Vol.2019-DPS-178 No.6
Vol.2019-CSEC-84 No.6
2019/3/4

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

井上佳祐[†1]　本多俊貴[†1]　向山浩平[†1]
大木哲史[†1]　堀川博史[†1]　西垣正勝[†1]

概要：SaaS などのクラウドサービスの普及に伴い，Web アプリケーションに対する攻撃が急増している．本稿では XSS 攻撃に焦点を当て，その対策を検討する．現在，XSS 攻撃の効果的な対策として CSP が有効になりつつある．XSS 攻撃は攻撃者が注入した開発者の意図しないスクリプトが動作してしまう点が問題であるため，CSP では，インラインスクリプトによるスクリプト動作を完全に無効化し，外部スクリプトとそのコード署名を検証することで意図したスクリプトのみ動作させることが可能である．しかし，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あるため，開発者により基本的な対策を徹底することは重要である．しかし，サニタイジングによる悪性コードの完全な無害化や，CSP の適切な設定は容易に行えるものではない．特に現在の 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サイトは数少ないため，CSP の利点を上手く活用できていない．そこで本研究では，CSP によるコード署名が可能な外部スクリプトと対策が難しいインラインスクリプトの対策を組み合わせた手法を採用し，インラインスクリプトの対策としてホワイトリストを利用した対策を提案する．本手法では，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に焦点を当て，各 Web アプリの仕様に合致するホワイトリストを自動生成する方法を確立する．Web アプリケーションのテストツールにホワイトリスト生成の機能を統合することで，従来のアプリ開発工程を変更することなくホワイトリストの自動生成が可能となる．作成したホワイトリストと CSP を上手く組み合わせて，より容易で効果的な XSS 対策を目指す．

キーワード：Cross Site Scripting (XSS)，Content-Security-Policy (CSP)，ホワイトリスト，自動生成，テストケース

1. はじめに

コンピュータ性能の向上とネットワークの広帯域化に伴い，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 開発用のフレームワークなどを使用し，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 サイトがユーザから受け取る入力値を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，ユーザのブラウザ上で任意のスクリプトを実行するこが可能となる．このような攻撃の結果，攻撃者にセッション ID などを奪取されてしまい，その Web サイトにおいて攻撃者によるユーザへのなりすましを許すことや，攻撃者がユーザに対してドライブバイダウンロード攻撃を行うきっかけとして利用される可能性がある．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 攻撃はユーザの PC に直接影響を与える．したがって，XSS 攻撃への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 攻撃を防止できることが分かる．しかし，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 XSS 脆弱性が発見されている[2]ことに鑑みるに，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，困難な作業で有ることが推測される．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．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 攻撃の実体は，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，ポリシー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ー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，既に多くの Web サーバや Web ブラウザに実装されている [3]．しかし，CSP が正しく機

†1 静岡大学
Shizuoka University

Vol.2019-DPS-178 No.6
Vol.2019-CSEC-84 No.6
2019/3/4

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．外部スクリプトの場合は，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ホワイトリストとして用いることによって，当該スクリプトの真正性を検査可能である．これに対し，Web アプリケーション内のインラインスクリプトの場合は，スクリプトへの入力に応じて Web アプリケーションの HTML 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、CPS による XSS 対策を完全に行うには，インラインスクリプトによるスクリプト動作を無効化した上で，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，入力フォームなどの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完全に利用していない Web アプリケーションは皆無と言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，本研究では，インラインスクリプトに対するホワイトリスト型対策を実装し、CSP による外部スクリプトに対するホワイトリスト型対策と併用する方法を提案する．提案手法におけるホワイトリストの作成戦略は，理論ベースのアプローチからテストベースのアプローチへと変更した手法である．提案方式の特徴の一つが，テストベースのホワイトリストを作成するために，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の結合テストの活用である．具体的には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し，Web アプリケーションの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ことにより，XSS 攻撃を検知し防御する．この方法は，(1) スクリプトの実行がテスト工程で事前に確認されたパターン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，仕様から構成される（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．

2. XSS 攻撃とそのメカニズム

クロスサイトスクリプティング（Cross Site Scripting：XSS）の脆弱性には，Reflected XSS，Stored XSS，DOM Based XSS の 3 種類が存在する．XSS 脆弱性は，非持続型か持続型／蓄積型か，サーバ側の処理で発生するのか，クライアント側の処理で発生するのかによって分類される．本章では，3 種類の脆弱性の概要を説明し，Reflected XSS，Stored XSS を例に攻撃メカニズムについて説明する．

