情報処理学会論文誌 Vol.63 No.12 1786–1803 (Dec. 2022)

マイクロフロントエンドにおける DOM Based XSS攻撃に対する テストベースホワイトリストの有用性

井坂 佑介[1] 天笠 智哉[1] 奥村 紗名[1] 佐々木 葵[1] 大木 哲史[1] 西垣 正勝[1,a)]

受付日 2022年3月8日, 採録日 2022年9月2日

概要：Webサービスの複雑化にともない，マイクロフロントエンド（MFE）型のWebアプリケーション開発へと移行している．しかし，MFEにおいては，マイクロサービスどうしを組み上げる際に，各マイクロサービス間でセキュリティポリシのコンフリクトが発生しうる．この課題に対処する方法としては，セキュアプログラミングガイドラインの運用によってマイクロサービス間のセキュリティポリシを共通化する方法や，API Gatewayによってセキュリティポリシの調停を行う方法があげられる．しかし，セキュアプログラミングガイドラインの導入は，サービスの開発に注力したい開発者にとって技術的・作業的負担が大きいという問題がある．また，DOM Based XSS攻撃のようにフロントエンドで攻撃が完結するものに対しては，API Gatewayでの調停が機能しないという問題がある．そこで本論文では，MFE型Webアプリケーションに対し，テストベースホワイトリストを利用したセキュリティ機構を採用することを提案する．提案方式により，既存のマイクロサービスの再利用性を高めつつ，MFE型Webアプリケーションの DOM Based XSS対策を達成する．

キーワード：マイクロサービスアーキテクチャ，マイクロフロントエンド，DOM Based XSS，テストベースホワイトリスト，セキュリティポリシ

Effectiveness of Examination-based Whitelist for DOM Based XSS Attacks in Micro Front-end

Yusuke Isaka[1] Tomoya Amagasa[1] Sana Okumura[1] Aoi Sasaki[1] Tetsushi Ohki[1]
Masakatsu Nishigaki[1,a)]

Received: March 8, 2022, Accepted: September 2, 2022

Abstract: As the complexity of Web services increases, Web application developments are migrating to Micro Front-end (MFE). However, in MFE, the security policies of each microservice may conflict each other when assembling microservices. There are two ways to deal with this challenge: one is to standardize security policies among microservices by implementing secure programing guidelines, and the other is to mediate security policies among microservices by using API Gateway. However, the introduction of secure programing guidelines is technically and operationally burdensome for developers who want to focus on service design. In addition, mediation using the API Gateway does not work for attacks that are completed in the front-end, such as DOM Based XSS attacks. Therefore, in this paper, we propose to adopt a security mechanism using a test-based whitelist for MFE Web applications. The proposed scheme achieves DOM Based XSS attacks countermeasures for MFE Web applications while enhancing the reusability of existing microservices.

Keywords: microservices architecture, micro front-end, DOM based XSS, examination-based whitelist, security policy

[1] 静岡大学
Shizuoka University, Hamamatsu, Shizuoka 432–8011, Japan
[a)] nishigaki@inf.shizuoka-u.ac.jp

© 2022 Information Processing Society of Japan

1. はじめに

近年，Web アプリケーションの複雑化にともない，設定変更への俊敏な対応や短期間での実装が Web アプリケーション開発の場で要求されるようになった．これらに対応するため，従来のモノリシックアーキテクチャから，マイクロサービスアーキテクチャ（MSA）[1] 型の Web アプリケーション開発へ，さらにはマイクロフロントエンド（MFE）[4] 型 Web アプリケーション開発へと移行が進んでいる．将来的には，サードパーティ製のマイクロサービスが部品化され，これらのマイクロサービスを組み上げてサービスを構築するような開発が行われると考えられる [2], [3]．

しかし，MFE 型の Web アプリケーション開発の場においては，複数のマイクロサービスを 1 つのサービスに統合する際に，マイクロサービス間のセキュリティポリシにコンフリクトが発生しうるため，セキュリティバイデザインの実現に根本的な課題をかかえている．この課題に対処する方法としては，セキュアプログラミングガイドラインの運用によってマイクロサービス間のセキュリティポリシを共通化する方法 [5] や，Web アプリケーションのフロントエンドとバックエンドの中間に位置する API Gateway によってセキュリティポリシの調停を行う方法があげられる [6]．しかし，セキュリティ要件が複雑であり，かつ設計にあたっての方法論が不足している MFE 型 Web アプリケーションにおいては，セキュアプログラミングガイドラインの導入は困難 [7] である．さらには，仮にガイドラインが導入できたとしても，サービスの開発に注力したい開発者にとって，つねに最新のガイドラインを熟知したうえで怠りのない配慮の下にサービス開発を行うことは技術的・作業的負担が大きいという問題がある．また，DOM Based XSS 攻撃のようにフロントエンドで攻撃が完結するものに対しては，フロントエンドの後段に位置する API Gateway での調停が機能しないという問題がある [8]．

本論文では，セキュリティポリシを利用したセキュリティ機構に代わる対処法として，テストベースホワイトリスト [9] を利用したセキュリティ機構を MFE 型 Web アプリケーションに適用することを提案する．セキュリティポリシを解除することにより既存のマイクロサービスの再利用性を高めつつ，ホワイトリストを用いて MFE 型 Web アプリケーションの DOM Based XSS 対策を実現する．

2. 課題設定

2.1 マイクロフロントエンド（MFE）

これまでの Web アプリケーションの開発では，単一のアプリケーションとして構築されるモノリシックアーキテクチャが主流であった．しかし，近年，Web アプリケーションの複雑化にともない，ビジネスや環境の変化に迅速かつ柔軟に対応することが困難になってきている．モノリシックアーキテクチャでは，1 つのモジュールの修正が他のモジュールに大きく影響しうるため，アジャイルな開発保守の達成が阻害される．このようなモノリシックアーキテクチャの問題を解消するため，2014 年に Martin らによって，マイクロサービスアーキテクチャ（MSA）が提唱された [10]．さらに，データベース，バックエンド，フロントエンドを完全に MSA 化したマイクロフロントエンド（MFE）型のアーキテクチャへと進化している [4]．MFE では，1 つのアプリケーションを機能ごとに分割したものをマイクロサービスとして扱い，それぞれのマイクロサービスが独立して開発・デプロイを行えるようにする．これにより，工数削減，開発サイクルの短縮，スケーラビリティの向上など様々な恩恵を受けることができる．将来的には，サードパーティ製のマイクロサービスが部品化され，これらのマイクロサービスを組み上げてサービスを構築するような開発も行われると考えられる [2], [3]．

しかし，マイクロサービスごとに独立して開発が行われることで，モノリシックアーキテクチャの際には潜在していた問題が顕在化する．その 1 つが，セキュリティバイデザインの達成の困難性である．モノリシックアーキテクチャであれば，開発チームとフロントエンドサービス，バックエンドサービスが一塊であるため，設計段階からセキュリティを確保することも比較的容易である．一方，MFE においては，機能ごとに分割されたマイクロサービスどうしが独立して開発が行われるため，個々のマイクロサービスの更新が進むにつれて，サービス全体のセキュリティの一貫性が崩れる可能性がある．また，サードパーティ製のマイクロサービスを統合してサービスを開発する際には，すでに完成済みのマイクロサービスを後から組み上げるという順序になるため，マイクロサービスごとのセキュリティポリシによっては統合の際にコンフリクトが発生しうる．

本論文では，特に，2.2 節で詳述する「マイクロサービス A の出力が A のセキュリティポリシによってオーバライドされた結果，マイクロサービス B のセキュリティポリシと非合致し，B の入力として拒絶される」というタイプのコンフリクトに焦点を当てる．

2.2 MFE におけるセキュリティポリシのコンフリクト

文字列入力を受け付ける変数に関して，異なるセキュリティポリシの下に開発された 2 つのマイクロサービス A，B を統合して，新たなサービス α を開発する場合を考える（**図 1**）．ここで，A，B，α の想定は以下のとおりとする．マイクロサービス A は，ユーザからの文字列入力を司るサービスである．A におけるセキュリティ対策では，CSP の DOM Based XSS 対策ディレクティブである TrustedTypes [11] の createHTML メソッドを用いることで，入力値に対してエスケープ処理（文字列の無害化）を

© 2022 Information Processing Society of Japan

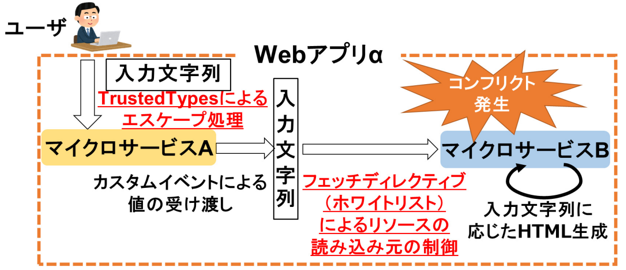


図 1　セキュリティポリシのコンフリクトの例
Fig. 1　Example of security policy conflict.

行っている．マイクロサービス B は，文字列型の内部変数に従って，ページの表示内容を変化させるサービスである．B におけるセキュリティ対策では，CSP のフェッチディレクティブ [34] の img-src [35] を用いることで，リソースの読み込み元に対するホワイトリスト制御を行っている．サービス α では，マイクロサービス A に入力された文字列をマイクロサービス B の入力として受け渡すことによって，ユーザから受け付けた文字列に応じて画像リソースの読み込み元を変更し，ページの表示内容を変えるサービスを実現する．

MFE では，マイクロサービス間の値の受け渡しを行うことは推奨されていないが，実際の開発においては，アプリケーションの仕様上，値の受け渡しが必要になる場合が往々にして存在する．あるマイクロサービスから他のマイクロサービスへの値の受け渡しには，たとえば，Custom Element を用いたカスタムイベント [12] を定義することで実装される．サービス α において，カスタムイベントを用いてマイクロサービス A に入力された文字列をマイクロサービス B へ受け渡しを行うにあたり，マイクロサービス A は自身のセキュリティポリシ（createHTML メソッドで定義されたルール）に従い，エスケープ処理によって入力値の文字列を変更している．この結果，マイクロサービス B への入力値が，マイクロサービス B にとっては想定外の文字列になり，A から B への値の受け渡しにあたってコンフリクトが発生する．（6 章の図 13 および図 14 に具体例を示す）．さらに，TrustedTypes は，Web アプリケーション全体に適用されるため，他のマイクロサービスにおいて innerHTML などの実装による文字列の入力を行おうとした場合，エラーが発生する．

これを解決するためには，マイクロサービス A と B の間でセキュリティポリシの調停を行う必要がある．しかし，マイクロサービスを越境した設計変更を生じさせてしまうことは，MFE のコンセプトから外れることになる．また，A や B がサードパーティ製のマイクロサービスであった場合には，A あるいは B の開発元に改修を依頼することは実質的に不可能であろう．このように，単純なマイクロサービスの連携においてでさえ，セキュリティポリシのコンフリクトが露見する．多数のマイクロサービスが連携する大規模 Web アプリケーションにおいては，セキュリティポリシのコンフリクトは大きな問題となる．

