が用意した有害な Web サイトへと被害者を誘導してドライブバイダウンロード攻撃を行う恐れなどが生じる．このような有害なスクリプトの動作はバックグラウンドで行われ，スクリプトの動作が被害者に対して明示的に示されることは少なく，攻撃を受けていても被害者は気づかない場合が多い．

2.2 サニタイジング

XSS における有害なスクリプトは，攻撃者がリンク情報の中に記載する不正なパラメータや，データベースに格納された悪性なパラメータを含む入力値を読み込むことにより Web アプリケーションに注入されてしまう．したがって，Web アプリケーションに対する入力値を適切に無害化することによって，XSS を防止できる．これは，「サニタイジング」と呼ばれる XSS 対策である．サニタイジングは主に入力値に対するブラックリスト型の「入力値チェック」と「エスケープ処理」で構成される．これらの処理は，Web アプリケーション開発時に開発者によって実装される．

2.2.1 入力値チェック

Web アプリケーションが，たとえば，「`<script>` 有害なスクリプト `</script>`」）というパラメータを受け取った際に，そのなかにスクリプトを示す HTML タグ（この例では `<script>` タグ）が含まれているかをチェックし，そのパラメータを無効にすることによって有害なパラメータを無毒化する．

2.2.2 エスケープ処理

Web アプリケーションが，たとえば，「`<script>` 有害なスクリプト `</script>`」というパラメータを受け取った際には，Web アプリケーション側でエスケープ処理を行い，「`<`」を「`&lt;`」に，「`>`」を「`&gt;`」にそれぞれ変更する．これによって，Web アプリケーションから Web ブラウザへ送信されるレスポンスにおけるパラメータは「`&lt;script&gt;` 有害なスクリプト`&lt;/script$gt;`」となり，ブラウザ上では「`<script>`有害なスクリプト`</script>`」という文字列として認識される．

2.2.3 問題点

Web ブラウザでスクリプトとして認識されるタグやイベントは「`<script>`」以外にもあり，単純なエスケープ処理を意識するだけでは対策漏れが生じることも少なくない．セキュリティ対策に十分なコストを費やしている大手 IT 企業の Web サービスにおいても脆弱性が発見されている [9] ことに鑑みると，多様化した悪意ある入力を完全に無害化することは，開発者の能力にかかわらず，困難な作業であることが予想される．また，今まで知られていなかった（ゼロデイ）脆弱性や攻撃手法が新たに発見される場合もある．そのため，サニタイジングは十分かつ容易な XSS 対策とはなっていない可能性がある．

2.3 Content Security Policy

XSS における有害なスクリプトは，Web ブラウザ上では，正規のスクリプトであるかのように認識されて実行されてしまう．いうならば，開発者の意図しないスクリプトが実行されてしまうことが XSS の実態である．したがって，実行されようとしているスクリプトが開発者の意図したものであるか否かを判断し，開発者の意図したスクリプトに限定して Web ブラウザ上での実行を許可することによって，XSS を防止できる．具体的には，ホワイトリストによるスクリプトの実行制御がこれに該当する．

ホワイトリスト型対策は，ホワイトリストに定義されているスクリプト以外はすべてが否定されるので，頑健な XSS 対策といえる．実際，現在最も普及している XSS 対策の 1 つである Content Security Policy（CSP）[10] は，ポリシベースのホワイトリストに基づいた対策となっている．

CSP における主な XSS 対策は，外部スクリプトやスタイルシート（CSS）のコード署名を実現する Subresource Integrity（SRI）[5]，外部リソースの読み込み元ドメインの制限やインラインスクリプトの実行を禁止できる default-src [3]，script-src [4] である．この機能は，Web サーバの設定において HTTP レスポンスヘッダにスクリプトに関するセキュリティポリシを付加したうえで，Web アプリケーションの HTML ソースコードに宣言記述を行うことで有効化される．

2.3.1 Subresource Integrity（SRI）

外部スクリプトが攻撃者から改ざん可能であった場合，開発者の意図しないスクリプトの実行を許してしまい，XSS 攻撃へとつながる可能性がある．その改ざんを検知するための機能が SRI である．SRI を利用するには，外部スクリプトのハッシュ値を Web アプリケーションの HTML ソースコード内で指定する．ハッシュ値の一致しない外部スクリプトの実行が抑止される．

2.3.2 default-src，script-src

Web アプリケーション開発者は，自サイト以外の Web サ

イトで公開されている外部スクリプトを利用することも可能ではある．しかし，外部スクリプトを提供している Web サイトが攻撃者のものであった場合，攻撃者は自身の Web サイトにコード署名付きの外部スクリプトを配備することができてしまうため，SRI では対策できない．また，動的な Web アプリケーションでは，ユーザからの入力やデータベース中の登録内容に応じて HTML ソースコードが動的に変化するため，インラインスクリプトに対しては，SRI を利用したコード署名による真正性検査を適用できない．これらの問題に対する緩和策が default-src と script-src である．default-src，script-src の宣言記述によって，外部スクリプトの読み込み先となる Web サイト（ドメイン）を制限し，信頼できる Web サイト（典型的には自サイト）の外部スクリプトの実行のみを許可することが可能となる．また，インラインスクリプトの実行を禁止することが可能である．

2.3.3 問題点

開発者の意図しないスクリプトの実行の可能性は，有害なインラインスクリプトとして注入される場合と，有害な外部スクリプトとして注入される場合の 2 種類である．したがって，CSP の上述の機能を用いて，インラインスクリプトを禁止したうえで，信頼できる Web サイトの外部スクリプトをコード署名で確認することによって，XSS を防ぐことができる．しかし，インラインスクリプトは汎用性に富むプログラム要素であり，現在の Web サービスにおいてインラインスクリプトを利用していない Web サイトはきわめて少ない．本件について著者らも独自に Alexa ランキング [11] TOP 50 サイトを調査したところ，インラインスクリプトが使われていないサイトは皆無であった．したがって，サニタイジング処理に不備があった場合には，意図しないスクリプトがインラインで注入されて XSS を受ける恐れが依然として残っているという現状にある．

2.4 インラインスクリプトに対するホワイトリスト型 XSS 対策

インラインスクリプトを堅牢化する既存方式としては，ホワイトリスト（セキュリティポリシ）型の XSS 対策が提案されている．ここでは，BEEP [12] と ConScript [13] について説明する．また，ホワイトリストを経験的に改善する方法として，CSP の report-only モード [14] についても言及する．

2.4.1 BEEP，ConScript

Jim は，Web アプリケーションに記載されたセキュリティポリシに従って，Web ブラウザ上で JavaScript のコードを書き換えることにより，セキュリティポリシの遵守を強制する Browser Enforced Embedded Policies（BEEP）[12] を提案している．BEEP では，セキュリティ上重要な API についても，Web ブラウザからの API コールをフックしてその内容を検査することにより，セキュリティポリシに記述された API へのアクセスのみを許可するようになっている．Meyerovich らも，アスペクト指向プログラミング（Aspect Oriented Programming）を用いて，BEEP と同様の API コール検査手法である ConScript [13] を提案している．

2.4.2 CSP の report-only モード

CSP には report-only モード [14] とよばれる動作モードがある．report-only モードでは，ユーザが Web アプリケーションを利用する際に，CSP のセキュリティポリシが実際に適用されることはない．しかし，ポリシが適用されていたとしたらポリシ違反が起こっていたという場合には，Web アプリケーション開発者にそのレポートが送信されるようになっている．すなわち，report-only モードは，セキュリティポリシが Web アプリケーションに適用された際に，どのような結果が生じるのかを試行するための動作モードである．試行を通じて，ポリシに過不足があることが判明した場合には，開発者によって修正が行われる．

