Vol.2022-SPT-46 No.22
2022/3/8

DOM Based XSS を防ぐための開発支援フレームワークの提案と ESLint を用いた実装

日浦　秀侑[1]　金岡　晃[1]

概要：ソフトウェア開発者が DOM Based XSS 脆弱性をソースコードに含めないようにするための開発支援フレームワークを提案する。その実証として、JavaScript の主要な静的解析ツールである ESLint と、開発環境として広く用いられている Visual Studio Code を用いて、ヒューリスティックな文字列変換を行い、 DOM Based XSS 脆弱性をソースコード内に含めないようにする仕組みをシンプルな形で実装した。また、その仕組みの妥当性を評価し今後の応用可能性について議論する。

Shu Hiura[1]　Akira Kanaoka[1]

1. はじめに

近年第 5 世代移動通信システム (5G) や人工知能 (AI) など、ソフトウェアやアプリケーション、またその周辺環境の技術が高度化している。これに伴い、それらのセキュリティへの重要性が高まっている。しかし、開発現場では技術の高度化への対応によりセキュリティの対応が後手に回っている。そこで安全にソフトウェアを開発できるように開発者向けのユーザブルセキュリティとユーザブルプライバシー (以後 USP) の研究が進んでいる。

開発者向けの USP 研究として開発者の行動特性を調査する研究と、その行動特性に基づいたユーザビリティに関する研究がある。行動特性の調査研究で多くの開発者が基本的なセキュリティの実装を適切に施していないことが明らかになった [1]。この行動特性に基づいた USP 研究では開発者がコーディングをする際に支援する技術の研究が行われている [2], [3]。しかしこれらの研究では基礎的なセキュリティへの対応を支援する限定的な研究にとどまっている。そこで本研究は、より専門的で経験的な知識をベースとする支援技術を検討する。

支援技術として本研究は安全なソフトウェア開発のための開発支援フレームワークを提案する。開発支援フレームワークを提案するにあたり、ソフトウェア工学の研究知見をセキュリティへ応用することを検討する。ソフトウェア工学の分野では、バグを自動で検出し自動で修正する技術 (以後 自動修復技術) の研究が進んでいる。バグと脆弱性はまったく同じものではないが、双方がプログラム上の欠陥であり、共通部分があることからセキュリティへの応用を検討する。

具体的なアプローチとしてバグの自動修復技術のアプローチをセキュリティへ応用する。また、セキュリティへの応用として本研究では、JavaScript の脆弱性である DOM Based XSS 脆弱性をソースコード内に含めないようにするための支援を実装した。実装するにあたり、ESLint を Visual Studio Code (以後 VSCode) に設定し、ヒューリスティックな文字列変換を行う。また、その仕組みの妥当性を評価し今後の展望について議論する。

本論文の構成は以下の通りである。第 2 章で既存の研究について述べ、第 3 章でソフトウェア工学の研究知見を基にした、セキュアな開発支援フレームワークを提案する。第 4 章で提案した手法を簡易的に実装し、評価を行う。第 5 章で評価結果から考察と今後の展望について述べ、最後に第 6 章で本論文のまとめを行う。

2. 関連研究

2.1 バグの自動修復技術

ソフトウェア工学の分野でバグの自動修復技術の研究が進んでいる。バグと脆弱性はまったく同じものではないが、双方がプログラム上の欠陥であり、共通部分があることからセキュリティへの応用が期待できる。そこでバグの自動修復技術についての関連研究を紹介する。

1　東邦大学
Toho Uniersity

ⓒ 2022 Information Processing Society of Japan

2.1.1 バグレポートの利用

バグの自動修復技術に関連する研究は、主にバグレポートを用いた研究が多い。Liu らは 2013 年に、バグレポートからバグの修正パッチを自動的に生成する手法として R2Fix を提案した [4]。また、Kim らの論文 [5] や Zhou らの論文 [6] では、バグレポートの内容を用いて修正するべきファイルを予測する手法が提案された。しかしこれらの手法は脆弱性への応用は難しい。脆弱性の含まれているソースコードはバグの含まれているソースコードと違い正常に動作するため、レポートが発生しない。