この問題に対し，Li らは，MSA 型 Web アプリケーションのコードからセキュリティポリシを自動生成する仕組みを提案・実装している [13]．しかし，文献 [13] のセキュリティポリシ自動生成機構は，マイクロサービス間のセキュリティポリシのコンフリクトを完全に解決する機能までを有しているわけではない．セキュリティポリシのコンフリクトをかかえる大規模 Web アプリケーションに対しては，セキュリティポリシの自動生成が達成できないケースも往々にして生じうると考えられる．

2.3　セキュアプログラミングガイドライン

マイクロサービスを統合して新規サービスを開発する際にマイクロサービス間でセキュリティポリシのコンフリクトが発生しないように，マイクロサービス開発において留意すべき点を明確化し，これを，個々のマイクロサービスを開発・実装する際に順守すべきセキュアプログラミングガイドラインとしてルール化するという方法が考えられる．しかし，セキュアプログラミングガイドラインの導入は，サービスの開発に注力したい開発者にとって技術的・作業的負担が大きいという問題がある．これは，ガイドライン順守の不徹底を招く要因となり，Web アプリケーションに脆弱性を潜在させてしまう原因となる恐れがある．

また，セキュアプログラミングガイドラインはマイクロサービスの実装方法をある程度共通化させることを意味する．これは，個々のマイクロサービスを機能分化させて部品化するという MFE 型の Web アプリケーション開発に対し，マイクロサービスの自由度を狭めてしまう可能性を孕みうる [13]．MFE では，個々のマイクロサービスの更改（デプロイ）が頻繁に発生する．デプロイのたびにセキュリティポリシの修正も（必要に応じて）必要となる．セキュリティポリシを共通化すると，セキュリティポリシの修正のたびに「共通ルールに則っているか」という点の確認までが要求される分，迅速なデプロイに対する阻害要因が増加する．

2.4　Content Security Policy

Content Security Policy（CSP）とは，クロスサイトスクリプティング（XSS）攻撃やデータインジェクション攻撃などの特定の種類の攻撃を検知し，影響を軽減するためのセキュリティ機構である [14]．CSP を導入し，設計段階において決定したセキュリティポリシを Web アプリケーションに適用することにより，セキュリティバイデザインの達成が期待できる．

しかし，サードパーティ製のマイクロサービスを統合して新たなサービスを開発するケースにおいては，個々のセキュリティポリシの下で設計・実装されている複数のマイ

© 2022 Information Processing Society of Japan

クロサービスを結合する形になる．すなわち，新規サービスを企画・設計する段階で決定したセキュリティポリシを，すでに実装が完了しているマイクロサービスに遡及させることはできない．このため，新規サービスのセキュリティポリシと既存のマイクロサービスのセキュリティポリシがコンフリクトする場合には，その不整合が，2.2 節で例示したようなマイクロサービス間のセキュリティポリシのコンフリクトとして顕在化することになる．

CSP については，セキュリティポリシ設定の難度の問題も存在する．MFE においては，攻撃に対して柔軟に対応できるように複雑なセキュリティポリシを採用する必要があるが，現状は開発者が手動でセキュリティポリシを設定しているため，時間がかかるうえにミスを犯しやすい．各マイクロサービスの更改（デプロイ）のつど，適切なセキュリティポリシに迅速に変更する必要があり，手動設定は非現実的である．

2.5 API Gateway

API Gateway [5] とは，フロントエンドとバックエンドの中間に位置する MFE のセキュリティ機構であり，API の管理をはじめとして，セキュリティポリシの作成および実装を行うことで様々なセキュリティ保護を施すことができる．2.2 節に示した新規サービスと既存のマイクロサービスのセキュリティポリシのコンフリクトの問題に対しても，API Gateway にポリシ間の調停を依頼することが可能である．しかし，API Gateway はフロントエンドとバックエンドの中間に配置されるがゆえに，フロントエンドで攻撃が完結してしまうようなサイバー攻撃に対しては，セキュリティ保護を行うことができない．このような攻撃の典型が，3.1 節で詳述する DOM Based XSS 攻撃である．

2.6 MFE における DOM Based XSS 攻撃とその対策

2.1，2.2 節で述べたように，MFE の開発アプローチは，セキュリティバイデザインとの親和性が低く，2.4 節で述べたように，CSP の機構を導入したとしてもマイクロサービス間のセキュリティポリシのコンフリクトの調停が難しい．そして，2.5 節で述べたように，DOM Based XSS 攻撃は API Gateway による防御が不可能である．よって，現状，MFE においては，DOM Based XSS 攻撃に対する有効的な対策が存在していない状況となっている．

そこで本論文では，MFE における DOM Based XSS 対策のアプローチとして，テストベースホワイトリスト [9] の適用を検討する．テストベースホワイトリストとは，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に実施される開発テストと結合し，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手法である．Web アプリケーション利用時には，開発者の想定外の入力（未テストの入力）によって有害な動作が実行されることを禁止することができる．

セキュリティバイデザインに基づいた開発においては，Web アプリケーションの設計段階でセキュリティ要件が精査されて，セキュリティポリシとして仕様書に組み込まれる．したがって，Web アプリケーションの仕様書に基づいて開発テストを行うことによって，Web アプリケーションの開発工程の中で「仕様書（セキュリティポリシ）に従った入力のみを許可するホワイトリスト」の自動生成が達成され，この結果，DOM Based XSS 攻撃が防がれる．

本論文の目的は，セキュリティポリシを利用したセキュリティ機構に代わる対処法として，テストベースホワイトリストを利用したフロントエンド側のセキュリティ機構の提案にある．本論文では，各マイクロサービスのセキュリティポリシが，CSP のディレクティブの記法によって規定されているという想定を置いている．開発者は，既存のマイクロサービスのセキュリティポリシ（CSP のディレクティブ）を取り外すことによって各マイクロサービスを部品化し，これらを再利用しながら MFE 型 Web アプリケーションを開発する．これにより，セキュリティポリシのコンフリクトの制約を受けることなく，自由度が高い形でマイクロサービスの再利用が可能となる．セキュリティポリシに基づくセキュリティ機構が解除される代わりに，ホワイトリストに基づくセキュリティ機構によって Web アプリケーションの安全性が維持される．

3. DOM Based XSS 攻撃に対する既存対策

3.1 DOM Based XSS 攻撃

DOM Based XSS 攻撃は，XSS 攻撃の一種であり，Web ブラウザ内で Web ページが生成（レンダリング）されるタイミングで，Web ページに有害なスクリプトを埋め込む攻撃である [15]．攻撃者は，攻撃用リンク（正規の URL のアンカ（「#」以降の文字列）やクエリパラメータ（「?」以降の文字列）に有害な文字列を記述したもの）を何らかの方法で被害者に送信する（図 **2** ①）．被害者が攻撃用リンクにアクセスすることにより，Web ブラウザから Web サーバにリクエストが送信される．その際，アンカや（Web サーバへの問合せの必要のない）クエリパラメータについては，Web ブラウザ内で処理される情報であるため，Web サー

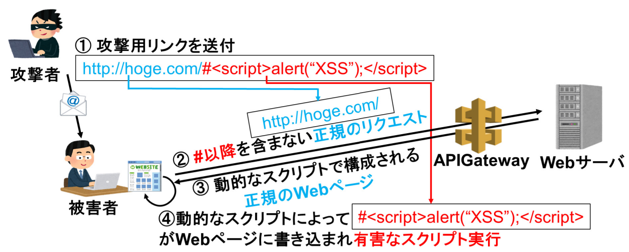


図 2 DOM Based XSS 攻撃の仕組み
Fig. 2 How DOM based XSS attack works.

© 2022 Information Processing Society of Japan

バには正規のURLのみがリクエストとして送信される形となっている（図2②）．リクエストを受けたWebサーバは，WebページのHTMLコードを被害者に送信する（図2③）．被害者のWebブラウザは，HTMLコードからWebページを動的に生成（レンダリング）する．その時点で，Webブラウザ内でアンカやクエリパラメータに含まれる有害なスクリプトが埋め込まれる（図2④）．この結果，被害者のWebブラウザ上で有害なスクリプトが実行されてしまう．

本論文では，レンダリング後のHTMLコードを「レンダードHTMLコード」と呼ぶ．Reflected XSS攻撃やStored XSS攻撃は，バックエンド（Webサーバ上）でHTMLコードが動的生成される際に有害スクリプトが表出するため，API Gatewayにおけるセキュリティ機構でその検知・対策が可能である（図2③）．しかしDOM Based XSS攻撃は，フロントエンド（Webブラウザ内）でレンダードHTMLコードが動的生成される際に有害スクリプトが表出するため，API Gatewayによる救済が不可能である．

3.2 Content Security PolicyによるDOM Based XSS対策

Webアプリケーションは，入力パラメータ（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Webページが動的に変化する．このWebページの動的生成の過程で，Webアプリケーション開発者の意図しないスクリプトがWebページに埋め込まれることがXSS攻撃の実体である．したがって，Webページ内の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その実行を許可することによって，XSS攻撃を防止できる．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の実行がすべて禁止されるため，多様化した有害スクリプトやゼロデイ脆弱性に対しても，頑健なXSS対策といえる．

Content Security Policy（CSP）[14]は，モノリシックアーキテクチャで開発されたWebアプリケーションにおける一般的なXSS対策の1つであり，ポリシベースのホワイトリスト対策である．CSPにおけるDOM Based XSS対策としては，Trusted Types [4], [18]というディレクティブがあげられる．Trusted Typesを用いて，DOM操作を行う関数が満たすべきセキュリティポリシを定義することにより，Trusted Typesオブジェクト（当該ポリシを満たしたオブジェクト）のみにDOM操作が許可される．これにより，セキュリティポリシから逸脱するDOM操作を禁止することができる．

しかし，MFEにおいてTrusted Typesを適用することを試みた場合，独立して開発されるマイクロサービスごとに異なるセキュリティポリシが実装されうる．この結果，種々のマイクロサービスを統合して新規サービスを開発する際に，セキュリティポリシのコンフリクトが起こりうる．CSPはモノリシックなWebアプリケーションを想定して開発されているため，現在のところ，1つのWebアプリケーションの中にTrusted Typesに関する複数のポリシを併存させることができない．このため，マイクロサービス間のセキュリティの構築を行うためのセキュリティポリシを新たに定義し，これを新規サービスのセキュリティポリシの中に記載するという方法もとれない．また，TrustedTypesを実装しているマイクロサービスと，TrustedTypesの実装をしておらずinnerHTMLなどによってHTMLへ文字列を入力しているマイクロサービスが，1つのサービスとして組み上げられた場合，後者のマイクロサービス中のinnerHTMLはTrustedTypesオブジェクトを介していないため，文字列入力が禁止されてしまう．このように，従来のモノリシックアーキテクチャにおいてはDOM Based XSS攻撃に対する一般的な対策であったCSP（Trusted Typesディレクティブ）は，MFEにおいては有効的な対策とはいえない．