2.4.3 問題点

ポリシベースの XSS 対策においては，必要十分なセキュリティポリシを作成するための方法論が体系化されておらず，開発者が経験的あるいは暗黙知的にポリシを作成している．つまり，開発者に Web アプリケーションセキュリティに関する相応の知識が要求されることとなる．現在の Web アプリケーションは肥大化しており，その構造と動作は複雑であるため，十分なセキュリティ知識を有した開発者であっても，ポリシの設定にエラーが混入しやすくなる．このため，リリース後の試行を通じてポリシを改善していく（CSP の report-only モード）という経験的なアプローチによって，セキュリティポリシを改善していくという方法がとられている．セキュリティポリシ（ホワイトリスト）を理論的に作成する方法が形式知化されていないという課題は，既存のすべてのホワイトリスト型 XSS 対策が直面している大きな問題である．

3. 提案方式

3.1 CSP とテストベースホワイトリストの併用

2.3.3 項で述べたように，開発者の意図しないスクリプトの実行の可能性は，有害な外部スクリプトとして注入される場合と，有害なインラインスクリプトとして注入される場合の 2 種類ある．このうち，有害な外部スクリプトの注入については，2.3 節で示した CSP の機能を用いて，信頼できる Web サイトの外部スクリプトのみを許可（default-src，script-src）し，かつ，外部スクリプト自体が真正であることを検証（SRI のコード署名）するという方法を採用することによって，これを防ぐことができる．これに対し，有害なインラインスクリプトの注入については，2.3.3 項や 2.4.3 項で述べたように，その対策が難しい．そ

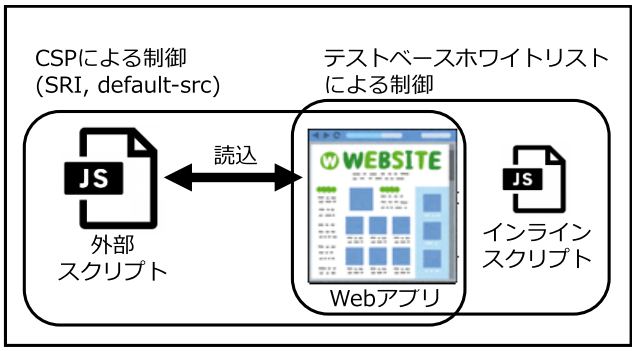


図 1 XSS 対策システムの全体像
Fig. 1 System overview of proposed XSS countermeasure.

こで本論文では，Web アプリケーションの開発工程の最終段階に実施されるソフトウェアテストの結果に基づいてホワイトリストを作成するアプローチを新たに提案し，その堅牢化を図る．

本論文で提案する XSS 対策システムの全体像を**図 1** に示す．外部スクリプトに対するホワイトリスト（コード署名）型対策である CSP と，インラインスクリプトに対するテストベースのホワイトリスト型対策を併用することによって，XSS を効果的に防ぐことが提案方式の特長である．提案方式において，外部スクリプトを守る CSP による XSS 対策については既存方式であるので，以降では，インラインスクリプトを守るテストベースホワイトリスト型 XSS 対策の部分に焦点を当て，方式の説明，実装，評価を行っていく．

3.2 テストベースホワイトリストを用いたインラインスクリプトの堅牢化

インラインスクリプトに対するホワイトリスト型 XSS 対策を堅牢化するために，Web アプリケーションの開発工程の最終段階で実施されるソフトウェアテストの結果に基づいてホワイトリストを作成するアプローチを提案する．テストベースホワイトリストにおいては，「Web アプリケーションのリリース前のテスト工程において確認されたスクリプトの動作」のみ本番環境での実行が許可される．具体的には，テスト工程で検証したスクリプトの動作をホワイトリストとして定義し，本番環境での Web アプリケーション使用時にホワイトリストに含まれるスクリプトの実行のみを許可する．

ソフトウェアテストは，通常，Web アプリケーションの開発工程の最終段階で（Web アプリケーションのリリース前に），Web アプリケーションの仕様書に基づいて行われる．したがって，テスト工程にホワイトリスト生成プロセスを統合することで，Web アプリケーションの開発プロセスを変更することなく，Web アプリケーションの仕様を逸脱することのないホワイトリストを自動生成することが可能となる．

仕様書の記載漏れ，あるいは，ソフトウェアテストの項目漏れが原因で，リリース前に確認をしておかなければならなかったスクリプトの動作のなかの一部がテストから漏れることが生じうる．しかし提案方式では，未テストの動作はホワイトリストに登録されないため，当該動作が本番環境で実行された時点でアラート（ホワイトリストからの逸脱）が発報される．このアラートがきっかけとなり，開発者はテスト漏れに気付くことができ，その時点で当該動作に対するテストをし直し，その結果を新たにホワイトリストへ追加することが可能である．

テストベースホワイトリストの特長を以下にまとめる．

(1) 本番環境でのスクリプトの実行が，テスト工程で確認された動作に対してのみ許可されている．
(2) テスト工程によって作成されたホワイトリストは，各 Web アプリケーションの仕様に基づいてソフトウェアテストが行われているため，仕様と一致する．
(3) テスト工程を通して，ホワイトリストを自動的に生成することが可能である．

3.3 テストと仕様書の連携

テストベースホワイトリストの重要なコンセプトは，「リリース前のソフトウェアテストの工程において確認された動作」のみをホワイトリストとして定義することにある．これは，理論的には必要かつ十分なホワイトリストとはいえないが，テストと仕様の連携によってヒューリスティックな意味で十分なホワイトリストを作成することができる．

XSS の原因は，Web アプリケーション開発者が意図しないスクリプトが実行されてしまうことにある．このような意図しないスクリプトの実行を禁止するためには，ソフトウェア開発の初期段階で作成される Web アプリケーションの仕様書を利用することが有効であると考えられる．仕様書には，実装されるべきすべての機能と，各機能の実行時にアプリケーションがどのように動作するかが示されている．すなわち，仕様書は「意図された動作の集合」であるといえる．そして，この仕様書は，Web アプリケーション開発の最終段階でも利用される．開発者は，Web アプリケーションをリリースする前に，自身が実装した Web アプリケーションが要求された仕様を満たしているか（仕様書に記載されているとおりの動作をするか）否かをテストする．このように，仕様書に示されているすべての動作がテスト工程で確認されることが期待されるため，テスト工程を通じて，意図されたすべての動作を含むホワイトリストを生成することが可能である．

ただし，ソフトウェアテストにおいて 100%のカバレッジを達成することは簡単な作業ではない．この問題を軽減するために，本論文では，「HTML ソースコード内のスクリプト以外の部分についてはワイルドカードとして扱う」という条件の下で，2 つの HTML ソースコードの文字列比

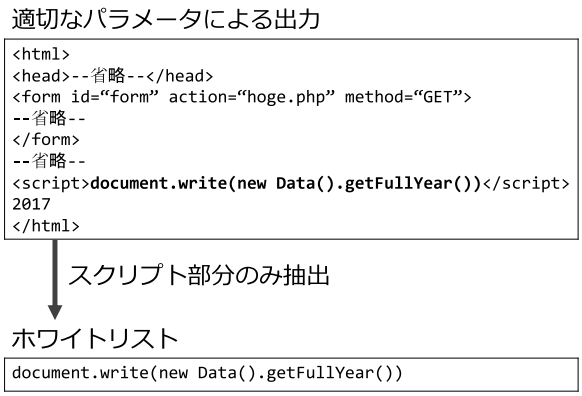


図 2 ホワイトリストの作成例
Fig. 2 Example of whitelist.