2.1.2 自動修復技術の手法

Gazzola らはソフトウェアの自動修復技術に関連する 108 本の論文を調査した [7]。Gazzola らは調査した論文から、プログラムの不具合を自動的に処理するアプローチを「ソフトウェアヒーリング (以後 ヒーリング) 」と「ソフトウェアリペアリング (以後 リペアリング) 」に分類した。ヒーリングは稼働中にソフトウェアの障害を検出し、正常に動作を回復するための必要な調整を行うことである。ソースコードレベルではなく、デプロイメントされたアプリ上で実行している際に適用される。リペアリングはソフトウェアの障害を検知し、修正が可能な箇所の特定を行い、障害を修正するために必要な調整を行うことである。ソースコードレベルで行われ、テスト時や設計時など開発中に行われる。

また、リペアリングのプロセスに含まれる修正方法に対して、Gazzola らはプログラムの修正を自動的に生成するアルゴリズムのアプローチを、”generate-and-validate” と”semantics-driven”の 2 つに分類した。generate-and-validate アプローチは、障害を解決しうる解の候補を定義し、探索することで修正を提供するアプローチである。解の空間には実際に問題を解くことのできる要素と解くことのできない要素が混在するので、障害を解決することができるとは限らない。semantics-driven アプローチは、障害の発生箇所を形式的にエンコードし、障害を解決することが保証された解を見つけるアプローチである。

generate-and-validate アプローチのプロセスを図 1 で示す。プロセスは、障害の解候補を生成する generate アクティビティと、生成された解候補の正しさを検証する validate アクティビティの 2 つのアクティビティから成る。generate アクティビティは、複数のオペレータを利用して障害の発生箇所を修正し、新たなプログラムを生成して解候補に追加するアクティビティである。そのオペレータには様々なものがあるが、Gazzola らは 3 種類のオペレータについて説明していた。atomic-change オペレータは、プログラムの AST の 1 か所を変更することで修正するオペレータである。pre-defined template オペレータは、事前に定義された障害を解決しうる解の候補に従って修正するオペレータである。example-based template オペレータは

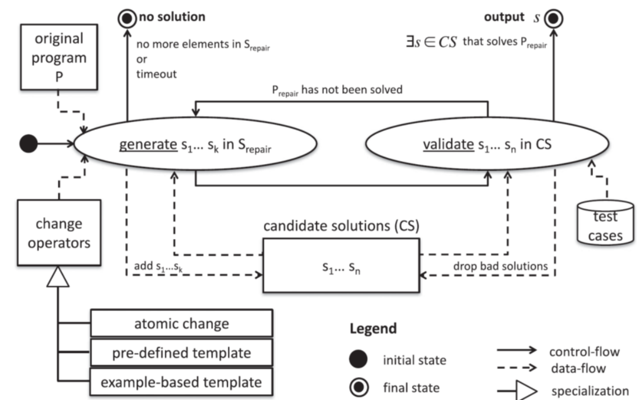


図 1: generate-and-validate ([7] の P.42 の Fig.5 を引用)

pre-defined template オペレータと同じ働きをするが、解の候補が過去のデータから手動、または自動で抽出されるオペレータである。generate アクティビティに対して validate アクティビティは、解候補の正しさを検証するアクティビティである。Gazzola らが調査した論文では、テストケースを実行することで validate アクティビティが提示する解候補の正しさを確立させている。

Gazzola らは、バグが存在するソフトウェアの自動修復技術に関連する論文を調査し、修復のアプローチを分類することにより自動修復技術の分野の知識を整理した [7]。脆弱性はバグとは異なるが、双方に共通部分があることからセキュリティへの応用が期待される。