3.3 API GatewayによるDOM Based XSS対策

MSAにおけるXSS対策は，API Gatewayによる対策が一般的である[6]．Amazon Web Serviceが提供しているAmazon API Gateway [19]は，API Gatewayの一種であり，基本的な機能は，クライアントからのリクエストを受け取り，マイクロサービスにルーティングを行うことである．この際，AWS WAF [20]やAWS Lambda [21]と連携することで，SQLインジェクションやXSSなどの様々なインジェクション攻撃から，Webアプリケーションを保護することができる．しかし，検知できるインジェクション攻撃は，API Gatewayを経由する攻撃に限るため，3.1節で述べたように，フロントエンド（Webブラウザ内）で攻撃が完結してしまうようなDOM Based XSS攻撃に対しては無力である．

3.4 テイント解析によるDOM Based XSS対策

Wangらは，ブラウザによるWebページのレンダリングの際に，Webページに含まれるDOM Based XSS攻撃の原因となりうる脆弱箇所を自動で検証することによって，DOM Based XSS攻撃を防止する手法を提案した[22]．しかし，この手法は，テイント解析の実行のために，JavaScriptエンジンとレンダリングエンジンを改造する必要がある．

3.5 テストベースホワイトリストによるDOM Based XSS対策

ポリシベースホワイトリストによる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

© 2022 Information Processing Society of Japan

的にセキュリティポリシが作成されていた．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．Web アプリケーションの複雑化にともない，仮に十分なセキュリティ知識を有した開発者であっても，セキュリティポリシの設定にエラーが混入する可能性が無視できない [13]．MFE 型の Web アプリケーションにおいては，マイクロサービス間のセキュリティポリシのコンフリクトにも対処しなければならないため，問題はさらに深刻となる．

3.2 節の冒頭で，Web アプリケーション開発者の意図したスクリプトのみを許可することによって，XSS 攻撃の防止が可能であることを述べた．しかし，上述のように，従来のポリシベースのホワイトリスト型対策においては，ホワイトリスト（セキュリティポリシ）の作成に難儀をかかえていた．この問題に対して，井上らは，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開発テストの結果に基づいてホワイトリストを作成する「テストベースホワイトリスト方式」を提案した [9]．Web アプリケーションの仕様書には，実装されるべきすべての機能と動作が示されている．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，Web アプリケーションのリリース前には，Web アプリケーションが仕様書どおりの動作をするか否かのテストが行われる．よって，Web アプリケーション開発の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，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．ホワイトリストは，開発テストを通じて自動的に生成される．

開発テストでは，仕様書に記載されているすべての入力パラメータ（ユーザからの入力やデータベース中の登録内容）を Web アプリケーションに 1 つ 1 つ入力し，Web ページの動作確認が行われる．テストベースホワイトリストでは，このテスト工程において生成されたレンダード HTML コードの「スクリプト構造（レンダード HTML コードからインラインスクリプト部分のみを抽出したもの）」がすべてホワイトリストとして登録される（文献 [9] の図 3）．本番環境で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（開発テストで検査済のパラメータが入力される限り），Web ブラウザ内で生成されるレンダード HTML 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．攻撃者によって，仕様から外れたパラメータ（開発テストにおいて未検査のパラメータ）が入力されると，Web ブラウザ内で生成されるレンダード HTML コードに，開発者の意図しないスクリプトが注入される．その結果，レンダード HTML 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わる（文献 [9] の図 4）．テストベースホワイトリスト方式は，このスクリプト構造の変化を用いて XSS 攻撃の発生を検知する．

MFE 型の Web アプリケーションの場合も，Web ブラウザ内で動的生成されるレンダード HTML コードのスクリプト構造が，DOM Based XSS 攻撃によって開発者の意図しないものへと変化する．このため，テストベースホワイトリストのコンセプト自体は MFE 型 Web アプリケーションの DOM Based XSS 対策としても有効である．ただし，MFE においては，Web アプリケーションを構成するすべてのモジュール（マイクロサービス）が，JavaScript 処理や Web コンポーネントを介してランタイム時にフロントエンド側（Web ブラウザ内）で統合される．その際，各マイクロサービスは非同期処理で読み込まれて統合されるため，Web ブラウザ内で動的生成されるレンダード HTML コードのスクリプト構造は，つねに一定の構造になるとは限らない．このため，仕様書に従う正常な入力パラメータであったとしても，開発テスト時にホワイトリストとして登録されたスクリプト構造と，本番環境のスクリプト構造とが異なりうる．よって，テストベースホワイトリストを MFE の DOM Based XSS 対策として活用するためには，非同期処理によるレンダード HTML コードの変動に対応する必要がある．4 章でその方法を詳述する．

4. 提案方式

本論文では，MFE 型 Web アプリケーションの設計に，テストベースホワイトリストのアプローチのセキュリティ対策を導入することで，個々のマイクロサービス開発の独立性を維持しながら，MFE においていまだ有効的な対策がなされていない DOM Based XSS 対策を実現する．以降で提案方式の具体的な設計（4 章）および実装（5 章）を行っていく．

4.1 非同期処理に対応したホワイトリスト

文献 [9] で提案されたテストベースホワイトリストは，モノリシック型 Web アプリケーションの XSS 対策を対象としていた．モノリシック型 Web アプリケーションにおいては，Web アプリケーション全体が一体成型されているため，アプリケーション内のトランザクションを同期処理型の手続きとして設計することが容易である．すなわち，Web ブラウザ内で動的生成されるレンダード HTML コードは基本的につねに同一となる．このため，レンダード HTML コードのスクリプト構造をホワイトリストの形式として採用することによって，効果的な XSS 攻撃の検知を実現していた．一方，MFE 型 Web アプリケーションにおいては，マイクロサービス間のトランザクションは非同期処理型の手続きとして設計される．すなわち，Web ブラウザ内で動的生成されるレンダード HTML コードが毎回変化しうる．このため，レンダード HTML コードのスクリプト構造をホワイトリストとして用いることは不適である．

© 2022 Information Processing Society of Japan

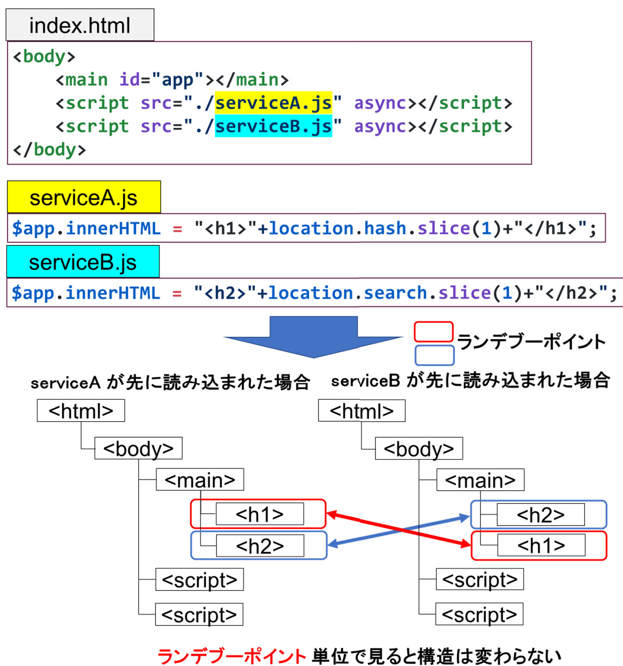


図 3 非同期処理による DOM ツリーの変化
Fig. 3 Changes in DOM tree due to asynchronous processing.

この問題を解決するために，本論文では，DOM Based XSS 攻撃によって埋め込まれる有害スクリプトの表出ポイントに着目する．HTML コードには，ユーザからの入力パラメータを受け取る「ソース（例：location.hash）」と，ソースを介して入力された文字列から JavaScript を生成する「シンク（例：document.write()）」が含まれる．DOM Based XSS 攻撃においては，攻撃者によって有害な文字列がソースに入力された結果，シンクによってレンダード HTML コード中に有害なスクリプトが注入される [23]．このため，Web ブラウザ内で HTML コードからレンダード HTML コードが動的生成される過程において，ソースの情報がシンクに引き渡されるポイント（以降，「ランデブーポイント」と呼ぶ）に有害スクリプトが表出しうる．

MFE においては各マイクロサービス（たとえば 2 つのマイクロサービス A と B）が非同期処理によって読み込まれるため，A と B が実際にどの順序で読み込まれたかに応じて，Web ブラウザ内で生成されるレンダード HTML コードが変化する．しかし，この場合のレンダード HTML コードの変化は，マイクロサービス A のコードブロックとマイクロサービス B のコードブロックが入れ替わるだけであり，個々のコードブロックの内部は不変である．したがって，たとえばマイクロサービス A のコードブロック中に含まれる「ランデブーポイントに紐づくソースの DOM ノード」のみに注目したならば，A と B のどちらが先に読み込まれたとしても，当該ランデブーポイントの HTML 要素は同一となる（図 **3**）．一方で，DOM Based XSS 攻撃によってランデブーポイントに有害なスクリプトが注入された場合，ランデブーポイントの HTML 要素のタグ構造（子ノード），属性，属性値のいずれか 1 つ以上が変化する（4.2 節で詳述する）．

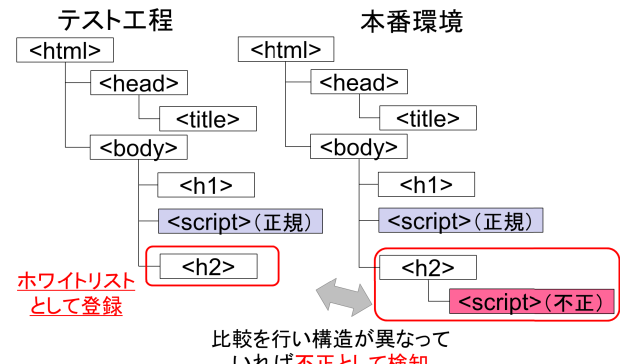


図 4 DOM Based XSS 攻撃の検知
Fig. 4 DOM based XSS attack detection.

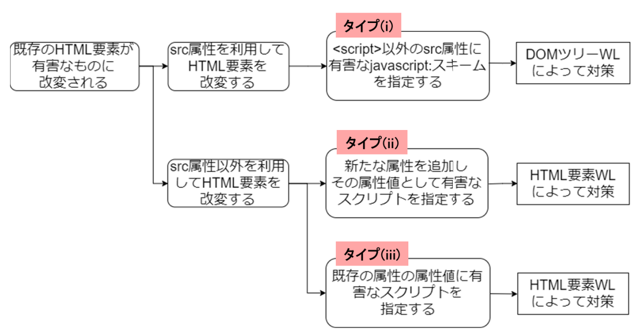


図 5 DOM Based XSS 攻撃の体系化
Fig. 5 Classification of DOM based XSS attacks.