較を行うことによって両者の一致/不一致を検査する方法を採用する．前述のように，XSS の原因は，HTML ソースコードへの悪意あるスクリプトの注入である．したがって，XSS は，HTML ソースコード内のスクリプトの部分のみを比較することによって検知することが可能である．

ワイルドカード（HTML ソースコード内のスクリプト以外）の部分は比較の対象外であるため，今回は，HTML ソースコードからインラインスクリプト部分[*2]のみを抽出し（本論文では，これを「スクリプト構造」と呼ぶこととする），これをホワイトリストとして登録することとする．スクリプト構造は，典型的な入力値をテストするだけで確認できるため，ホワイトリストを作成するために入力値の網羅的なテストは必要でないと考えられる．さらに，これによりホワイトリストのデータサイズが縮小され，比較が容易になる．HTML ソースコードに対して生成されたテストベースホワイトリストの例を**図 2** に示す．

3.4 インラインスクリプトに対する XSS の検知

動的な Web アプリケーションでは，表示されるコンテンツはユーザからの入力や，データベース中の登録内容に応じて変化する．すなわち，これらのパラメータに従って，Web ページの HTML ソースコードの構造が変化する．テストベースホワイトリストでは，テスト工程において表示，実行された HTML ソースコードのスクリプト構造（インラインスクリプトの構造）がすべてホワイトリストとして登録されている．したがって，ユーザによって入力されたパラメータが Web アプリケーションの仕様内にある限り（意図されたパラメータが入力される限り），Web アプリケーションによって生成される HTML ソースコードは，ホワイトリストに登録されているスクリプト構造から逸脱することはない．

一方，XSS 脆弱性のある Web アプリケーションにおいて，開発者が想定していないパラメータが攻撃者によって入力されると，Web アプリケーションは，開発者の意図しないスクリプトが注入された HTML ソースコードを生成する．その結果，Web アプリケーションによって生成された HTML ソースコードのスクリプト構造が，ホワイトリストに登録されていない構造に変更される．テストベースホワイトリストによる XSS 対策では，この特徴を用いて XSS を検知する．この特徴は，Reflected XSS だけでなく，Stored XSS や DOM-based XSS などすべての XSS 攻撃で見られるため，同様の手法で攻撃の検知が可能である．

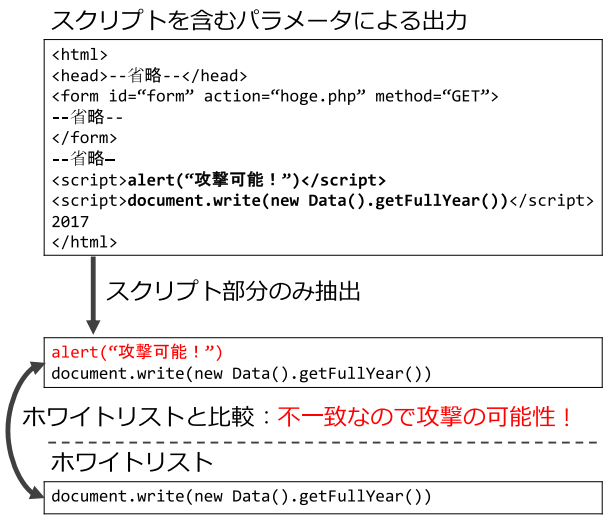


図 3 XSS の可能性がある場合の例
Fig. 3 Example of XSS attack.

テストベースホワイトリストを用いたインラインスクリプトに関する XSS 検知機能は，ユーザの Web ブラウザ上で実行される．ユーザが Web サービスにアクセスしてその Web アプリケーションを利用するたびに，Web ブラウザ内で，Web アプリケーションによって生成された HTML ソースコードのスクリプト構造と，ホワイトリストに登録されている HTML ソースコードのスクリプト構造がつねに比較される．両方の構造が一致しない場合，XSS が発生していると判断できる．**図 3** に XSS の実例を示す．図 2 の Web ページに「`<script>alert("攻撃可能！")</script>`」というパラメータを入力することにより，図 3 に示す意図しないインラインスクリプトを含んだ HTML ソースコードが生成される．図 2 のホワイトリストと図 3 のスクリプト構造が異なるため，これが XSS の発生として検知され，当該スクリプトの実行が禁止される．

3.5 ホワイトリストの自動生成

テストベースのホワイトリストは，仕様書に示されたパラメータ（開発者によって意図されたパラメータ）が入力されたときに，Web アプリケーションが生成する HTML ソースコード内のインラインスクリプトの構造を抽出することで生成される．これは，ホワイトリスト生成プロセス

[*2] 今回は，Script タグ内の JavaScript，W3C で定義されているマウスイベント内の JavaScript，および，javascript:で始まるリンクの JavaScript を抽出して，それらの構造をホワイトリストとして登録している．

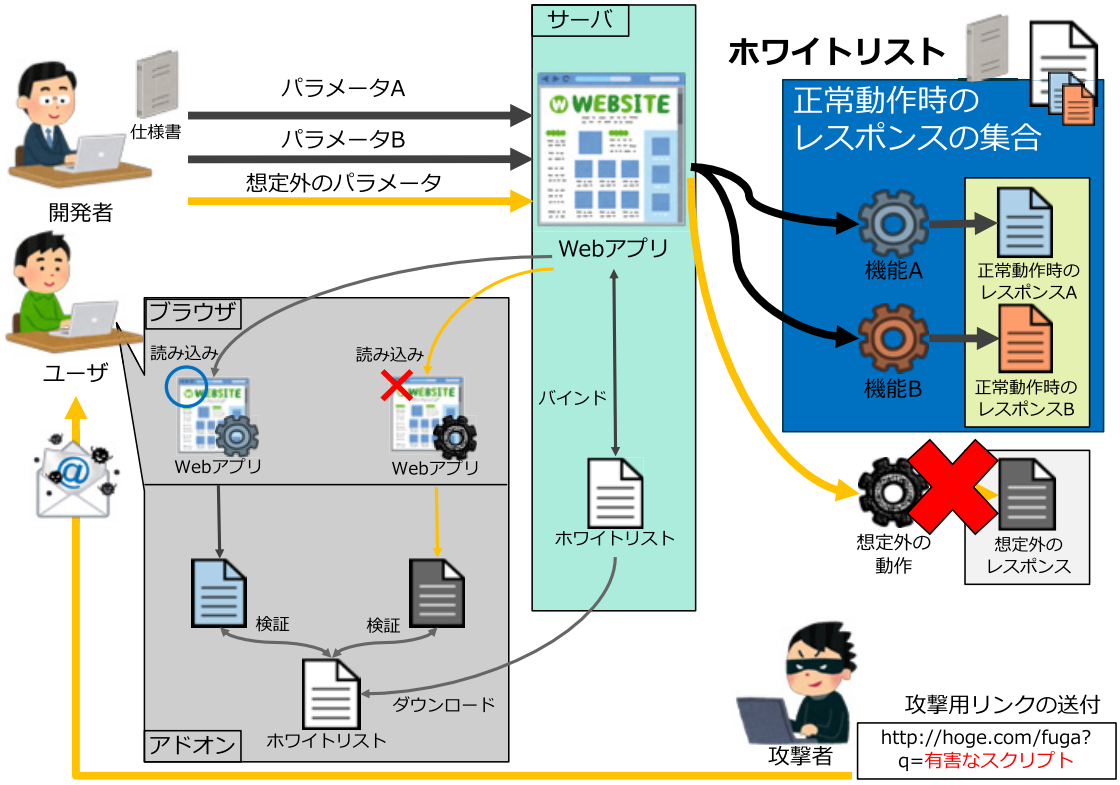


図 4 テストベースホワイトリスト型 XSS 対策の実装
Fig. 4 Inmplementation of examination-based whitelist against XSS.