2.2 ESLint

ESLint は JavaScript のための Linter である。2013 年に Nicholas C. Zakas によって作成された。JavaScript のソースコードをパースしてできたソースコードの AST（抽象構文木: Abstract Syntax Tree）をルールで検証し、エラーや警告を出力する。ESLint は Espree をパーサとして使用している。

ESLint に関する研究はこの数年で徐々に行われてきている。Tómasdóttir らは 2020 年に、ESLint に対する開発者の認識を調査した [8]。ESLint について 15 人の専門家にインタビューを行い ESLint を使う理由を定性分析から調査した結果、エラーを防ぐために ESLint を用いることに対して「強くそう思う」と同意する度合いが 93%であった。また、ESLint のカテゴリ事に重要性を調査した結果、エラーの可能性を示すことに対する度合いとして 92%が非常に重要であると示した。Tómasdóttir らの調査結果より、エラーを防ぐために ESLint を用いることも多く、セキュリティの必要性を示唆している。

Rafnsson らは 2020 年に、実際に ESLint を用いた新たなセキュリティルールを作成し、Linter がどの程度セキュリティに活用できるのか調査した [9]。Rafnsson らは、XSS、SQL インジェクション、ミスコンフィギュレーションの 3

ⓒ 2022 Information Processing Society of Japan

種類の脆弱性を検出し、修正するルールを作成した。また、ルールを作成した経験から ESLint をセキュリティへ応用する際の見解を述べた。その中で、Linter はソフトウェアの脆弱性を検出して排除するために使用することができ、脆弱性の Lint はまだ十分に行われていないことや、ルールを作成したことによる影響は大きいと述べた。それに加えて、ファイルが他のファイルと互いに依存して起こる脆弱性など、Linter だけでは発見することができない脆弱性が多いことから、より多くの脆弱性を発見するために、コードをほかの異なる方法でスキャンすべきであるとも述べている。

Rafnsson らは、ESLint の新しいルールを作成し、セキュリティへの応用可能性の議論を活性化させたが、脆弱性の原因に対する網羅性が少ないないことや、ルールに対する実験を行えておらず、実現可能性の根拠が十分でないことなどの課題があげられる [9]。

2.3 DOM Based XSS

DOM Based XSS とは、攻撃者がスクリプトを含めたリンクを閲覧者に送信し、閲覧者がそのリンクにアクセスしてしまうことにより、サイトが Web ページを動的に操作する正規のスクリプトを含む Web ページを返し、閲覧ページ上で正規のスクリプトが攻撃者の含めたスクリプトを動的に書き込むことでスクリプトが実行してしまうタイプの XSS である。DOM Based XSS はインジェクション攻撃の一種であり、重大なセキュリティリスクのリストである OWASP Top 10 に記載がある。また、DOM Based XSS 脆弱性は正規のスクリプトによる処理に脆弱性が含まれる。

その DOM Based XSS を防ぐための研究も多く行われている。Lekies らは 2013 年の研究で、DOM Based XSS 脆弱性を検出し検証するための自動化手法を提案した [10]。この手法ではブラウザの JavaScript エンジンに直接組み込むことで、安全ではないデータの流れを特定することを実現した。さらに Parameshwara らは 2015 年に、DOM Based XSS 脆弱性を自動パッチするツールである DEXTERJS を提案した [11]。DEXTERJS は脆弱性の含まれている文字列をより安全なコードに書き換える安全なパッチを生成する。DEXTERJS を用いて数百の DOM Based XSS 脆弱性に適度なパフォーマンスでパッチを当てることができ 、解析駆動型パッチが JavaScript アプリケーションのセキュリティ確保のための実用的な代替手段になることを示した。しかしこれらの手法はすでに完成されたコードに対する対策でソースコードに DOM Based XSS 脆弱性を含めないための根本的な解決にならない。この他にも DOM Based XSS を防ぐための研究が多く行われており、その研究をまとめた Liu らの調査論文があるがここでは参考文献にとどめるまでとする [12]。