そこで，テスト工程の Web ページ（レンダード HTML データ）に含まれるランデブーポイントの HTML 要素（タグ構造，属性，属性値）を記録し，これを「非同期処理にロバストなホワイトリスト」として使用する．本番環境の Web ページ（レンダード HTML データ）に含まれるランデブーポイントの HTML 要素（タグ構造，属性，属性値）が，そのつどホワイトリストと比較されることにより，DOM Based XSS 攻撃の検知が行われる．一例として，DOM Based XSS 攻撃によってタグ構造に変化が生じ，ホワイトリストとの非合致により攻撃発生と判定されるケースを図 **4** に示した．

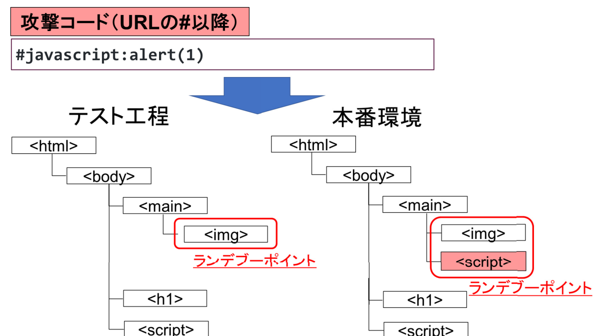
4.2 DOM Based XSS 攻撃の体系化と検知メカニズム

DOM Based XSS 攻撃には多様な攻撃方法が存在する [16], [17]．したがって，DOM Based XSS 攻撃の対策を考えるうえで，攻撃を体系化することが肝要である．これまでに，攻撃者によって制御可能な HTML 要素の領域の観点からの体系化 [16] や，有害なスクリプトの埋め込み方法の観点からの体系化 [17] が存在する．これらの体系化を基に，本論文では新たに，攻撃によって有害なスクリプトがどのような形で表出するかという観点から DOM Based XSS 攻撃の体系化を行った（図 **5** 中の (i)～(iii)）．

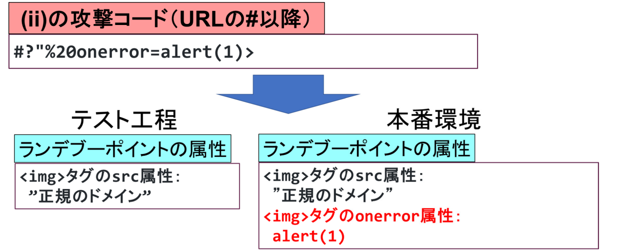
index.html

```
<html>
 <body>
  <main id="app"></main>
  <h1>example_i_and_ii</h1>
  <script>
   $app.innerHTML ='<img src="${decodeURIComponent(location.hash.slice(1))}">';
  </script>
 </body>
</html>
```

(a) HTML コード



(b) 攻撃タイプ(i)発生時のタグ構造の変化



(c) 攻撃タイプ(ii)発生時の属性情報の変化

図 6 攻撃タイプ (i) および (ii) の典型例
Fig. 6 Typical example of attack type (i) and (ii).

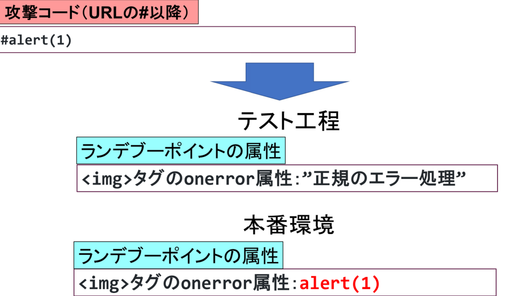
DOM Based XSS 攻撃は，既存の HTML 要素（開発者の意図する HTML 要素）が有害なものに改変されることで発生する．この HTML 要素の改変は，Web ページ内の既存の src 属性を利用して HTML 要素を改変するタイプと，src 属性以外を利用するタイプに大別される．このうち，src 属性を利用する攻撃は，<script> 以外のタグの src 属性に有害な javascript: スキームを指定するタイプ（図 5 の (i)）があげられる．また，src 属性以外を利用する攻撃は，Web ページ内の既存の HTML 要素に新たな属性を追加し，その属性値として有害スクリプトを指定するタイプ（図 5 の (ii)）と，既存の属性の属性値に有害スクリプトを指定するタイプ（図 5 の (iii)）に細別される．

攻撃タイプ (i) の典型例が図 **6** (b) である．開発テスト時には図 6 (b) 左であったランデブーポイントのタグ構造（子ノード）が，攻撃によって図 6 (b) 右に変化する．よって，テスト工程と本番環境の Web ページ（レンダード HTML データ）のランデブーポイントのタグ構造を比較することによって，DOM Based XSS 攻撃の検知が可能である．ランデブーポイントのタグ構造の変化は，DOM ツリーの構造比較によって検査できる．そこで，開発テスト時のランデブーポイントの DOM ツリー構造をホワイトリスト（以降，「DOM ツリーホワイトリスト」と呼ぶ）として登録する．

index.html

```
<html>
 <body>
  <main id="app"></main>
  <h1>example_iii</h1>
  <script>
   $app.innerHTML ='<img src="X" onerror="${location.hash.slice(1)}">';
  </script>
 </body>
</html>
```

(a) HTML コード



(b) 攻撃タイプ(iii)発生時の属性情報の変化

図 7 攻撃タイプ (iii) の典型例
Fig. 7 Typical example of attack type (iii).

攻撃タイプ (ii) の典型例が図 6 (c) である．開発テスト時には図 6 (c) 左であったランデブーポイントの属性と属性値が，攻撃によって図 6 (c) 右に変化する．よって，テスト工程と本番環境の Web ページ（レンダード HTML データ）のランデブーポイントの属性情報（属性および属性値）を比較することによって，DOM Based XSS 攻撃の検知が可能である．ランデブーポイントの属性・属性値の変化は，HTML 要素の構造比較によって検査できる．そこで，開発テスト時の HTML 要素の構造をホワイトリスト（以降，「HTML 要素ホワイトリスト」と呼ぶ）として登録する．

攻撃タイプ (iii) の典型例が図 **7** である．開発テスト時には図 7 (b) 左であったランデブーポイントの属性値が，攻撃によって図 7 (b) 右に変化する．HTML 要素ホワイトリストには属性値の情報も含まれているので，攻撃タイプ (ii) と同様の方法（HTML 要素ホワイトリスト）を用いることによって，DOM Based XSS 攻撃の検知が可能である．ただし，開発テストにおいては，代表的な入力パラメータのみを用いて Web ページの動作確認が行われることが一般的である．たとえば，drawing.jpg の表示サイズを大・中・小の 3 サイズから選択できる Web ページに対し，大サイズの画像を表示するケースのテストのみが行われたとしよう．テストベースホワイトリスト方式においては，開発テストの際に動作確認された大サイズ画像の情報のみがホワイトリストに登録されるため，中・小サイズ画像の情報がホワイトリストから漏れてしまう．この問題に対応するために，大サイズ画像に関するホワイトリストを正規表現で

© 2022 Information Processing Society of Japan

記述するなどの方法によって，中・小サイズ画像を含むホワイトリストへと拡張する．本論文においては，この部分の具体的な実装については今後の課題とする．

5. ホワイトリストの生成と検査

5.1 実装方針

4.1 節，4.2 節で説明したホワイトリストを実現するにあたっては，Web アプリケーションのランデブーポイントを特定したうえで，ランデブーポイントの HTML 要素を取得する必要がある．そこで提案方式では，第 1 に，HTML コードの中からランデブーポイントを特定し，特定されたランデブーポイントを HTML コード自体に埋め込むための処理（以降，「ランデブーポイント特定処理」と呼ぶ）を実行する．ランデブーポイント特定処理を通じて，ランデブーポイントが埋め込まれた HTML コード（以降，「ランデブー HTML コード」と呼ぶ）が生成される．HTML コードとランデブー HTML コードは，ランデブーポイントのマーカが付与されていること以外は同一である（5.2 節で詳述する）ため，HTML コードの代わりにランデブー HTML コードを Web アプリケーションの実体として取り扱うことができる．

この結果，テスト工程および本番環境においては，ランデブー HTML コード中のマーカを手がかりに，ランデブーポイントの HTML 要素を簡便に取得可能となる．テスト工程では，Web アプリケーションに対して仕様書に基づく動作確認を通じ，ホワイトリストを生成する処理（以降，「ホワイトリスト生成処理」と呼ぶ）を実行する．本番環境では，ホワイトリストとの照合を行うことによって DOM Based XSS 攻撃の有無を判定する処理（以降，「攻撃判定処理」と呼ぶ）を実行する．

以降，5.2～5.4 節でそれぞれ，ランデブーポイント特定処理，ホワイトリスト生成処理，攻撃判定処理の手順を詳述する．また，5.5 節では提案方式の実装について説明する．

5.2 ランデブーポイント特定処理

HTML コードの中からランデブーポイント（ソースの情報がシンクに引き渡されるポイント）を特定する処理（ランデブーポイント特定処理）は，HTML コードからレンダード HTML コードを生成する工程で構築される DOM ツリーをパースすることで実装可能である．具体的な手順を以下に示す．

1. HTML コード（レンダリング前の HTML コード）に対し，ランデブーポイントの候補となりうる DOM ノードに一意の識別子（以降，「ノード ID」と呼ぶ）を付与する．具体的には，HTML コードを 1 行ずつパースし，シンクとして機能する予約語 [24] が含まれていた場合には，当該コマンドラインに <div id=nodeID> タグを挿入する（図 **8** の (a)→(b)）[*1]．ここで，nodeID は DOM ノード識別のために生成された乱数である．以降，ノード ID が付与された HTML コードを「マークド HTML コード」と呼ぶ．

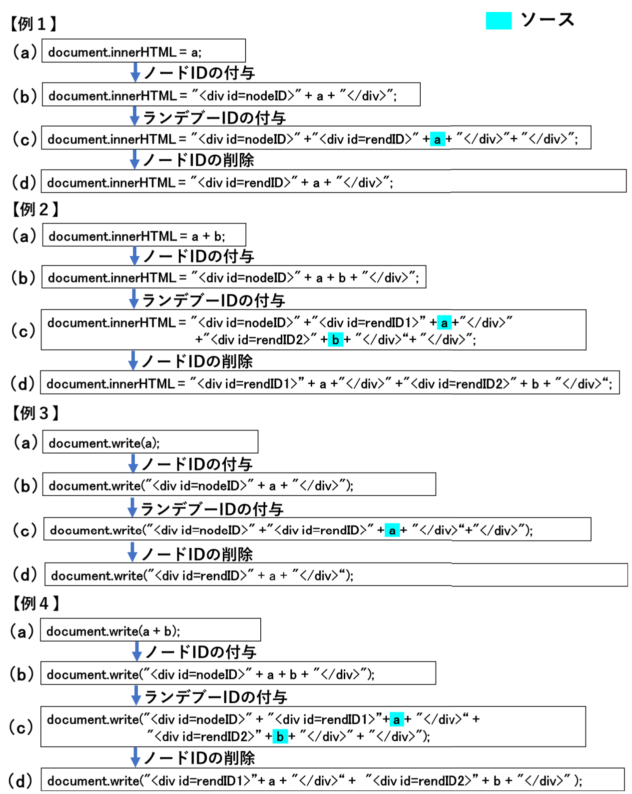


図 8　HTML コードに対する ID の操作
Fig. 8　Operate on ID against HTML code.