が，Web アプリケーションのソフトウェアテストの工程と非常に類似していることを意味する．

テスト工程の目的は，Web アプリケーションが仕様に従って実装されているかどうかを確認することである．Web アプリケーションをリリースするとき，開発者は，Web アプリケーションがパラメータテストを含めて，仕様に従って動作しているかどうかを検証する動作テストを行う．このパラメータテストでは，仕様書に示されているいくつかのパラメータのパターンを入力し，Web アプリケーションが正しく応答するかどうかを検証する．

ここで，ソフトウェアテストは，エラーとなる入力に対するテストも行われることに注意されたい．そして，エラーとなる入力に対しても XSS が成立する場合があるため，提案方式におけるテストも，正常な入力に対するテストと，エラーとなる入力に対するテストの両方が対象となる．提案方式では，開発者が想定する範囲内の入力に対しては，正常な場合もエラーの場合も，その結果がホワイトリストに登録される．これによって，Web アプリケーションにエラーとなるパラメータ（想定内のエラー）が入力された場合には，開発者が想定した形でエラーページが表示されることが保証され，それ以外の場合には，ホワイトリストから逸脱によって XSS として検知がなされる．

このように，テストベースのホワイトリストを生成するために必要なすべての HTML ソースコードは，Web アプリケーションのテスト工程において取得することができる．なお，Web アプリケーションのソフトウェアテストは，一般的に，Web アプリケーションの各 Web ページに対して実施されることに鑑み，提案方式のホワイトリスト（スクリプト構造）は，Web アプリケーションによって生成される Web ページ単位で生成することとする．

4. 実装

本章では，インラインスクリプトを守るテストベースホワイトリスト型 XSS 対策の実装（図 4）部分を説明する．3.4 節で述べたように，インラインスクリプトに対する XSS 検知はユーザの Web ブラウザ上で実行される．これを実現するためには，次の 3 つの機能の実装が必要となる．1 つ目に，Web アプリケーションのソフトウェアテストを通じてホワイトリストを自動生成する機能，2 つ目に，Web アプリケーションとホワイトリストの関連付けを行う機能，3 つ目に，Web ブラウザ内で実行されるインラインスクリプトをホワイトリストによって検査する機能である．

4.1 Selenium によるホワイトリスト自動生成

1 つ目の「Web アプリケーションのソフトウェアテストを通じてホワイトリストを自動生成する機能」については，Selenium [15] と呼ばれる Web アプリケーション用テストツールを用いて実装した．Selenium は，Web ブラウザのオートメーションツールであり，Web アプリケーションにおけるソフトウェアテストの自動化に広く利用されている．

Seleniumでは，テスト実施者が「ユーザからの入力（入力データ）」と「その入力に対して生成されるべきWebページ（教師データ）」のペアから成る検査項目をテストケースとして用意する．Seleniumは，テストケース中のすべての検査項目に対し，「Webアプリケーションに入力データを入力 → その結果生成されたWebページを教師データと比較」というテスト手順を繰り返すことによって，ソフトウェアの動作を自動検査する．

Seleniumは，テスト中に得られる任意のデータを他の処理へ利用することも容易である．そこで今回の実装では，Seleniumにスクリプト抽出機能をアドオンし，テスト中に得られるWebページのHTMLソースコードをスクリプト抽出アドオンに入力することによって，テスト工程におけるホワイトリストの自動生成を達成する．

スクリプト抽出アドオンが実行する具体的な処理は，以下の3つである．

(1) Seleniumによるテスト結果が「開発者が想定したとおりのWebページ」であった場合のみ，当該検査項目のWebページのHTMLソースコードを取得する．
(2) HTMLソースコードからインラインスクリプトを抽出する．今回は正規表現を利用し，Scriptタグ内のJavaScript，W3Cで定義されているマウスイベント内のJavaScript，javascript:で始まるリンクのJavaScriptを抽出する[*3]．
(3) そのスクリプト構造をホワイトリストに追加して保存する．ホワイトリストは，Webアプリケーションによって生成されるWebページ単位で保存される．

4.2 Webアプリケーションとホワイトリストのバインド

2つ目の「Webアプリケーションとホワイトリストの関連付けを行う機能」については，今回はWebExtensions API [16] を利用し，Firefox用のブラウザアドオンとして実装した．具体的には，「arbitrary_page.php」というWebアプリケーションファイルが存在しているフォルダの同階層に，arbitrary_page.phpのホワイトリストを「arbitrary_page.php.wl」というファイル名で配置するとともに，「arbitrary_page.phpへのアクセス時には同時にarbitrary_page.php.wlをダウンロードする」というアドオンをWebブラウザに加えることとした[*4]．

4.3 検査用ブラウザアドオン

3つ目の「Webブラウザ内で実行されるインラインスクリプトをホワイトリストによって検査する機能」については，今回は，4.2節での実装と同様に，Firefox用のブラウザアドオンとして実装した．アドオンの具体的な動作は，3.4節（図2，図3）で説明したとおりであり，Webアプリケーションにより生成されたWebページのスクリプト構造がホワイトリスト中に存在しなかった場合には，XSS攻撃の可能性を知らせる検知メッセージをWebブラウザに表示し，スクリプトの実行を禁止する．なお，前節のアドオンも，今回は，この検査用ブラウザアドオンの中の一機能として実装した．

5. 評価

5.1 実験1：ホワイトリスト自動生成実験

5.1.1 実験目的

4.1節で実装したホワイトリスト自動生成機能をアドオンしたSeleniumを用い，Webアプリケーションに対するテスト工程でホワイトリストが正しく自動生成されることを確認する．

5.1.2 対象Webアプリケーション

著者らが独自に入手したWebアプリケーションの仕様書を参考にして，PHP言語でWebアプリケーションを実装した．これを，今回のテスト対象のWebアプリケーションとして用いる．

今回の対象Webアプリケーションは，企業などにおける備品管理用データベースシステムである．当該アプリケーションは，登録されたユーザIDとパスワードでログインすることで，備品の機種情報，価格情報，営業情報，ユーザ情報に関するデータベース管理（登録・変更，削除）を，Webブラウザから行うことができる．Webアプリケーションは処理ごとにプログラムが分離されており，24個のPHPプログラムとそれにともなって表示される24個のWebページから構成される（**図5**）．

ページの制約上，ここでは，「価格情報管理メニュー」を例にとり，Webアプリケーションの仕様とテストケースを説明するが，実際のテストは当該Webアプリケーションの全メニューに対して実施している．

価格情報管理メニューでは，データベースに対する備品の「登録・変更」および「削除」の機能が実装されている．各機能における仕様は以下のとおりである．

【登録・変更機能】

（仕様1） 価格情報登録・変更ページにおいて，入力フォームに登録内容を入力し，「登録する」ボタンを押すと，価格情報登録・変更処理ページへのページ遷移が発生する．

[*3] 本論文の実装においては，実装コストとのバランスに鑑み，正規表現を用いてJavaScriptを抽出する方法を選択したが，提案方式が実用に供せられる段階においては，WebブラウザのHTML構文解析パーサを採用する予定である．これにより，HTMLソースコード内のJavaScriptを確実に抽出することができる．