2.1 XSS 脆弱性の種類

2.1.1 Reflected XSS

Reflected XSS は，ユーザからの入力をパラメータとして受け取り，サーバ側のプログラムでパラメータによって生成される Web ページをクライアント側に返すタイプの Web アプリケーション（図 1）において発生する可能性のある XSS 脆弱性である．有害な入力値をサーバ側のプログラムがパラメータとして無害化せずに利用することによって発生する．サーバではその値を保持しないため攻撃の持続性はない．

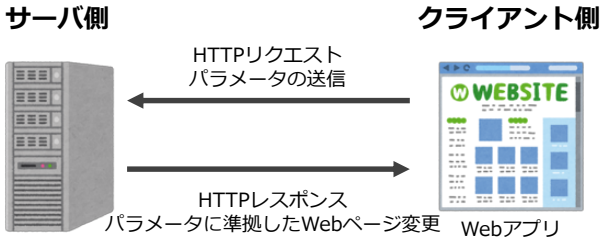

図 1 Reflexted XSS の概要

2.1.2 Stored XSS

Stored XSS は，ユーザからの入力を受け取り，サーバ側のプログラムがその入力値をデータベース(DB)などのストレージに保存し，サーバ側のプログラムで DB に保存された入力値をパラメータとして生成される Web ページをクライアントに返すタイプの Web アプリケーション（図 2）において発生する可能性のある XSS 脆弱性である．サーバがパラメータとして受け取った悪意ある入力値はサーバの DB に保存されており，Web アプリケーションは，その保存された値をアプリケーションが実行される毎に読み込み続けるため，攻撃の持続性がある．

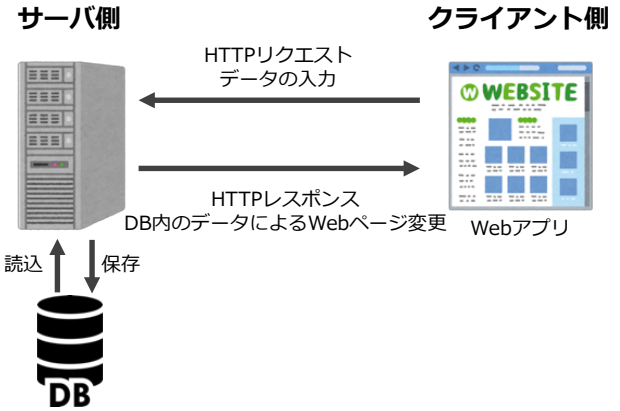

図 2 Stored XSS の概要

2.2 XSS 攻撃のメカニズム

XSS 攻撃は，「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．

2.2.1 Reflected XSS のメカニズム

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（標的 Web アプリケーションの URL+XSS 脆弱性を突くパラメータ）を E メール

Vol.2019-DPS-178 No.6
Vol.2019-CSEC-84 No.6
2019/3/4

に記載するなど，何らかの方法で被害者に送信する（図 3 ①）．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ブラウザを通して標的 Web アプリケーションに送信する（図 3 ②）．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（図 3 ③）．

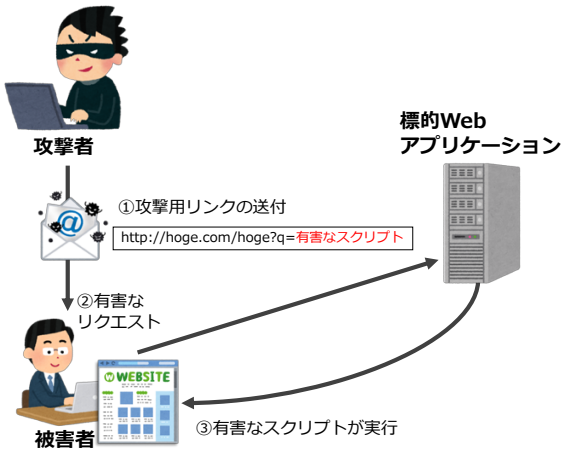

図 3 Reflected XSS の仕組み

2.2.2 Stored XSS のメカニズム

攻撃者は，標的 Web アプリケーションに対して有害なスクリプトを埋め込む，例えば掲示板サイトなどであった場合は，有害なスクリプトを書き込む（図 4 ①）．書き込まれた内容は DB に保存される（図 4 ②）．被害者は標的 Web アプリケーションをよく利用するユーザであるため，標的 Web アプリケーションにアクセスしてしまう（図 4 ③）．標的 Web アプリケーションは，書き込まれている内容を DB から読み込む（図 4 ④）．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（図 4 ⑤）．