2. マークド HTML コードを Web ブラウザで読み込む．window.onload() イベントを利用し，Rendering フェーズ完了の時点（Web ブラウザ内で WEB ページがレンダリングされたタイミング）で，DOMParser インタフェースを介して，レンダード HTML コード中の全 DOM ノードの要素を取得する．手順 1 で付与したノード ID（<div id=nodeID> タグ）により，マークド HTML コード内の各コマンドラインと DOM ツリー内の当該ノードが紐付けされている（図 **9**）．
3. ノード ID（<div id=nodeID> タグ）が付与されている DOM ノードを起点として（必要に応じて）親ノード・子ノードをたどることによって，手順 2 で取得した全 DOM ノードの情報の中から <div id=nodeID> タグによって囲まれた要素を発見する．<div id=nodeID> タグによって囲まれた要素の内容を検査し，ソースとして機能する予約語 [24] が含まれていたならば，ここ

[*1] ランデブーポイントは次の 4 つの形態に分けられる．図 8 の例 1：1 つのソースが代入操作によってシンクへ渡される場合．図 8 の例 2：2 つ以上のソースが代入操作によってシンクへ渡される場合．図 8 の例 3：1 つのソースが引数としてシンクへ渡される場合．図 8 の例 4：2 つ以上のソースが引数としてシンクへ渡される場合．

© 2022 Information Processing Society of Japan

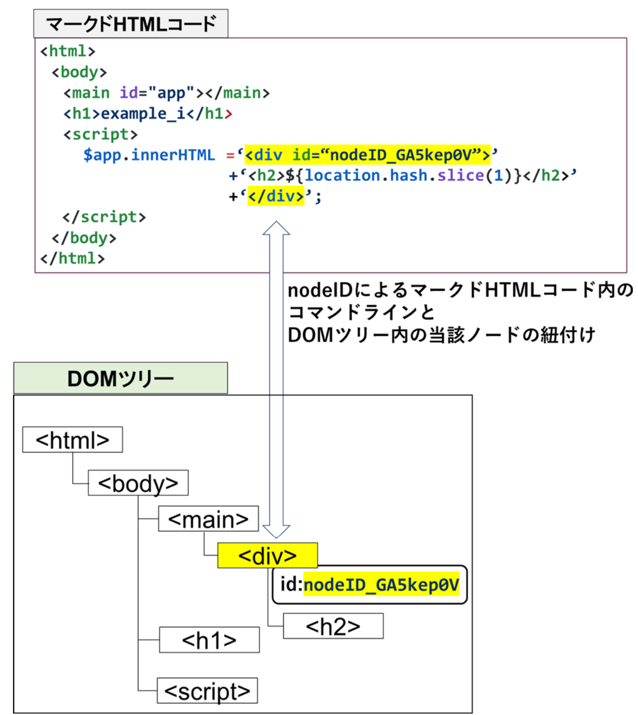


図 9　nodeID による紐付け
Fig. 9　Linking by nodeID.

がランデブーポイントである[*2]．

4. マークド HTML コードに対し，手順 3 で特定されたランデブーポイントに一意の識別子（以降，「ランデブー ID」と呼ぶ）を付与する．具体的には，マークド HTML コード内のノード ID（<div id=nodeID> タグ）を起点として（必要に応じて）前後のコマンドラインをたどることによって，手順 3 で特定されたソースを発見し，当該箇所に <div id=rendID> タグを挿入する（図 8 の (b)→(c)）．ここで，rendID はランデブーポイント識別のために生成された乱数である．
5. 手順 4 のマークド HTML コードから，手順 1 で付与したノード ID（<div id=nodeID> タグ）を削除する（図 8 の (c)→(d)）．
6. この結果，HTML コード内のランデブーポイントの発見と，HTML コードに対するランデブーポイントの挿入が完成する．以降，ランデブー ID が付与された HTML コードを「ランデブー HTML コード」と呼ぶ．

HTML コードとランデブー HTML コードの違いは，ランデブーポイントの位置情報が <div id=rendID> タグによって付与されているか否かである．すなわち，Web ブラウザで HTML コードを表示した場合の Web ページと，ランデブー HTML コードを表示した場合の Web ページは基本的に同一となる．このため，HTML コードの代わりに，ランデブー HTML コードを Web アプリケーションの実体として Web サーバに搭載しても，実害は生じない．ランデブーポイント特定処理（上記の手順 1〜6）の実行によって，HTML コードからランデブー HTML コードが自動生成されるので，Web アプリケーション開発者はこれまでどおりの設計・開発を通じて HTML コードを作成すればよい．ユーザが Web アプリケーションを利用するにあたってのユーザインタラクションにも変化はない．

5.3　ホワイトリスト生成処理

提案方式では，Web サーバに搭載される Web アプリケーションの実体がランデブー HTML コードとなっている．Web ブラウザによってランデブー HTML コードが読み込まれると，Web ブラウザ内で Web ページがレンダリングされ，その過程で DOM ツリーが構築される．ランデブー HTML コードにはランデブーポイントの位置が <div id=rendID> タグとして埋め込まれている．よって，rendID を手がかりに，DOMParser インタフェースを使ってランデブーポイントに対応する DOM ノードの HTML 要素を取得することができる．

テスト工程においては，仕様書に基づいて，Web アプリケーションに入力すべきすべての入力パラメータがテストケースとして用意されている．そして，「テストケースに含まれる入力パラメータを Web アプリケーションに 1 つ 1 つ入力し，Web ブラウザ内で動的生成される Web ページの動作が仕様書どおりであるか確認する」という検査がすべてのテストケースに対して繰り返される．

Web アプリケーションの開発テストでは，仕様書に基づいて，Web アプリケーションの動作確認が実施される．テストベースホワイトリスト方式では，テスト工程で確認された Web ページをホワイトリスト（正常な Web ページ）として登録する．下記の手順によって，開発テストを通じてホワイトリストが自動的に生成される．

1. テストケースに含まれる 1 番目の入力パラメータを用いて，Web アプリケーションにリクエストを送信する．3.1 節で説明したとおり，Web サーバから Web ブラウザにランデブー HTML コード[*3]が届き（図 2 ③），Web ブラウザ内で入力パラメータが適用される（図 2 ④）．
2. window.onload() イベントを利用し，Rendering フェーズ完了の時点で，DOMParser インタフェースを介して，レンダード HTML コード（レンダリング後のランデブー HTML コード）中の全 DOM ノードの要素を取得する．5.2 節の「ランデブーポイント特定処理」に

[*2] 手順 1 において，シンクに対してのみノード ID が付与されていることに注意されたい．手順 3 において，ノード ID が付与された DOM ノード（シンク）の中からソースを発見することにより，ノードの情報がシンクに引き渡されたポイントを特定することができる．

[*3] 5.2 節で説明したとおり，提案方式では，Web アプリケーションの実体としてランデブー HTML コードが Web サーバに搭載されているため，図 2 ③ における「HTML コード」が，ここでは「ランデブー HTML コード」に置き換わっている．

© 2022 Information Processing Society of Japan

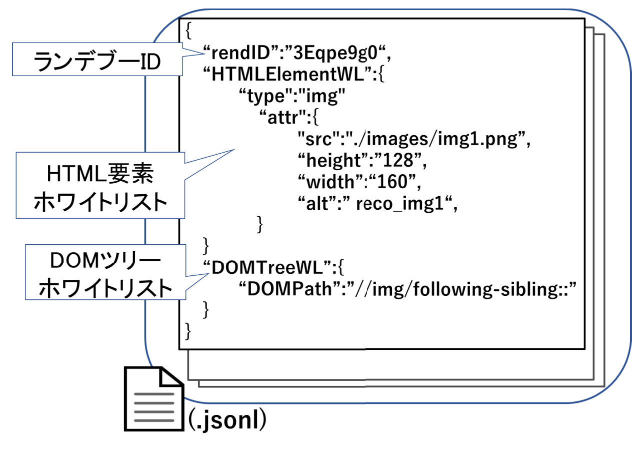


図 10　ホワイトリストの書式
Fig. 10　Document format for whitelist.

よって付与されたランデブー ID（<div id=rendID> タグ）により，ランデブー HTML コード内のランデブーポイントに対応する DOM ノードを発見することができる．

3. ランデブー HTML コード中のすべてのランデブーポイントの HTML 要素を取得する．4.1 節で説明したとおり，HTML 要素のタグ構造（子ノード），属性，属性値をホワイトリストとして登録する．具体的には，4.2 節で説明したとおり，タグ構造を DOM ツリーホワイトリストに，属性・属性値を HTML 要素ホワイトリストに，それぞれ登録する*4．図 10 にホワイトリストの一例を示す．タグ構造は XPath [25] での記述とした．ホワイトリストは，ランデブー ID ごとに HTML 要素ホワイトリストと DOM ツリーホワイトリストを対応させたものを JSON 形式の 1 レコードとして扱い，全体を JSONL 形式とした．
4. テストケースに含まれる 2 番目の入力パラメータに対し，手順 1～手順 3 を実行する．すべての入力パラメータに対し，これを繰り返す．

5.4　攻撃判定処理

本番環境においても，テスト工程と同様の手順で，ランデブーポイントに対応する DOM ノードの HTML 要素を取得することができる．ただし，本番環境においては，Web ブラウザ内で Web ページに記述された JavaScript 内のランデブーポイントが解析，実行された時点で DOM Based XSS 攻撃が実行されてしまう．このため，Web ブラウザのデバッグ機能を利用して Scripting フェーズに介入し，ランデブーポイントが解析，実行されるよりも前の段階で，Web ページからランデブーポイントの HTML 要素を抽出し，ホワイトリストとの照合を実行する．

5.3 節の「ホワイトリスト生成処理」によって生成されたホワイトリストには，仕様書に基づいて動作確認されたすべての Web ページの情報が登録されている．このため，本番環境のランデブーポイントの HTML 要素がホワイトリストに含まれていない場合には，想定外の入力パラメータ（未テストの入力パラメータ）が入力されたのだと判断できる．テストベースホワイトリスト方式では，これを DOM Based XSS 攻撃として検知する．この手順を以下に記す．

1. 本番環境で Web アプリケーションが利用されると，Web サーバから Web ブラウザにランデブー HTML コードが届き（図 2 ③），Web ブラウザ内で入力パラメータが適用される（図 2 ④）．
2. chrome.debugger [26] を利用し，ランデブー HTML コード中のランデブーポイント（<div id=rendID> タグ）を発見し，すべてのランデブーポイントにブレークポイントを設定する．
3. Scripting フェーズにおいて，ブレークポイント到達するたびに，Acorn.js [27] を用いてランデブーポイントの HTML 要素を取得する．
4. 手順 3 で取得した HTML 要素をホワイトリストと照合する．ホワイトリストに登録されている HTML 要素のいずれとも合致しない場合は，DOM Based XSS 攻撃であるとして検知する．

5.5　実装

提案方式による DOM Based 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 4 つの機能が必要となる．1 つ目に，HTML コードからランデブー HTML コードを自動生成する機能，2 つ目に，Web アプリケーションの開発テストを通じてホワイトリストを自動生成する機能，3 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，4 つ目に，Web ブラウザ内で動的生成されるレンダード HTML コードをホワイトリストと比較検査する機能である．このうち，1 つ目と 2 つ目がテスト工程で実行される機能であり，3 つ目と 4 つ目が本番環境で実行される機能である．