[*4] 本論文の実装においては，実装コストとのバランスに鑑み，WEBページとホワイトリストのバインド機能をブラウザアドオンによって実装したが，提案方式が実用に供せられる段階においては，CSS（カスケーディングスタイルシート）を採用する予定である．これにより，ユーザがWebアプリケーションにアクセスした際には，WebブラウザがWebページのHTMLソースコードに記されているCSSを解釈し，当該Webページのホワイトリストを同時にダウンロードすることができる．

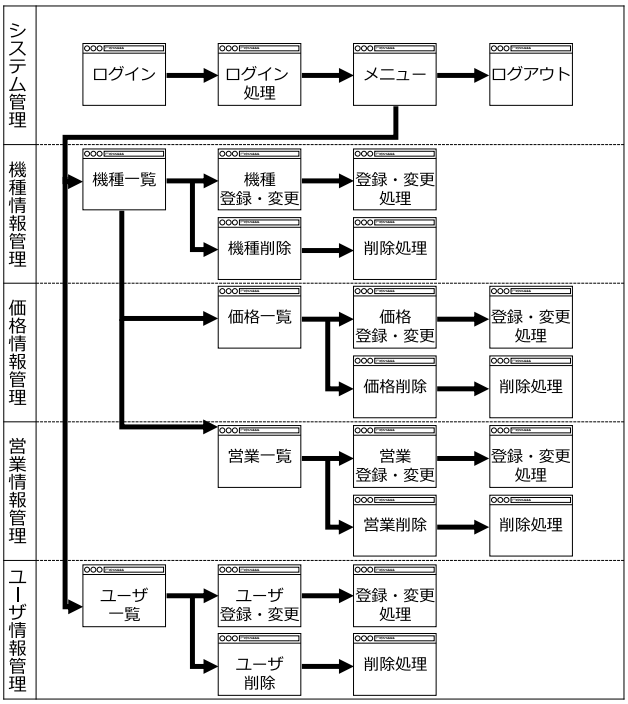


図 5 対象 Web アプリケーションのページ構成
Fig. 5 Webpage structure of experimental web application.

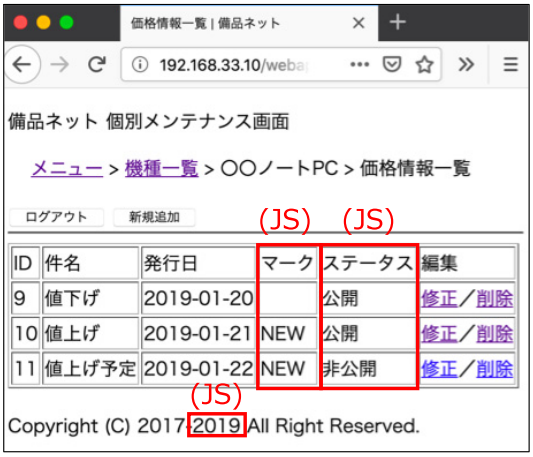


図 6 価格情報一覧ページ
Fig. 6 A page listed price information.

（仕様 2） 価格情報登録・変更処理ページにおいて，「登録が完了しました.」と表示される．
（仕様 3） 価格情報登録・変更処理ページにおいて，「価格情報一覧画面へ戻る」リンクをクリックすると，価格情報一覧ページへのページ遷移が発生する．
（仕様 4） 価格情報一覧ページにおいて，登録もしくは変更した内容が反映された項目を含め，データベースに登録されている項目がすべて表示される．

【削除機能】
（仕様 1） 価格情報一覧ページにおいて，削除したい項目の「削除」リンクをクリックすると，価格情報削除ページへのページ遷移が発生する．
（仕様 2） 価格情報削除ページにおいて，「以下の内容の価格情報を削除します.」と表示される．
（仕様 3） 価格情報削除ページにおいて，「削除する」ボタンをクリックすると，価格情報削除処理ページへのページ遷移が発生する．
（仕様 4） 価格情報削除処理ページにおいて，「削除が完了しました.」と表示される．
（仕様 5） 価格情報削除処理ページにおいて，「価格情報一覧画面へ戻る」リンクをクリックすると，価格情報一覧ページへのページ遷移が発生する．
（仕様 6） 価格情報一覧ページにおいて，削除項目を除いたデータベースに登録されている項目がすべて表示される．

価格情報一覧ページにおいては，ページ下部に Copyright 情報が表示されるようになっている．この Copyright 表示のために，Web アプリケーションは現在の西暦年 (2019) を取得する．また，データベースの内容と比較することによって，新規登録である場合には Web ページ中に「NEW マーク」を表示する．さらに，データベースに記載されているステータス情報に応じて，Web ページ中に「公開」あるいは「非公開」の文字列を表示する．以上の処理を実行する部分に JavaScript が利用されている．価格情報一覧ページ上のこれらの場所を，**図 6** 中の赤文字の「(JS)」を付した赤枠によって示した．価格情報一覧ページ以外の Web ページにおいても，それぞれのページにおいて必要な JavaScript が利用されている．これらのインラインスクリプトが XSS 脆弱性の潜在箇所となる．

5.1.3 テスト工程

ソフトウェアテストの目的は，Web アプリケーションが仕様に従って実装されているかを確認することである．したがって，今回のテストケースは，5.1.2 項に記した「登録・変更機能の仕様 1〜4 に対する検査項目，ならびに，「削除機能」の仕様 1〜6 に対する検査項目から成る．今回は，各機能の検査項目を機能ごとの一連のテスト手順としてテストケースを記述した．具体的なテスト手順は，以下のとおりである．また，テストにおける画面遷移の状態を**図 7** に示す．

【登録・変更機能】
(1) Web ブラウザを起動
(2) 図 7 上「価格情報登録・変更ページ」の表示
(3) 図 7 上「価格情報登録・変更ページ」の表示を検証
(4) 図 7 上「価格情報登録・変更ページ」の入力フォームに登録備品の価格情報を入力
(5) 図 7 上「価格情報登録・変更ページ」の「登録する」ボタンを押下（図 7 上「価格情報登録・変更処理ページ」に遷移）
(6) 図 7 上「価格情報登録・変更処理ページ」の表示を検証