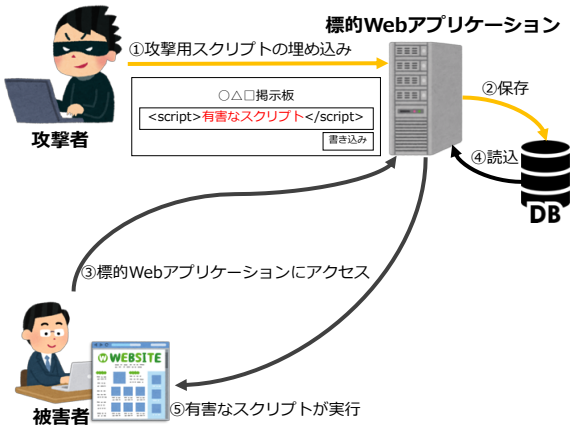

図 4 Stored XSS の仕組み

2.3 攻撃成功後の被害

この脆弱性が悪用され，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が生じることとなる．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，XSS 攻撃を受けていても被害者は気づかない場合が多いと想定される．

3. サニタイジングによる基本対策

XSS 攻撃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パラメータや，DB に格納された悪性な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，パラメータなどのユーザからの入力値を適切に無害化することが，現在一般的に利用されている XSS 対策方法である．具体的には，入力値チェックとエスケープ処理によるサニタイジングである．サニタイジング処理は，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．

3.1 入力値チェック

XSS 攻撃によってブラウザ側でスクリプトを実行させるには，ブラウザがスクリプトであると認識するコード（例えば，「<script>有害なスクリプト</script>」）を含めた形でパラメータに含める必要がある．このようなパラメータに対しては，その中にスクリプトを示す HTML タグ（例:<script>）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サニタイジング対策が可能である．

3.2 エスケープ処理

例えば，ブラウザ上に「<script>有害なスクリプト</script>」という文字列を表示させたい場合は，パラメータ中の<script>タグを無効化することができない．このような場合には，タグが DOM の要素として認識されないように，エスケープ処理を行うという対策が可能である．Web アプリケーションが「<script>有害なスクリプト</script>」というパラメータを受け取った場合は，Web アプリケーション側でエスケープ処理を行い，「<」を「<」に，「>」を「>」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「<script>有害なスクリプト</script$gt;」となり，ブラウザ側ではこれを（スクリプトではなく）文字列として認識し，ブラウザ上では「<script>有害なスクリプト</script>」と正しく表示される．

3.3 問題点

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>」以外にも無数にあり，単純なエスケープ処理をしていてもそれを回避できるバターンが複数存在する．よって，現在の XSS 対策は対策漏れが往々にして発生すると考えられる．大手 IT 企業の Web サービスにおいても同様の脆弱性が発見されていることを考えると，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，現在の一般的な対策方法であるサニタイジングは十分かつ容易な対策となっていない可能性がある．

4. ホワイトリスト型の対策

ホワイトリストを採用した対策では，ホワイトリストになにがホワイトとなるのかを定義することができれば，定義したもの以外を否定するだけなので，安全性の面からみるととても頑丈な対策と言える．Web アプリケーションの動作は普遍的なため，このようなアプリケーションへの攻撃に対しては，ホワイトリストに基づく対策を採用し，事前定義された機能のみを有効にすることが効果的である．実際，BEEP[4], ConScript[5], CSP[6]など，セキュリティポリシーで設定された以外のスクリプトの動作を禁止し XSS を防止する手法がある．これはホワイトリストを利用した対策手法に限りなく類似した手法であり実質ホワイトリストベースの XSS 対策と言える．ホワイトリストベースの対策では，ホワイトリストの品質を向上させることが非常に重要である．この点に関して，ホワイトリストをより拡充させるための経験的アプローチ [7][8] が存在する．