文献 [9] では，テスト工程で実行される機能を，Selenium [28] 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ている．また，本番環境で実行される機能については，Firefox 用のブラウザアドオンとして実装している．本論文においても，文献 [9] と同様の方針で提案方式を実装した．ブラウザアドオンは，ユーザが自身の Web ブラウザにインストールして利用してもらうことが前提となる．

1 つ目の機能については，ランデブーポイント特定処理（5.2 節）を実行するプログラムを作成して Selenium にアドオンした．2 つ目の機能については，ホワイトリスト生成処理（5.3 節）を実行するプログラムを作成して Selenium

*4 src 属性に有害な javascript: スキームが指定される図 5 の攻撃タイプ (i) については，HTML 要素ホワイトリストの属性・属性値を検査することによっても検知可能である．

© 2022 Information Processing Society of Japan

にアドオンした．3 つ目の機能（以降，Web アプリケーションとホワイトリストの「バインド処理」と呼ぶ）については，文献 [9] と同じプログラムを作成して Web ブラウザにアドオンした．4 つ目の機能については，攻撃判定処理（5.4 節）を実行するプログラムを作成して Web ブラウザにアドオンした．なお，今回は 1 つ目と 2 つ目の両方の機能を 1 つの Selenium 用アドオンの形で実装している．Selenium 用アドオンは，ランデブーポイント特定処理の実行によってランデブー HTML コードを，ホワイトリスト生成処理の実行によってホワイトリストを生成する．また，3 つ目と 4 つ目の機能については，両者を 1 つの Chrome 用ブラウザアドオンの形で実装している．

6. 評価

6.1 対象 Web アプリケーション

文献 [29] を基に MFE 型 Web アプリケーションを開発し，これを，今回の評価対象の Web アプリケーションとして用いる．今回は，簡易的なオンラインショップ（以下，EC サイト）を模した Web アプリケーションを実装した．

EC サイトのビューを図 **11** に示す．この EC サイトは，「商品選択サービス（図 11 オレンジ枠部分）」，「ボタンサービス（図 11 青枠部分）」，「カートサービス（図 11 ピンク枠部分）」，「レコメンドサービス（図 11 緑枠部分）」の 4 つのマイクロサービスから構築されている．そして，これら 4 つのマイクロサービスを統括し，Web アプリケーション全体を EC サイトたらしめているのが「統合サービス」である．個々のマイクロサービスは EC サイトの開発者にとってはブラックボックスであり，EC サイトの開発者が設計・実装するのは 4 つのマイクロサービスを利用（参照）しながら全体の機能を提供する統合サービスのみとなる．今回は，個々のマイクロサービスを，Vagrant [36] を用いて立ち上げた個別のサーバ上で稼働させており，複数の Web サイト間で各マイクロサービスが呼び出されながら統合サービスが運用されている．

各マイクロサービスと統合サービスの EC サイトの仕様書（機能，インタフェース（入力値，出力値），セキュリティポリシ）を以下に概説する．また，マイクロサービス間のインタフェースの接続関係を図 **12** に示す．

【商品選択サービス】

- 入力値：商品カテゴリ（URL のアンカ（「#」以降の値）として与えられる）．
- 機能：location.hash により商品カテゴリ（item_category）を取得．当該カテゴリに含まれるすべての商品のサムネイル画像と商品名を陳列棚に表示．陳列棚の商品をユーザが選択すると，選択された商品のオリジナル画像を表示．
- 出力値：選択された商品のカテゴリ内商品番号（categorized_item_NO）．
- セキュリティポリシ：CSP の TrustedTypes ディレクティブの createHTML メソッドによって，入力値に対してエスケープ処理を適用．

【ボタンサービス】

- 入力値：商品 ID（item_ID）．
- 機能：当該商品の売価を購入ボタンに表示．ボタンが押されたら，イベントの発生を通知する信号（button_event）を出力．
- 出力値：イベント通知（button_event），商品 ID（item_ID）．
- セキュリティポリシ：CSP の TrustedTypes ディレクティブの createHTML メソッドによって，入力値に対してエスケープ処理を適用．

【カートサービス】

- 入力値：イベント通知（button_event），商品 ID（item_ID）．
- 機能：イベント通知を受け取ったら，当該商品をカートに追加する．カートアイコン上にカート内の商品点数（cart_item_num）を表示．
- 出力値：なし．
- セキュリティポリシ：CSP のフェッチディレクティブの img-src によって，入力値（画像の参照元である商品 ID）をホワイトリストとして列挙し，制御する．

【レコメンドサービス】

- 入力値：商品カテゴリ（URL のアンカ（「#」以降の

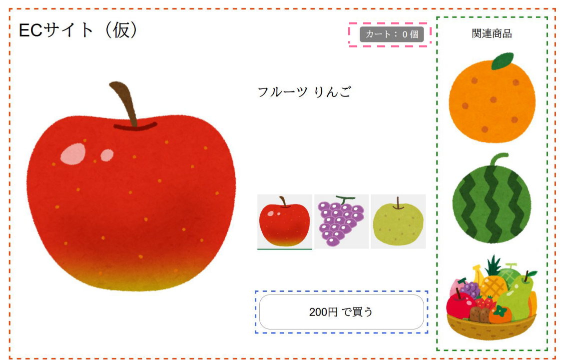


図 **11**　評価用 EC サイトのビュー
Fig. 11　View of e-commerce site for evaluation.

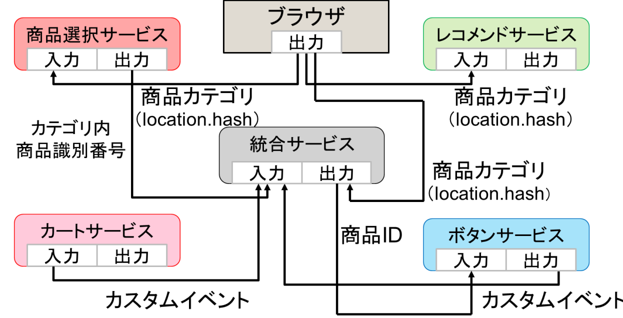


図 **12**　EC サイト中のマイクロサービス間の関係
Fig. 12　Relationships between microservices in EC site.

値）として与えられる），カテゴリ内商品番号（URL のクエリパラメータ（「?」以降の値）として与えられる）．
- 機能：location.hash により商品カテゴリ (item_category) を取得．location.search によりカテゴリ内商品番号 (categorized_item_NO) を取得．商品カテゴリとカテゴリ内商品番号を連結することにより，商品 ID (item_ID = item_category＋categorized_item_NO) を生成．当該商品の関連商品を表示．
- 出力値：なし．
- セキュリティポリシ：CSP のフェッチディレクティブの img-src によって，入力値（画像の参照元である商品カテゴリ）をホワイトリストとして列挙し，制御する．

【統合サービス】
- 入力値：商品カテゴリ（URL のアンカ（「#」以降の値）として与えられる），カテゴリ内商品番号（商品選択サービスから与えらえる），イベント通知（ボタンサービスから与えられる）．
- 機能：location.hash により商品カテゴリ (item_category) を取得．商品選択サービスからカテゴリ内商品番号 (categorized_item_NO) を取得．商品カテゴリとカテゴリ内商品番号を連結することにより，商品 ID (item_ID = item_category＋categorized_item_NO) を生成しボタンサービスに通知．ボタンサービスからのイベント通知（button_event）をカートサービスに転送．
- 出力値：商品 ID（item_ID），イベント通知 (button_event)．
- セキュリティポリシ：開発テストを通じてテストベースホワイトリストを自動生成（5.2 節，5.3 節）．

今回の EC サイトにおいては，商品選択サービスとレコメンドサービスが 2.2 節の図 1 の説明した関係（商品選択サービスが図 1 のマイクロサービス A，レコメンドサービスが図 1 のマイクロサービス B）をなしている．このため，商品選択サービスのセキュリティポリシ（入力値に対するエスケープ処理）とレコメンドサービスのセキュリティポリシ（入力値に対するホワイトリスト型検査）の間に，セキュリティポリシのコンフリクトが生じうる（**図 13**）．また，ボタンサービスとカートサービスも同様の関係をなしており，両者のセキュリティポリシの間にも図 1 のコンフリクトが生じうる構成となっている（**図 14**）．したがって，統合サービス（EC サイト）においては，各マイクロサービスをそのままの形で（セキュリティポリシを変更することなく）部品化することができない．

提案方式であれば，個々のマイクロサービスのセキュリティポリシの代わりに，EC サイト開発のテスト工程を通じて自動生成したホワイトリストを用いることによって，本番環境のセキュリティ確保が可能である．このため，各

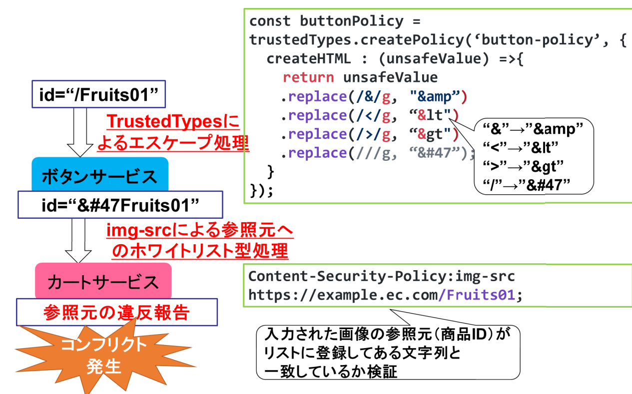


図 13　商品選択サービスとレコメンドサービス間におけるセキュリティポリシのコンフリクト

Fig. 13　Security policy conflicts between product selection services and recommendation services.

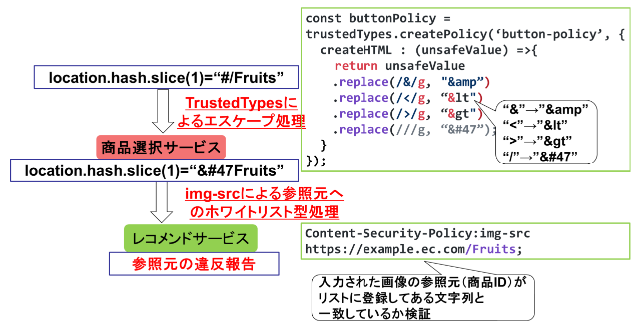


図 14　ボタンサービスとカートサービス間におけるセキュリティポリシのコンフリクト

Fig. 14　Security policy conflicts between button services and cart services.

マイクロサービスのセキュリティポリシ（CSP のディレクティブ）を取り外すことによってこれらを部品化し，統合サービス（EC サイト）の中で使用（参照）することが可能である．今回の EC サイトは，実際にそのような方法で実装を行った．

今回の EC サイトには，4.2 節で説明した DOM Based XSS 攻撃のタイプ (i)～(iii) の脆弱箇所を実装してある．また，それぞれの脆弱箇所は，5.2 節の図 8 の 4 種類のランデブーポイントの形態で実装されている．すなわち，今回の EC サイトによって，起こりうる DOM Based XSS 攻撃のすべてのタイプの脆弱性（攻撃タイプ 3 種 × ランデブーポイントの形態 4 種 = 計 12 種）を，網羅的に評価することが可能になっている．

6.2　ホワイトリスト生成実験

5.5 節で実装したランデブーポイント特定処理とホワイトリスト生成処理をアドオンした Selenium を用い，Web アプリケーション（6.1 節で説明した EC サイト）開発の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．

© 2022 Information Processing Society of Japan

表 1　EC サイトの仕様書とテストケース 1
Table 1　Specification and test-cases-1 for EC site.