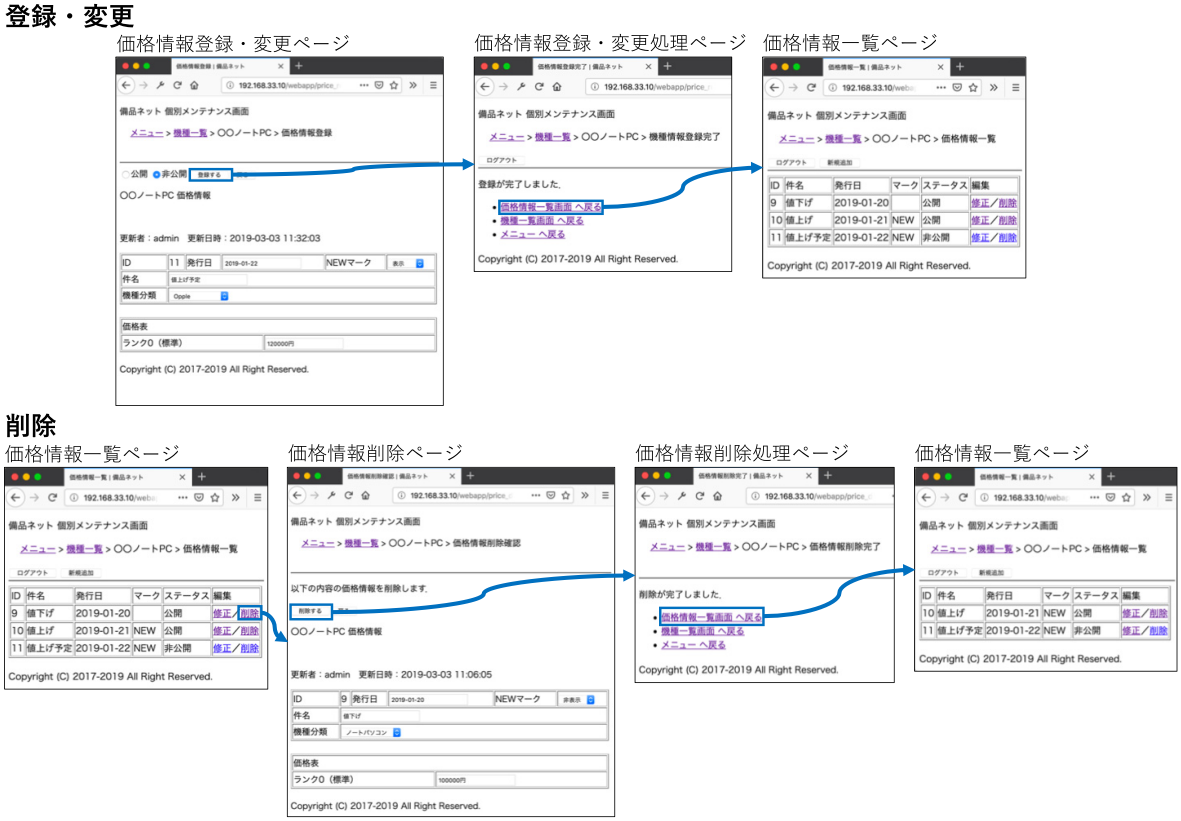


図 7　動作時における Web ページの画面遷移
Fig. 7　Webpage transitions.

(7) 図 7 上「価格情報登録・変更処理ページ」の「価格情報一覧画面へ戻る」リンクを押下（図 7 上「価格情報一覧ページ」に遷移）
(8) 図 7 上「価格情報一覧ページ」が表示され，手順 (4) で登録したデータが表示されていることを検証
(9) テスト終了

【削除機能】
(1) Web ブラウザを起動
(2) 図 7 下「価格情報一覧ページ」の表示
(3) 図 7 下「価格情報一覧ページ」の表示を検証
(4) 図 7 下「価格情報一覧ページ」の任意の項目の「削除」リンクを押下（図 7 下「価格情報削除ページ」に遷移）
(5) 図 7 下「価格情報削除ページ」の表示を検証
(6) 図 7 下「価格情報削除ページ」の「削除する」ボタンを押下（図 7 下「価格情報削除処理ページ」に遷移）
(7) 図 7 下「価格情報削除処理ページ」の表示を検証
(8) 図 7 下「価格情報削除処理ページ」の「価格情報一覧画面へ戻る」リンクを押下（図 7 下「価格情報一覧ページ」に遷移）
(9) 図 7 下「価格情報一覧ページ」が表示され，手順 (4) で「削除」リンクを押下したデータが表示されていないことを検証
(10) テスト終了

5.1.4　実験結果

4.1 節で説明したホワイトリスト自動生成機能を実装した Selenium を用い，5.1.3 項に示したソフトウェアテストを実施した結果，「開発者が想定したとおりの Web ページ」に対するホワイトリストが正しく自動生成されたことが確認できた．なお，ソフトウェアテストは，実際には対象 Web アプリケーションの全機能に対して行っている．今回の対象 Web アプリケーションは，入力に応じて 24 ページの WEB ページが生成される．3.5 節および 4.1 節で説明したとおり，提案方式のホワイトリストは Web ページ単位で生成されるため，24 ページ分のホワイトリストがすべて正しく生成されていることが確認できた．ただし，生成されるホワイトリストの汎用性については懸念が生じることが判明したため，この点については 6.2 節にて考察する．

5.2　実験 2：検知実験

5.2.1　実験目的

4.1 節で実装した検査用ブラウザアドオンが，インラインスクリプトに対する XSS 攻撃を検知できることを確認する．

5.2.2　実験方法

検知実験は，実験用 PC 上に Web サーバ役の仮想マシンを VirtualBox で構築して実験を行った．実験用 PC の諸元を**表 1** に，仮想マシンの諸元を**表 2** に，それぞれ示

表 1 実験用 PC の緒元
Table 1 Experimental PC specifications.

ホスト	ホスト OS	macOS Mojave 10.14.3
	メインメモリ	16,384MB（16GB）
	Web ブラウザ	Firefox Quantum 64.0

表 2 仮想マシンの緒元
Table 2 Virtual Machine specifications.

仮想マシン	仮想化ソフト	VirtualBox 5.2.22 r126460 (Qt5.6.3)
	構成管理	Vagrant 2.2.3
	ゲスト OS	Ubuntu 16.04 LTS
	メインメモリ	1,024MB（1GB）
	WEB サーバ	Apache 2.4.18
	サーバサイド言語	PHP 7.1
	ネットワーク速度	1Gbps

す．仮想マシン上の Web サーバに 5.1.2 項の備品管理用 Web アプリケーションと 5.1.4 項のホワイトリストを配置して，実験用 PC 上の Web ブラウザから当該 Web アプリケーションにアクセスする．

攻撃コードに関しては，OWASP Foundation [17] により提供されている XSS フィルタ回避チートシート [18] に公開されている全 68 個の XSS 攻撃コードを収集した．これらのうち，54 個については，攻撃コードとして動作しなかった[*5]．したがって，今回の実験環境で XSS が実際に発動した残りの 14 個の攻撃コードを用いて実験を行った．実験用 PC 上の Web ブラウザから，これらの攻撃コードを Web アプリケーションに注入した．

5.2.3 実験結果

14 個の攻撃コード中，13 個の攻撃コードが検査用ブラウザアドオンによって検知されたことが確認できた．1 個の攻撃コードについては検知に失敗した．その原因については 6.3 節にて考察する．

5.3 実験 3：誤検知実験

5.3.1 実験目的

正規ユーザが開発者の意図したとおりの Web アプリケーション利用を行った際に，検査用ブラウザアドオンが XSS を誤検知することがないことを確認する．

5.3.2 実験方法

「開発者の意図したとおりの Web アプリケーション利用」とは，仕様書に準拠した Web アプリケーションの利用を意味する．そこで今回は，仕様書の内容から作成された 5.1.3 項のテストケースを誤検知実験のための入力として利用することとした．このテストケースを，5.2 節の検知実験において実行し，検査用ブラウザアドオンが XSS として誤検知することがないか確かめる．

5.3.3 実験結果

すべてのテストケースの実行に対し，検査用ブラウザアドオンが XSS として誤検知することはなかった．

5.4 実験 4：パフォーマンス実験

5.4.1 実験目的

テストベースのホワイトリストにおいては，ソフトウェアテストの際に実施された検査項目の数に応じて，ホワイトリストのボリュームが増加することになる．そして，このホワイトリストは，ユーザが Web アプリケーションを利用する際に，ユーザの Web ブラウザにダウンロードされる．したがって，ユーザが Web アプリケーションを利用する際のパフォーマンス低下が懸念される．本実験では，ホワイトリストのダウンロードとホワイトリストを用いたスクリプト構造の検査に関するオーバヘッドを確認する．