4.1 ポリシーベースホワイトリスト

Trevor は，開発者がアプリケーションにセキュリティポリシーを記述することによってクライアントのブラウザ上で JavaScript のコードを書き換え，セキュリティ上重要な API をブラウザからフックすることにより，セキュリティポリシーに記述された API へのアクセスを限定する手法であるブラウザ強制埋込ポリシー（Browser-Enforced Embedded Policies : BEEP）[4] を提案している．

Meyerovich らは，BEEP [4] と非常に類似しているが，セキュリティ上重要な API に対してアスペクト指向プログラミング（Aspect Oriented Programming : AOP）を用いてクライアントのブラウザ上で API をフックすることにより，ホワイトリスト型のセキュリティポリシーをアプリケーションに適用する手法である ConScript [5] を提案している．

Sid らは，サーバの HTTP レスポンスヘッダにセキュリティポリシーを付加することでブラウザにおけるアプリケーションの動作を制限する手法であるコンテンツセキュリティポリシー（Content Security Policy : CSP）[6] を提案している．CSP は，実際にいくつかのブラウザとサーバに機能が実装されており，すでに利用可能となっている[3].

4.2 経験ベースホワイトリスト

厳密には XSS 攻撃対策ではないが，角田らは，マルウェア感染検知のためのホワイトリストおよびブラックリストを自動的に拡充する手法 [7] を提案している．これは，ネットワークのアクセスログを分析し，Web サイトの悪性度を算出することで，Web サイトをホワイトリスト，ブラックリスト，グレーリストに分類する手法である．グレーリストに分類された Web サイトにおいては，それが良性なのか悪性なのかを再判別するため，マルウェア（自動化プログラム）では突破が困難となるような形式で追加の認証テストが行われる．

Sid らが提案した CSP [6] には，report-only モード [8] とよばれる動作モードがある．この動作モードは CSP の適用により，ポリシー違反が起こった場合に開発者にどのような違反が発生したのかなどのレポートが送信されるようになっている．厳密には， report-only モードの場合はポリシー違反のレポートの送信は行うが，スクリプトの制御・ポリシーの強制までは行わない．XSS 攻撃により許可されていないスクリプトが実行されそうになった場合もポリシー違反となるため開発者はこれらのレポートからフィードバクを受け Web アプリケーションを改善させることができる．したがってこれらも経験ベースのアプローチといえる．

4.3 問題点

ポリシーベースの手法では，ホワイトリストとなるセキュリティポリシーは開発者自身が決定する必要がある．つまり，開発者は Web アプリケーションのセキュリティについての妥当な知識が必要である．

また，それだけでなく Web アプリケーションの肥大化も問題になる．現在の Web アプリケーションの構造と動作は非常に複雑であるため，開発者でさえアプリケーションの動作を正確に把握することは容易ではない．

したがって，ポリシーを設定することは困難で，不十分な設定が原因でエラーが発生しやすくなる．このようなエラーは，他の脆弱性を引き起こし，攻撃者がセキュリティポリシーをバイパスして不正な動作を行わせる可能性がある．言い換えれば，必要十分なセキュリティポリシーを作成する方法論はまだ確立されておらず，このようなポリシーベースの XSS 対策の効果は非常に限定されていると言える．

経験ベースの方法は，ホワイトリストをある程度拡充するのに有用ではあるが，このような経験的アプローチでは理論的根拠がまだ不足している．この種の不完全さは，既存のすべてのホワイトリストベースのアプローチが直面している大きな問題である．

Vol.2019-DPS-178 No.6
Vol.2019-CSEC-84 No.6
2019/3/4

5. Content Security Policy

CSP はサニタイジングの不備により注入されてしまったスクリプトが動作し致命的な影響を及ぼしてしまうのを軽減しそれを報告するための追加対策として採用される．この機能は 4.1 節のとおり 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の動作範囲を示すセキュリティポリシを付加し，Web アプリケーションのソースコードに適切な記述を行うことで有効化される．本章では，外部読み込み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サブリソース完全性（Subresource Integrity : SRI）[9] ，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 [10] script-src [11] ，およびこれらの機能を利用した XSS 攻撃対策のための設定とその問題点を説明する．