仕様	テストアプローチ		入力操作	期待する出力
	テスト対象	テスト観点		
商品名や商品画像に応じてボタンに表示される価格が変化	商品選択サービス，ボタンサービス	商品選択サービスで発行された商品IDに応じてボタンサービスの	商品選択サービスにて全商品画像を選択	Fruits01は200が表示
				Fruits02は300が表示
				Fruits03は400が表示
商品名や商品画像に応じて関連商品の画像が変化	商品選択サービス，レコメンドサービス	商品選択サービスとレコメンドサービスの連動	商品選択サービスにて全商品画像を選択	Fruits01はimg1,img2,img3
				Fruits02はimg3,img4,img5
				Fruits03はimg1,img4,img6
購入ボタンを押すとカートの中身が増える	ボタンサービス，カートサービス	ボタンサービスとカートサービスの連動	「○○円で買うボタン」を1回押す	cart_item_numが1
			「○○円で買うボタン」を5回押す	cart_item_numが5
			「○○円で買うボタン」を6回押す	エラーメッセージ出力

表 2　EC サイトの仕様書とテストケース 2
Table 2　Specification and test-cases-2 for EC site.

テストアプローチ		入力操作	期待する出力
テスト対象	テスト観点		
商品選択サービス	全商品画像の表示	商品選択サービスにて全商品画像を選択	選択した商品IDが出力されるか
レコメンドサービス	全商品IDに対する関連商品の表示	product_keyに対してFruits01	reco = [img1, img2, img3]
		product_keyに対してFruits02	reco = [img3, img4, img5]
		product_keyに対してFruits03	reco = [img1, img4, img6]
ボタンサービス	全商品IDに対する値段の表示	product_keyに対してFruits01	Price_Fruits01 = 200
		product_keyに対してFruits02	Price_Fruits02 = 300
		product_keyに対してFruits03	Price_Fruits03 = 400
カートサービス	全商品IDに対する商品名と値段の表示	product_keyに対してFruits01	商品名：りんご
			価格：200
		product_keyに対してFruits02	商品名：ぶどう
			価格：300
		product_keyに対してFruits03	商品名：梨
			価格：400

Selenium は，Web ブラウザのオートメーションツールであり，自動で Web ブラウザを操作することで Web サイトの動作のテストを行う．Selenium では，テストを実施する開発者が「EC サイトに与える入力パラメータ（入力データ）」と「その入力に対し生成される EC サイトの Web ページ（正解データ）」のペアからなる検査項目を，テストケースとして用意する．すべての検査項目に対し，「EC サイトに入力データを入力 → その結果生成された Web ページを正解データと比較」というテスト手順を繰り返すことによって，EC サイトの動作を自動検査する．

今回の EC サイトのテストケースを**表 1** および**表 2** に示した．表 1 には，統合サービスが各マイクロサービスを単体使用して提供する機能に対するテストケースがまとめられている．表 2 には，統合サービスが複数のマイクロサービスを連携使用して提供する機能に対するテストケースがまとめられている．MFE 型の開発工程におけるテストは，単体テスト，サービステスト，統合テスト（UI テスト）などから構成される [30]．サードパーティ製のマイクロサービスを組み合わせて 1 つのサービスを構築する開発においては，すでに個々のマイクロサービス単位でテストが行われているため，統合テストが重点的に行われると考えられる．表 1 および表 2 は，統合テストにおけるテストケースを想定したものである．

Selenium による開発テストを実施し，ホワイトリストを得た．これらのテストケースに基づき，開発者（著者ら）がその内容を点検した結果，「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．今回，生成されたホワイトリストの数は 216 個であった．

6.3　検知実験

5.5 節で実装したバインド処理と攻撃判定処理をアドオンした Chrome 用ブラウザアドオン（以降，当該アドオンを実装した Web ブラウザを「検査用ブラウザ」と呼ぶ）が，Web アプリケーション（6.1 節で説明した EC サイト）に対する DOM Based XSS 攻撃を検知できることを確認する．

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．6.2 節のホワイトリスト生成実験の際に，ランデブーポイント特定処理を通じて EC サイトの HTML コードからランデブー HTML コードが，ホワイトリスト生成処理を通じて EC サイトのホワイトリストが，それぞれ生成されている．このランデブー HTML コードとホワイトリストを仮想マシン上の Web サーバに搭載したうえで，実験用 PC 上の検査用ブラウザから当該 EC サイトにアクセスする．

攻撃コードに関しては，4.2 節で述べた DOM Based XSS 攻撃のタイプ (i)〜(iii) に該当する 3 個の典型的な攻撃コードを用いて実験を行った．攻撃コード注入の結果，今回の EC サイトに埋め込まれている 12 個（攻撃タイプ 3 種 × ランデブーポイントの形態 4 種）の脆弱箇所のすべてにおいて，提案方式による DOM Based XSS 攻撃の検知が可能であったことを確認した．

6.4　パフォーマンス実験

テストベースのホワイトリストにおいては，開発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検査に関するオーバヘッドを確認する．

パフォーマンス実験は，検知実験と同じく，実験用 PC 上

© 2022 Information Processing Society of Japan

表 3 実験用 PC の諸元
Table 3 Specifications of experimental PC.

OS	Windows10 Pro
CPU	Intel i9-10900K
メモリ	16GB
Webブラウザ	Google Chrome version 98.0.4758.82

に Web サーバ役の仮想マシンを VirtualBox で構築し，今回の EC サイトのランデブー HTML コードとホワイトリストを搭載して行った．本実験ではフロントエンドにおける処理速度を測るため，バックエンド側に仮想マシンを配置することが，計測結果に対して影響を及ぼすことはない．実験用 PC 上の検査用ブラウザから，EC サイトへのアクセスを行い，リクエストからレンダリング完了までの所要時間を DevTools [32] を用いて計測する．実験用 PC の諸元を**表 3** に示す．なお，DevTools のネットワーク通信量を基に EC サイトのデータ容量を計測したところ，320 kB であった．

ホワイトリスト検査に要するオーバヘッドを確認するため，提案方式を適用した検査用 Web プラットフォームと，従来の一般的な Web プラットフォームのそれぞれの環境において速度の計測および比較を行う．また，HTML コードやホワイトリストのボリュームによってオーバヘッドがどの程度変化するか調べるために，HTML 要素のボリュームを 2 倍にした HTML コードを用いた環境と，ホワイトリストのボリュームを 2 倍にした環境をそれぞれ用意*5し，同様の計測を行う．以上の 4 種類のそれぞれにおいて，EC サイトアクセスの所要時間の計測を 3 回ずつ行う．なお，Web ブラウザのキャッシュ，Cookie，および履歴などの計測結果に影響を与えうる情報は，計測ごとに消去する．

5.4 節で述べたように，提案方式においては，Web ブラウザのデバッグ機能を利用して Scripting フェーズに介入し，Acorn.js を用いてランデブーポイントの HTML 要素を逐一取得する．そこで，今回のオーバヘッドの測定にあたっては，本番環境（ホワイトリスト検査）での Scripting フェーズ完了までのタイムを計測する．

検査なし環境（**表 4** 中の α），検査あり環境（表 4 中の β），HTML 要素 2 倍環境（表 4 中の γ），ホワイトリスト 2 倍環境（表 4 中の δ）の実験結果を，それぞれ表 4 に示す．検査なし環境と検査あり環境の所用時間の比較より，提案方式の適用によって約 3 倍のオーバヘッドが生じることが判明した．また，検査あり環境と HTML 要素 2 倍環境およびホワイトリスト 2 倍環境の所用時間の比較より，HTML 要素数あるいはホワイトリスト数が倍になると，オーバヘッドはそれぞれ約 1.4 倍，約 1.3 倍になる（HTML コードやホワイトリストが 2 倍になってもオーバヘッドが 2 倍に増加するわけではない）ことが判明した．MFE 型アプリケーションの開発において，デプロイを重ねるごとに HTML コードやホワイトリストのボリュームが増加する傾向にあると考えられるが，提案方式においては，それらの増加にともなうオーバヘッドの増加が抑えられる可能性が認められた．

*5 この実験の目的はオーバヘッドの比較であり，検知精度の測定は不要であることを考慮し，今回は，HTML 要素数については，レンダード HTML コードを単純に複製して冗長化したうえで，アプリケーションとして成立するために適切な補正を施すことによって，HTML 要素数が 2 倍の HTML コードを生成した．同様に，ホワイトリストについては，6.2 節の実験を通じて生成されたホワイトリストの全エントリを複製することによって，リスト数が 2 倍のホワイトリストを生成した．

表 4 各環境における計測結果
Table 4 Measurement results.

環境	回数	所用時間	環境	回数	所用時間
α	1回目	61ms	β	1回目	185ms
	2回目	89ms		2回目	221ms
	3回目	80ms		3回目	238ms
γ	1回目	305ms	δ	1回目	270ms
	2回目	301ms		2回目	299ms
	3回目	316ms		3回目	273ms

6.5 提案方式の制限と今後の課題

テストベースホワイトリストの本質は，開発者が意図する動作のみを許可することである．このため，開発テストにおけるテストパターンのカバレッジが低い場合には，提案方式の誤検知・検知漏れが発生する可能性が高まる．

この問題は，開発対象の Web アプリケーションの規模が大きくなる場合や，動的コンテンツが多用される場合に顕著になる．HTML 要素のホワイトリストを適切にワイルドカード化（4.2 節の攻撃タイプ (ii) に対するホワイトリスト）あるいは正規表現化（4.2 節の攻撃タイプ (v) に対するホワイトリスト）することが，この問題を緩和する対策となる．しかし，「ホワイトリスト中の一部文字列をドントケアとする」などの安易な方法によって，ホワイトリストに曖昧性を含めてしまうと，かつてのダイナミックプロパティ [33] のような脆弱性を孕みかねないため，十分な検討が必要である．

提案方式においては，ランデブーポイント（レンダード HTML コードの中でソースの情報がシンクに引き渡されるポイント）の HTML 要素を，ホワイトリストとして用いるため，HTML コードにランデブーポイントの情報（<div id=rendID> タグ）を埋め込んだランデブー HTML コードを生成して利用している．しかし，WEB サーバ側で HTML コードが動的生成されるタイプの Web アプリケーションの場合は，ランデブー HTML コードを Web サーバ側に搭載しておくという運用をとることが難しい．WEB サーバ側で動的生成される HTML コードについて

© 2022 Information Processing Society of Japan

は，2.5 節で説明したように，API Gateway による検査が可能であるので，提案方式と API Gateway を相補的に利用するなどのアプローチが必要である．