5.4.2 実験方法

実験環境は表 1，表 2 と同一である．比較的大きなインラインスクリプト（JavaScript，272 KB，10,365 行）を含む Web アプリケーションを準備し，そのホワイトリストとともに仮想マシン上に Web サーバに配置した．なお，3.3 節で示したとおり，提案方式では，「スクリプト以外の部分をワイルドカードとした HTML ソースコード」がホワイトリストとなるため，ホワイトリストのサイズはインラインスクリプトのサイズと等しい．この Web アプリケーションへのリクエストから Web ブラウザのコンテンツレンダリング完了までの所要時間を Firefox 標準ツールで計測する．テストベースホワイトリストを実装した Web プラットフォーム（検査あり）との速度比較のために，テストベースホワイトリストを導入していない従来の Web プラットフォーム（検査なし）においても，同様の計測を行った．ブラウザ計測は，検査あり，検査なしのそれぞれで 5 回ずつ，計 10 回行った．毎回の計測ごとに，Web ブラウザのキャッシュ，Cookie，履歴など計測結果に影響を与えるデータは逐一消去した．

5.4.3 実験結果

検査ありの実験結果を**表 3** に，検査なしの実験結果を**表 4** に示す．計測値は，(1) DOM Content Loaded[*6]，(2) Load[*7]，(3) レンダリング完了のそれぞれの時点までのラップタイムである．検査ありの所要時間は，検査なしの所要時間より 1.3 倍〜1.6 倍のオーバヘッドが生じる結果となった．この妥当性については 6.4 節で考察する．

[*5] その理由については詳細に特定できなかったが，攻撃コードとして不完全であったため，あるいは，今回利用した Web ブラウザにおいてはすでに対策パッチが適用されていたため，などの理由が考えられる．

[*6] DOM ツリーの読み込み後の DOM 解析完了イベント

[*7] DOM ツリーの構築完了後の外部リソース（画像など）の読み込み完了イベント

表 3 パフォーマンス実験結果（検査あり）
Table 3 Result of performance experiment (with inspection).

	検査あり（単位：msec）		
	(1)	(2)	(3)
1 回目	200	209	214
2 回目	193	201	232
3 回目	179	189	205
4 回目	180	191	219
5 回目	190	200	232

表 4 パフォーマンス実験結果（検査なし）
Table 4 Result of performance experiment (without inspection).

	検査なし（単位：msec）		
	(1)	(2)	(3)
1 回目	140	150	161
2 回目	137	147	151
3 回目	147	160	162
4 回目	139	150	163
5 回目	137	148	150

6. 考察

6.1 テストベースホワイトリストの有効範囲

今回の実装では，Web ページのスクリプト構造（スクリプト以外の部分をワイルドカードとした HTML ソースコード）を比較することによってホワイトリストとの検査が行われる．したがって，「XSS が成立した場合の Web ページのスクリプト構造」が「ホワイトリストに含まれているスクリプト構造（開発者が想定したパラメータを Web アプリケーションに入力した場合に生成される Web ページのスクリプト構造）」のいずれかに一致してしまった場合は，検知漏れとなる[*8]．

今回の評価実験に用いた Web アプリケーションは，「仕様書に基づいて開発され，ソフトウェアテストを経てリリースされる Web アプリケーション」の簡素な一具体例である．今後，大規模あるいは複雑な Web アプリケーションにおいても評価を行う必要がある．また，世の中には綿密な仕様書なしで開発される Web アプリケーションや十分なソフトウェアテストなしでリリースされる Web アプリケーションも存在する．それらに対しては提案方式の効果は限定的なものとなる．

[*8] ただし，スクリプト構造が一致するということは，HTML ソースコード内のスクリプトの出現位置だけでなく，スクリプトの種類も一致することを意味する．XSS の原因は HTML ソースコードへの悪意あるスクリプトの注入であるため，「XSS が成立した場合の Web ページのスクリプト構造」が「ホワイトリストに含まれているスクリプト構造」と偶然に一致してしまうケースは限定的であるのではないかと推測している．

```
<EMBED SRC="
Wxuczpzdmc9Imh0dH A6Ly93d3cudzMub3JnLzIwMDAvc
3ZnIiB4bWxucz0iaHR0cDovL3d3dy53My5vcmcv MjAwM
C9zdmciIHhtbG5zOnhsaW5rPSJodHRwOi8vd3d3LnczLm9
yZy8xOTk5L3hs aW5rIiB2ZXJzaW9uPSIxLjAi [illegible] BpZD0i
eHNzIj48c2NyaXB0IHR5cGU9InRleHQvZWNtYXNjcmlw
dCI+YWxlcnQoIlh TUyIpOzwvc2NyaXB0Pjwvc3ZnPg=="
type="image/svg+xml" AllowScriptAccess="always"></E
MBED>
```

図 8 検知できなかった攻撃コード
Fig. 8 Undetectable attack code.

```
<svg xmlns:svg="http://www.w3.org/2000/svg" xmlns="htt
p://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org
/1999/xlink" version="1.0" [illegible] id="xss"><script type="text/ecmascript">alert("
XSS");</script></svg>
```

図 9 BASE64 部分のデコード結果
Fig. 9 Decoding result of BASE64 sentences.

6.2 スクリプト構造の正規表現によるホワイトリスト

5.1 節で扱った Web アプリケーションの「価格情報一覧ページ」では，データベースに保存されている登録備品数に応じて表示される備品一覧の行が増えていく．しかし．ソフトウェアテストの工程においては，登録備品数が 0 の場合，1 の場合，2 の場合，…という形での総当たりでの検査は行わず，たとえば代表値を用いての確認が行わることが一般的である．これでは，テストユーザが当該 Web アプリケーションを利用する際に表示されうるすべての Web ページに対応するホワイトリストを，ソフトウェアテストを通じて取得することができない．この問題に対処するためには，ホワイトリストのスクリプト構造を正規表現で記述するなどの工夫が必要となる．本論文においては，この部分の実装については今後の課題とする．

6.3 スクリプト難読化に対応したホワイトリスト

5.2 節の検知実験において，検知に失敗した攻撃コードを図 8 に示す．このコードは，一見，BASE64 エンコードされた SVG 画像を表示させようとしているコードに見える．しかし，このコードをデコードすることによって得られる実際の攻撃コードは図 9 のようになる．今回，4.3 節で実装した検査用ブラウザアドオンにおいては，正規表現によりスクリプト部分を抜き出す方式を採用していた．このため，図 8 のようなスクリプトエンコーディングを見逃す結果となった．この問題に対処するためには，Web ブラウザで使用されている構文解析パーサを利用するなどの方

表 5 インラインスクリプトサイズの大きい Web サイト
Table 5 Inline script sizes in popular websites.