5.1 Subresource Integrity

外部読み込み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 攻撃へと繋がる可能性がある．それらの改ざんを検知し，スクリプトの実行を防ぐための機能が SRI である．SRI を利用するには，外部から読み込むスクリプトのハッシュ値と一致するハッシュ値を HTML ソース内に指定する．当然，ハッシュ値の一致しない外部読み込みスクリプトは実行されることはない．具体的には，図 5 で示す JavaScript ファイル（helloalert.js）を外部から読み込む場合，この JavaScript ファイルの SRI ハッシュ値（SHA384, BASE64）を生成すると図 6 になるため，HTML ソース内の外部スクリプトの読み込みは図 7 のように記述する．また，HTTP レスポンスヘッダにおいては図 8 を返す必要がある．

helloalert.js

```
function helloalert(){
    alert(“Hello, World!”);
}
```

図 5 外部読み込み JS ファイル（helloalert.js）の例

```
o52/r5ior7XHmkV+V+pu7LEVdnDlX5GTUZveXiaI4CDcoIrFFBf2FgHOlsKftBBi
```

図 6 helloalert.js の SRI ハッシュ値

```
<script src=”https://example.com/helloalert.js” integrity="sha384-o52/r5ior7XHmkV+V+pu7LEVdnDlX5GTUZveXiaI4CDcoIrFFBf2FgHOlsKftBBi "></script>
```

図 7 HTML 内での SRI ハッシュ値の指定例

```
Content-Security-Policy: require-sri-for script;
```

図 8 CSP（SRI）の HTTP レスポンスヘッダ設定

5.2 default-src / script-src

この機能を利用することで，外部スクリプトの読み込み先ドメインを制限することが可能となる．XSS 攻撃により任意のスクリプト・HTML タグが注入可能であった場合，攻撃者は自身のサーバに配置した外部スクリプトをコード署名付きで注入できるということである．そのような場合，5.1 節で説明したコード署名による対策を施している場合であっても，検証するハッシュ値は一致するのでスクリプトの動作を止めることはできない．しかし，読み込み先のドメインを制限（例えば，自サイトのドメインからのみ）することで，その問題を緩和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ように設定することも可能である．例として，自サイトのドメインが example.com であった場合，図 9 で示す CSP の HTTP レスポンスヘッダを返すことで，図 10 のような読み込み元のドメインが自サイトのものでない場合や，図 11 のようなインラインスクリプトが HTML 内に記述されている場合もスクリプトが実行されない．

```
Content-Security-Policy: default-src 'self'
```

図 9 CSP（default-src）の HTTP レスポンスヘッダ設定

```
<script src=”https://ex-evil.com/attack.js”></script>
```

図 10 自サイト外からの外部スクリプト読み込み例

```
<script>alert(“XSS!”);</script>
```

図 11 インラインスクリプトの例

5.3 XSS 攻撃対策設定とその問題点

XSS 攻撃により任意のスクリプト実行されてしまう原因となっているのは，インラインとしてのスクリプト注入と，外部スクリプトとしての注入の 2 種類のため，5.1 節，5.2 節で示した CSP の機能を両方用いて，インラインスクリプトの完全な無効化とコード署名があり，かつ自サイトドメインに配置されている外部スクリプトのみの実行を許可するよう設定することが，XSS 攻撃においては最も効果的な対策方法であると考えられる．

しかし，CSP は，まだ広く普及しているわけではない．その理由の 1 つとして，ブラウザやサーバ毎に実装の対応状況が異なるため想定通りに動作をしない可能性があるためである．またポリシーの設定が多少複雑であり適切な設定を行うことが相応のセキュリティ知識が貧しい開発者では困難であるからだと考えられる．文献 [12] では，人気サイトを調査した結果 CSP 利用率が低いと指摘している．またこの文献ではインラインスクリプトについても言及しており，インラインスクリプトを利用していないサイトについては極めて少ないと指摘していた．本件について独自に Alexa ランキング [13] TOP50 のうち 43 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．インラインスクリプトの制御が行われていない場合，サニタイジングされていない文字列からの XSS 攻撃を依然として受けてしまう可能性がある．

すなわち，いまだ現在の 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サイトは数少ないため，CSP の利点を上手く活用できていないのが現状である．

6. 提案手法

6.1 システム概要

本研究では，インラインスクリプトを廃止できない Web アプリケーションにおいても効果的な XSS 攻撃対策を実現できるように，CSP のみのスクリプト制御から，CSP によるコード署名が可能な外部スクリプトと対策が難しいインラインスクリプトの対策を組み合わせた手法を採用し，インラインスクリプト側の対策としてテストベースホワイトリストを利用した対策を提案する．テストベースホワイトリストと CSP を上手く組み合わせて，より容易で効果的な XSS 攻撃対策を目指す. XSS 攻撃対策システムの概要を図 12 に示す．コード署名によりスクリプト動作が意図したものかどうかを確認できる外部スクリプトの制御は，CSP の SRI と default-src 機能に任せ，サニタイジングの不備によりインラインスクリプトとして注入されるスクリプトの制御は，次節で提案するテストベースホワイトリストで対策を行う．