本論文では，単一の Web ページを対象として提案方式の説明，実装，評価を行ったが，実際には，HTML コードが iframe によって複数に分けられているケースや，CSS から呼び出されるケースも一般的である．現時点の提案方式の実装では，iframe や CSS に対応できていないが，複数の HTML コードに対して提案方式を同様な形で適用することによって，提案方式の実装は可能であると考えている．また，本論文では，HTML+JavaScript による動的 Web ページを対象として提案方式の説明，実装，評価を行ったが，近年は React，Angular，Vue.js の利用も増加してきている．これらの JavaScript フレームワークが提案方式に影響を及ぼす箇所は，HTML コードの記法（React では JSX ベース，Angular と Vue.js はテンプレートベース）と DOM 操作（仮想 DOM や Incremental DOM を用いた差分レンダリング）である．現時点の提案方式の実装では，これらの記法および DOM 操作に対応できていないが，提案方式のコンセプト自体は十分に適応可能であると考えている*6．

以上は，現時点の提案方式が有する制限であり，本論文においてはこれらの改善については今後の課題とする．

7. まとめ

本論文では，マイクロフロントエンド（MFE）型の Web アプリケーションに対して，テストベースホワイトリストのアプローチを採用することによって，セキュリティポリシを利用したセキュリティ機構に代わるフロントエンド側のセキュリティ機構を実現した．本論文 DOM ツリー構造と HTML 要素の情報により構成されるホワイトリストを利用することで，MFE 型 Web アプリケーションの DOM Based XSS 対策を達成した．提案方式の評価においては，簡素な MFE 型 Web アプリケーションを実際に構築し，典型的な DOM Based XSS 攻撃に対して効果的な対策ができることを検証した．今後は，現時点の提案方式が有する制限の解除方法とホワイトリストの実装におけるオーバヘッドの軽減方法の検討，大規模な MFE 型 Web アプリケーションにおける提案方式の有効性評価および改良を行っていく．

*6 具体例として，5.2 節の「ランデブーポイント特定処理」を React に対応させる方法を説明すると次のとおりである．ランデブーポイント特定処理は，① HTML コード中のランデブーポイントの候補（シンクとして機能する予約語 [21]）を特定する（5.2 節手順 1），② ランデブーポイントの候補を示すマーカ（nodeID）を HTML コードに埋め込んでマークド HTML コードを生成する（5.2 節手順 1），③ マークド HTML コードをレンダリングする（5.2 節手順 2），④ DOM 操作を通じてランデブーポイントを特定する（5.2 節手順 3），⑤ ランデブーポイントを示すマーカ（rendID）をマークド HTML コードに埋め込んで（同時に nodoID を削除する）ランデブー HTML コードを生成する（5.2 節手順 4～6），という流れになっている．これに対し，① においては，HTML コードの中からシンクとして機能する予約語 [21] に紐づいている dangerouslySetInnerHTML() などの関数を発見するようにしてやれば，ランデブーポイントの候補の特定が可能である．② においては，createElement() 関数を使用することによって，HTML コードにマーカ（nodeID）埋め込むことが可能である．③ については変更不要である．④ においては，メモリ上に展開される DOM 要素の取得を行う機構（仮想 DOM への対応），ならびに，無変更部分の DOM 要素の保存と復帰を行う機構（Incremental DOM への対応）を実装してやれば，提案方式と同じ手順で DOM 操作を通じてランデブーポイントの特定が可能である．⑤ においては，② と同様，createElement() 関数を使用することによって，HTML コードにマーカ（rendID）埋め込むこと可能である．また，③ で埋め込まれた createElement() 関数を削除することにより，HTML コードからマーカ（nodeID）を削除することも可能である．なお，5.4 節の「攻撃判定処理」に関しては，Web ブラウザのデバッグ機能を利用して Scripting フェーズに介入することになる．このため，現時点の提案方式の実装では Acorn.js を用いて行っている「HTML 要素の抽出」操作についても，JSX ベースの記法/テンプレートベースの記法に対応させた形でこれを実行することになる．

参考文献

[1] Linthicum, D.S.: Practical Use of Microservices in Moving Workloads to the Cloud, *IEEE Cloud Computing*, Vol.3, No.5, pp.6–9 (2016).

[2] Butzin, B., Golatowski, F. and Timmermann, D.: Microservices approach for the internet of things, *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation*, pp.1–6 (2016).

[3] Bui, T., Rao, S., Antikainen, M. and Aura, T.: XSS Vulnerabilities in Cloud-Application Add-Ons, *Proc. 15th ACM Asia Conference on Computer and Communications Security*, pp.610–621 (2020).

[4] Micro Frontends | martinFowler.com, available from ⟨https://martinfowler.com/articles/micro-frontends.html⟩ (accessed 2022-06-28).

[5] Yarygina, T. and Bagge, A.H.: Overcoming Security Challenges in Microservices Architectures, *2018 IEEE Symposium on Service-Oriented System Engineering*, pp.11–20 (2018).

[6] Yang, D., Gao, Y., He, W. and Li, K.: Design and Achievement of Security Mechanism of API Gateway Platform Based on Microservice Architecture, *Journal of Physics Conference Series*, Vol.1738, No.1 (2021).

[7] Mateus-Coelho, N., Cruz-Cunha, M. and Ferreira, L.G.: Security in Microservices Architectures, *Procedia Computer Science*, Vol.181, pp.1225–1236 (2021).

[8] Kothawade, P. and Bhowmick, P.S.: Cloud Security: Penetration Testing of Application in Micro-service architecture and Vulnerability Assessment, Dissertation (2019).

[9] 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 の組合せによる効果的な XSS 対策の実現，情報処理学会論文誌，Vol.61, No.9, pp.1374–1387 (2020).

[10] Microservices, available from ⟨https://martinfowler.com/articles/microservices.html⟩ (accessed 2022-06-28).

[11] Trusted Types | w3c, available from ⟨https://w3c.github.io/webappsec-trusted-types/dist/spec/⟩ (accessed 2022-06-28).

[12] イベントの作成と起動—イベントリファレンス | MDN Web Docs，入手先 ⟨https://developer.mozilla.org/ja/docs/Web/Events/Creating_and_triggering_events.⟩（参照 2022-06-28）.

[13] Li, X., Chen, Y., Lin, Z., Wang, X. and Chen, J.H.:

© 2022 Information Processing Society of Japan

Automatic Policy, Generation for Inter-Service Access Control of Microservices, *30th USENIX Security Symposium*, pp.3971–3988 (2021).

[14] 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 2022-07-02）.

[15] 「DOMBasedXSS」に関するレポート | IPA テクニカルウォッチ，入手先 ⟨https://www.ipa.go.jp/files/000024729.pdf⟩（参照 2022-07-02）.

[16] Stock, B., Lekies, S., Mueller, T., Spiegel, P. and Johns, M.: Precise Client-side Protection against DOM-based Cross-Site Scripting, *23rd USENIX Security Symposium* (2014).

[17] Wang, X., Wu, R., Ma, J., Long, G. and Han, J.: Research on Vulnerability Detection Technology for WEB Mail System, *Procedia Computer Science*, Vol.131 (2018).

[18] CSP: trusted-types | MDN Web Docs), available from ⟨https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/Content-Security-Policy/trusted-types⟩ (accessed 2022-07-02).

[19] Amazon API Gateway（規模に応じた API の作成，維持，保護）| AWS，入手先 ⟨https://aws.amazon.com/jp/api-gateway/⟩（参照 2022-06-28）.

[20] AWS WAF（ウェブアプリケーションファイアウォール）| AWS，入手先 ⟨https://aws.amazon.com/jp/waf/⟩（参照 2022-06-28）.

[21] AWS Lambda（イベント発生時にコードを実行）| AWS，入手先 ⟨https://aws.amazon.com/jp/lambda/⟩（参照 2022-06-28）.

[22] Wang, R., Xu, G., Zeng, X., Li, X. and Feng, Z.: TT-XSS: A novel taint tracking based dynamic detection framework for DOM Cross-Site Scripting, *Journal of Parallel and Distributed Computing*, Vol.118, Part 1, pp.100–106 (2018).

[23] Lekies, S., Stock, B. and Johns, M.: 25 million flows later: large-scale detection of DOM-based XSS, *CCS 2013*, pp.1193–1204 (2013).

[24] Introducing DOM Invader: DOM XSS just got a whole lot easier to find | PortSwigger Blog, available from ⟨https://portswigger.net/blog/introducing-dom-invader⟩ (accessed 2022-06-28).

[25] XPath | MDN Web Docs, available from ⟨https://developer.mozilla.org/ja/docs/Web/XPath⟩ (accessed 2022-06-28).

[26] Chrome.debugger, available from ⟨https://developer.chrome.com/docs/extensions/reference/debugger/⟩ (accessed 2022-06-28).

[27] Acorn.js, available from ⟨https://github.com/acornjs⟩ (accessed 2022-06-28).

[28] Selenium, available from ⟨https://www.selenium.dev/⟩ (accessed 2022-07-02).

[29] Micro Frontends – extending the microservice idea to frontend development | Michael Geers, available from ⟨https://micro-frontends.org/⟩ (accessed 2022-07-02).

[30] Newman, S.，佐藤直生，木下哲也：*Microservice Architecture*, O'REILLY (2016).

[31] Open Source Foundation for Application Security | OWASP Foundation, available from ⟨https://owasp.org/⟩ (accessed 2022-07-02).

[32] Chrome DevTools | Chrome Developers, available from ⟨https://developer.chrome.com/docs/devtools/⟩ (accessed 2022-07-02).

[33] Ending Expressions | Microsoft Docs, available from ⟨https://docs.microsoft.com/en-us/archive/blogs/ie/ending-expressions⟩ (accessed 2022-07-02).

[34] Fetch directive (フェッチディレクティブ) – MDN Web Docs 用語集：ウェブ関連用語の定義 | MDN，入手先 ⟨https://developer.mozilla.org/ja/docs/Glossary/Fetch_directive⟩（参照 2022-07-02）.

[35] CSP: img-src – HTTP | MDN, available from ⟨https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/Content-Security-Policy/img-src⟩ (accessed 2022-07-02).

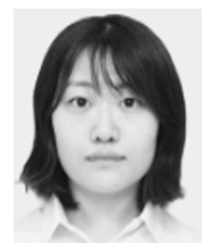
[36] Vagrant by HashiCorp, available from ⟨https://www.vagrantup.com/⟩ (accessed 2022-07-02).

井坂 佑介

2022 年静岡大学情報学部情報科学科卒業．同年三菱電機株式会社入社．在学中，Web アプリケーションに関する研究に従事．

天笠 智哉

2022 年静岡大学情報学部情報科学科卒業．2022 年同大学院総合科学技術研究科情報学専攻修士課程在学中．現在は情報セキュリティに関する研究に従事．



奥村 紗名

2022 年静岡大学情報学部情報科学科卒業．2022 年同大学院総合科学技術研究科情報学専攻修士課程在学中．現在は情報セキュリティに関する研究に従事．

佐々木 葵

2022 年静岡大学情報学部情報科学科卒業．同年株式会社ラック入社．在学中，デジタル・フォレンジックに関する研究に従事．

© 2022 Information Processing Society of Japan

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，2020年同大学准教授．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．電子情報通信学会会員．

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，2019～2020年情報環境領域委員長，2020年調査研究運営委員長．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．2016～2020年日本セキュリティマネジメント学会編集部会長，2021年より副会長．情報処理学会フェロー．

© 2022 Information Processing Society of Japan