サイト名	容量	行数
Facebook	578KB	16
Amazon.com	312KB	577
Yahoo!	383KB	1384

法によって，インラインスクリプトが難読化されていたとしても，これを確実に発見することが必要である．本論文においては，この部分の実装については今後の課題とする．

6.4 パフォーマンス実験の妥当性

テストベースのホワイトリストを用いた XSS 検知は，インラインスクリプトの複雑さに応じて，ホワイトリストの大きさが肥大化する．5.4 節においては，インラインスクリプトの複雑さとサイズの間にはある程度の正の相関があるという前提に立ち，比較的大きな JavaScript（272 KB，10,365 行）をインラインスクリプトとして含む Web アプリケーションを使用してのパフォーマンス実験を行った．これに対し，著者らが，Alexa ランキング [11] の TOP 50 サイトを対象として，Web サイト内で使用されているインラインスクリプトのサイズを調査したところ，5.4 節の実験で用いたインラインスクリプトよりも大きな容量のインラインスクリプトが利用されていた Web サイトは，3 サイトのみであった（**表 5**）．したがって，テストベースホワイトリストを実装した Web プラットフォームが運用されることになった場合も，世の中の多くの Web サイトのホワイトリストのサイズは，5.4 節の実験で使用した Web アプリケーションのホワイトリストサイズよりも同程度以下となるものと見込まれる．この観点から，5.4 節で行ったパフォーマンス実験において利用したホワイトリストサイズは妥当であると考えている．

7. おわりに

本論文では，コード署名による対策が難しくサニタイジングの不備の影響を大きく受けるインラインスクリプトに対する XSS 対策として，テストベースホワイトリストを提案した．そして，外部スクリプトに対するコード署名ベースのホワイトリスト型対策である CSP と，インラインスクリプトに対するテストベースのホワイトリスト型対策を併用することによって，効果的な XSS 対策を実現した．

テストベースホワイトリストは，XSS 検知用ホワイトリストを自動的に生成するための，「テストベース」のアプローチによる XSS 対策である．本方式では，テスト工程で検証されるスクリプト構造に焦点を当て，ホワイトリストを定義している．テスト工程で事前に確認されたスクリプトのみが実行を許される点が，完全性の観点からのテストベースホワイトリストの特長である．各 Web アプリケーションの仕様書に基づいてソフトウェアテストを行うため仕様書に一致するホワイトリストを自動的に生成することが可能である点が，健全性の観点からのテストベースホワイトリストの特長である．

今後は，スクリプト構造の正規表現によるホワイトリストの検討やスクリプト難読化に対応したホワイトリストの検討，またそれらの有効性評価を含め，本手法を検証および改良していく．また，CSP とテストベースホワイトリストを併用した XSS 検知システムの全体を対象とした実証実験を行っていく．

参考文献

[1] 安全なウェブサイトの作り方改訂第 7 版，入手先 ⟨https://www.ipa.go.jp/files/000017316.pdf⟩（参照 2017-12-18）．
[2] コンテンツセキュリティポリシー（CSP）- HTTP | MDN，入手先 ⟨https:// developer.mozilla.org/ja/docs/Web/HTTP/CSP⟩（参照 2019-02-01）．
[3] CSP: default-src - HTTP | MDN，入手先 ⟨https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/Content-Security-Policy/default-src⟩（参照 2019-02-01）．
[4] CSP: script-src - HTTP | MDN，入手先 ⟨https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/Content-Security-Policy/script-src⟩（参照 2019-02-01）．
[5] サブリソース完全性 - ウェブセキュリティ | MDN，入手先 ⟨https://developer.mozilla.org/ja/docs/Web/Security/Subresource_Integrity⟩（参照 2019-02-01）．
[6] 井上佳祐，本多俊貴，向山浩平，大木哲史，西垣正勝：ホワイトリスト型 XSS 攻撃検知の効果的な実現方法に関する検討，2018 年暗号と情報セキュリティシンポジウム，SCIS2018，pp.1–7 (2018).
[7] 井上佳祐，本多俊貴，向山浩平，大木哲史，西垣正勝：XSS 攻撃検知のためのテストベースホワイトリスト自動生成手法に関する検討，コンピュータセキュリティシンポジウム 2018，CSS2018，pp.1–8 (2018).
[8] 井上佳祐，本多俊貴，向山浩平，大木哲史，堀川博史，西垣正勝：テストベースホワイトリストと CSP の組み合わせによる効果的な XSS 攻撃対策の実現に関する検討，IPSJ SIG Technical Reports, 2019-CSEC-84-6, pp.1–8 (2019).
[9] 「マウスオーバーの」問題についての全容，入手先 ⟨https://blog.twitter.com/official/ja_jp/a/ja/2010-26.html⟩（参照 2017-12-09）．
[10] Stamn, S., Sterne, B. and Markham, G.: Reining in the web with content security policy, *Proc. 19th international conference on World Wide Web*, pp.921–930, ACM (2010).
[11] Alexa Top 500 Global Sites, available from ⟨https://www.alexa.com/topsites⟩ (accessed 2019-01-09).
[12] Jim, T.: Defeating script injection attacks with browser-enforced embedded policies, *Proc. 16th international conference on World Wide Web*, pp.601–610, ACM (2007).
[13] Meyerovich, M. and Livshits, B.: Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser, *IEEE Symposium on Security and Privacy*, pp.481–496, IEEE (2010).
[14] Content-Security-Policy-Report-Only - HTTP | MDN, available from ⟨https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/Content-Security-Policy-Report-Only⟩ (accessed 2019-02-01).
[15] Selenium - Web Browser Automation, available from

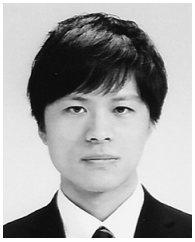
⟨https://www.seleniumhq.org/⟩ (accessed 2017-12-05).
[16] ブラウザ拡張機能 – Mozilla | MDN，入手先 ⟨https://developer.mozilla.org/ja/docs/Mozilla/Add-ons/WebExtensions⟩（参照 2019-02-01）.
[17] OWASP, available from ⟨https://owasp.org/⟩ (accessed 2019-02-01).
[18] XSS Filter Evasion Cheat Sheet - OWASP, available from ⟨https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet⟩ (accessed 2019-02-01).

井上 佳祐

2017 年加計学園岡山理科大学総合情報学部情報科学科卒業．2019 年静岡大学大学院総合科学技術研究科修士課程修了．同年株式会社ラック入社．情報処理安全確保支援士（第 019097 号）．在学中，Web アプリケーションセキュリティに関する研究に従事．

本多 俊貴

2017 年静岡大学情報学部情報科学科卒業．2019 年同大学院修士課程修了．在学中，ランサムウェア検知に関する研究に従事．



向山 浩平

2016 年静岡大学情報学部情報科学科卒業．2018 年同大学院修士 課程修了．同年東海旅客鉄道株式会社入社．在学中，ユーザビリティと Web セキュリティに関する研究に従事．

大木 哲史 （正会員）

2002 年早稲田大学理工学部電子情報通信学科卒業．2004 年同大学大学院理工学研究科電子・情報通信学専攻修士課程修了．2010 年早稲田大学理工学術院情報・ネットワーク専攻博士（工学）取得．2010 年早稲田大学理工学総合研究所次席研究員，2013 年産業技術総合研究所特別研究員を経て 2017 年より静岡大学大学院総合科学技術研究科講師．情報セキュリティ全般，特に個人認証を中心としたネットワークセキュリティに関する研究に従事．情報処理学会，電子情報通信学会各会員．

堀川 博史 （正会員）

1978 年名古屋工業大学情報工学科卒業．1980 年北海道大学工学研究科情報工学科修了．同年三菱電機株式会社入社．2017 年静岡大学創造科学技術大学院博士課程了．博士（情報学）．現在，三菱電機インフォメーションネットワーク株式会社所属．情報処理安全確保支援士．

西垣 正勝 （正会員）

1990 年静岡大学工学部光電機械工学科卒業．1995 年同大学大学院博士課程修了．日本学術振興会特別研究員（PD）を経て，1996 年静岡大学情報学部助手．同講師，助教授の後，2010 年より同創造科学技術大学院教授．博士（工学）．情報セキュリティ全般，特にヒューマニクスセキュリティ，メディアセキュリティ，ネットワークセキュリティ等に関する研究に従事．2013～2014 年情報処理学会コンピュータセキュリティ研究会主査．2015～2016 年電子情報通信学会バイオメトリクス研究専門委員会委員長．2016 年日本セキュリティマネジメント学会編集部会長．2019 年情報処理学会情報環境領域委員長，2020 年同調査研究運営委員長．本会フェロー．