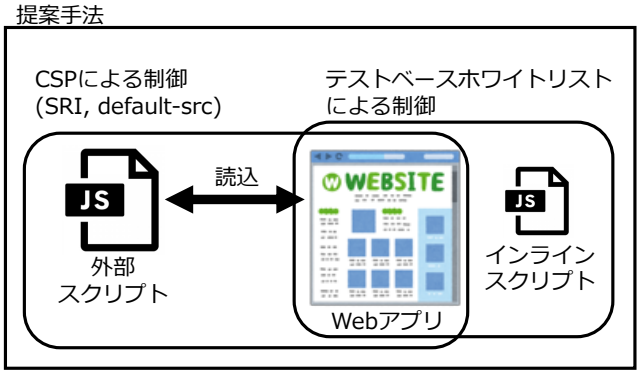

図 12 提案システムの概要図

6.2 テストベースホワイトリスト

6.2.1 提案概要

本提案では，4 章で述べたホワイトリストベース XSS 攻撃対策に存在する問題を解決するため，ホワイトリストの作成戦略を理論ベースのアプローチからテストベースのアプローチへと変更することを提案する．本稿では，テストベースのホワイトリストを作成するため，Web アプリケーションの開発工程の最終段階として実施されるソフトウェアの結合テストに焦点を当てる．テストベースのホワイトリストは，テスト工程で事前に確認されたスクリプトのみを実行することを許可する．具体的には，テスト工程で検証された動作をホワイトリストとして定義し，Web アプリケーションの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ことで XSS 攻撃を検知して防止する．ソフトウェアテストは，各 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，各 Web アプリケーションのスクリプトごとに，仕様を逸脱することのないテストベースのホワイトリストを自動生成することが可能となる．

まとめると，この手法には以下のような利点がある．

(1) スクリプトの実行がテスト工程で事前に確認されたパターン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，仕様から構成される（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．

6.2.2 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，事前にホワイトリストとして確認された動作のみを定義することである．これは理論的に必要かつ十分なホワイトリストではないが，テストと仕様の連携によってヒューリスティックに十分なホワイトリストを作成することができる．

XSS 攻撃の原因は，Web アプリケーション開発者が意図していなかったスクリプトが注入されることである．つまり，問題の本質は，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防ぐためには，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可能であると考えられる．仕様書には，実装されるべき全ての機能と，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ある. すなわち, 仕様書というのは，「意図された動作の集合」であると言える. 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ている. 開発者は, Web アプリケーションをリリースする前に, 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どうか，または仕様書で示されている通りの動作をするかどうかをテストする．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テストベースのホワイトリストを生成することが可能である．

ただし，このようなテストにおいて 100%のカバレッジを達成することは簡単な作業ではない．これを軽減するために，HTML ページソースコードのスクリプト構造に焦点を当てる．前述のように，XSS 攻撃の原因は，HTML ページ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 攻撃は，HTML ページソースコードのスクリプト構造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．HTML のスクリプト構造は，典型的な入力値をテストするだけで収集できるため，ホワイトリストを作成するために網羅的なテストは必要でないと考えられる．さ

Vol.2019-DPS-178 No.6
Vol.2019-CSEC-84 No.6
2019/3/4

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．そのため本稿においては， HTML ページソースコードからスクリプト部分（Script タグ内の JavaScript および W3C で定義されているマウスイベント内の JavaScript）のみを抽出し，テスト工程を通して，ホワイトリストに登録している．特定の HTML ページソースコードに対して生成されたホワイトリストの例を図 13 に示す．

6.2.3 検知手法

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，ページの HTML ソースコードの構造が変化する．提案手法では，テスト工程において表示・実行された HTML ページソースコードのスクリプト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（通常の意図されたパラメータを入力する限り），Web アプリケーションによって生成される HTML ページソースコードは，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，攻撃者によって Web アプリケーション開発者が想定していないパラメータが入力されると，ホワイトリストに登録されていない HTML ページソースコードが生成される．XSS 脆弱性のある Web アプリケーションにおいて，スクリプトを含むパラメータが入力されると，Web アプリケーションは，意図しないスクリプトが注入された HTML ページ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 ページ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．提案方式では，この特徴を用いて XSS 攻撃を検知する．この特徴は，Reflect 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．

提案手法における XSS 攻撃検知は，クライアントブラウザ上に実装される．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 アプリケーションによって生成された HTML ページ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 ページソースコードのスクリプト構造が常に比較される．両方の構造が一致しない場合，XSS 攻撃が発生していると判断できる．図に XSS 攻撃の例を示す．図 13 の Web ページにおいて「`<script>alert(“Attack!”)</script>`」などの JavaScript を入力することにより，図 14 で示す意図しない HTML ページソースコードが生成される．図 13 および図 14 で示すように，スクリプト構造が異なるため，攻撃を検知することができる．

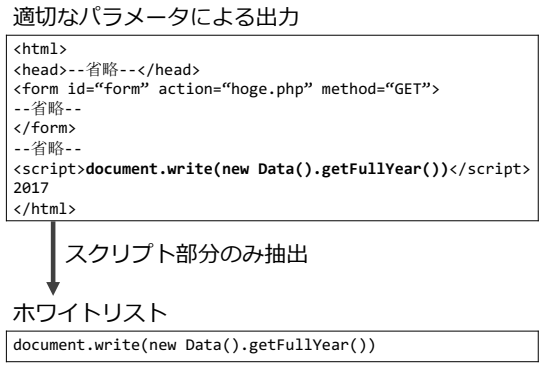

図 13 ホワイトリストの作成例

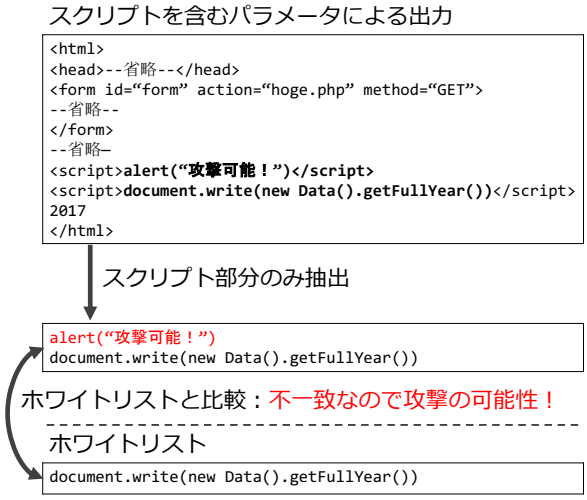

図 14 XSS 攻撃の可能性がある場合の例

6.2.4 ホワイトリスト自動生成

6.2.2 項で述べたように，本稿では，前もってホワイトリストとして確認された動作のみを定義し，ヒューリスティックに適切なテストベースのホワイトリストを作成するために仕様書を利用することを提案している．テストベースのホワイトリストは，仕様書に示された意図されたパラメータが入力された時に，Web アプリケーションが生成する HTML ページソースコード内のスクリプト部分で構成されている．これは，提案方式のホワイトリスト生成プロセスが，Web アプリケーションのテスト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時，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．つまり，ホワイトリスト生成に必要なすべての HTML ページソースコードは，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．

我々が着目したように，提案方式は，Web アプリケーシ

ョンのテスト工程においてホワイトリスト生成プロセスを統合することが可能である．より具体的には，テスト工程において生成された各 HTML ページソースコードからすべてのスクリプト部分を抽出することにより，提案方式におけるテストベースホワイトリストを生成することができる．この点に着目し，Web アプリケーションのテストツールを改良して自動ホワイトリスト生成機能を実装することを提案する．この機能を採用することで，通常の Web アプリケーション開発工程で Web アプリケーションごとにテストベースのホワイトリストを生成することが可能となる．これにより，ホワイトリストの XSS 攻撃検知をより効果的かつ合理的に展開することができる．

6.2.5 デプロイ

6.2.3 項で述べたように，本検知方法では，実行されるスクリプトの構造はユーザ側のクライアントブラウザで検証される．したがって，提案手法を実用化するためには，生成されたホワイトリストをユーザ側に送信し，クライアントブラウザから利用できるようにする必要がある．これは，対応する Web コンテンツの HTML ページソースコード中にホワイトリストへのハイパーリンクを埋め込むことである．これにより，Web アプリケーションをサーバから受信すると同時にホワイトリストがダウンロードされる．ホワイトリストのダウンロードと検証は，ブラウザの拡張機能によって実装することが可能である．本提案手法の実現全体像を図 15 に示す．

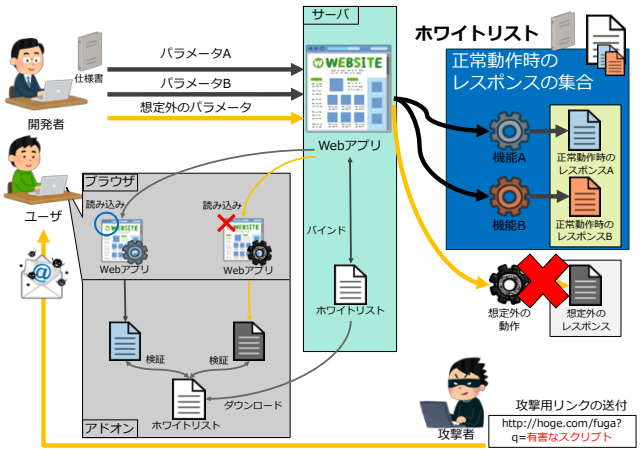

図 15 テストベースホワイトリストのデプロイ

7. まとめと今後の課題

本稿では，XSS 攻撃対策として，コード署名が可能な外部スクリプトを CSP により防御し，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をテストベースホワイトリストで防御する 2 つを組み合わせた手法を提案した．

テストベースホワイトリストは，XSS 攻撃検知のために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手法である．本手法では，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．完全性の観点から，テストベースのホワイトリストは，テスト工程で事前に確認されたスクリプトのみが実行される．健全性の観点から，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．

今後は，各機能の実装とブラウザでの動作確認をはじめとして，実際の仕様書とソフトウェアテストを用いたホワイトリストの自動生成，検知アドオンのパフォーマンステスト，検出率・誤検出率の実験による有効性評価を含め，本手法を検証および改良していく．

参考文献

[1]”安全なウェブサイトの作り方 改訂第 7 版”. https://www.ipa.go.jp/files/000017316.pdf, (参照 2017-12-18).
[2]“「マウスオーバーの」問題についての全容”. https://blog.twitter.com/official/ja_jp/a/ja/2010-26.html, (参照 2017-12-09).
[3]”コンテンツセキュリティポリシー (CSP) - HTTP | MDN”. https://developer.mozilla.org/ja/docs/Web/HTTP/CSP, (参照 2019-02-01).
[4]Jim, T.. Defeating script injection attacks with browser-enforced embedded policies. Proceedings of the 16th international conference on World Wide Web. ACM. 2007, p. 601-610.
[5]Meyerovich, M. and Livshits, B.. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. Security and Privacy IEEE Symposium on. IEEE. 2010, p. 481-496.
[6]Stamn, S., Sterne, B. and Markham, G.. Reining in the web with content security policy. Proceedings of the 19th international conference on World Wide Web. ACM. 2010, p. 921-930.
[7]角田朊，大鳥朊哉，藤井康広，谷口信彦，木城武康．グレーリストを用いたホワイトリスト/ブラックリストの自動生成によるマルウェア感染検知方法の検討. IPSJ SIG Technical Reports, 2014-CSEC-66-16. 2014, p. 1-7.
[8]“Content-Security-Policy-Report-Only - HTTP | MDN”. https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/Content-Security-Policy-Report-Only, (参照 2019-02-01).
[9]“サブリソース完全性 - ウェブセキュリティ | MDN”. https://developer.mozilla.org/ja/docs/Web/Security/Subresource_Integrity, (参照 2019-02-01).
[10]“CSP: default-src - HTTP | MDN”. https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/Content-Security-Policy/default-src, (参照 2019-02-01).
[11]“CSP: script-src - HTTP | MDN”. https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/Content-Security-Policy/script-src, (参照 2019-02-01).
[12]Kerschbaumer, C., Stamm, S., & Brunthaler, S.. Injecting CSP for fun and security. Paper presented at the 2nd International Conference on Information Systems Security and Privacy, Rome, ITA. 2016, p. 15-25.
[13]“Alexa Top 500 Global Sites”. https://www.alexa.com/topsites, (参照 2019-01-09).