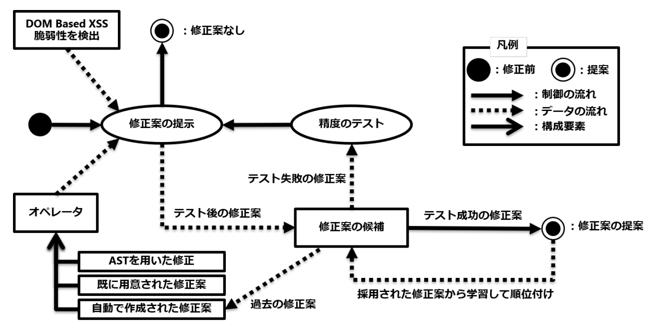


図 2: 提案フレームワーク

3. 開発支援フレームワークの提案

本研究の目的は、ソフトウェア開発者が安全なソフトウェアを開発することである。そこで、コード作成時に脆弱性などの欠陥を含めないように支援する開発支援フレームワークを提案する。

提案するフレームワークは、Gazzola らの論文で自動修復技術のアルゴリズムとして挙げられた generate-and-validate アプローチを脆弱性の修復へ応用する。応用するにあたり、本研究では脆弱性を DOM Based XSS 脆弱性に限定する。OWASP Top 10 が公開された 2003 年から記載があるにもかかわらず、未だ重要なセキュリティリスクであることから対象とした。同様に XSS として含まれる Reflected XSS や Stored XSS は発生する原因はサーバ側にある。それに対して DOM Based XSS は、JavaScript のソースコードに含まれる脆弱性が原因で発生することもあり、コード作成時の対応である支援が有用と考えられる。

本論文で提案する、DOM Based XSS 脆弱性をソースコードに含めないための開発支援フレームワークを図 2 に示す。2.1.2 で紹介した Gazzoola の generate-and-validate アプローチを応用したものである。“generate” にあたる部分が「修正案の提示」に、”validate” にあたる部分が「精度のテスト」に該当する。その他にも ”candidate solutions” は「修正案の候補」に対応するなど、Gazzola らのアプローチと対応している部分が多い。Gazzola らのアプローチと異なる部分として主に、「過去の修正案」と「採用された修正案から学習して順位付け」のデータの流れが上げられる。「過去の修正案」では、オペレータに対して作成された修正案を送る。オペレータはその修正案を学習して新たな修正案を作成し、提示する仕組みである。「採用された修正案から学習して順位付け」では、ユーザが採用した修正案のデータを記録する。そのデータをもとに、次回修正案を提示する際に適切な修正案を推奨する仕組みである。

ⓒ 2022 Information Processing Society of Japan

4. フレームワークのプロトタイプ実装とその評価

本章では、提案したフレームワークの実現可能性を検討するため、実装のプロトタイプを ESLint を用いて作成し、その評価を行った。

4.1 ESLint によるプロトタイプ実装

本研究では、開発支援フレームワークのプロトタイプを ESLint を用いて実装した。ESLint はソースコードを AST にパースして静的解析を行う JavaScript 用の Linter である。ESLint に組み込まれているルールにより検証を行い、ソースコード上の問題点を指摘し、修正案を提示することができる。ESLint は既存のルールだけではなく、独自で作成したルールをプラグインとして ESLint に組み込み、機能を拡張することができる。作成したプラグインはローカルな環境で用いることに加え、npm に登録して公開することも可能である。ESLint は汎用性が高く、ソースコードを AST にパースしてことから、オペレータとしての実装が期待できる。

ESLint を用いたプロトタイプの構成はフレームワークの構成要素である「DOM Based XSS 脆弱性を検出」、「オペレータ」、「修正案の提示」を実装している。「DOM Based XSS 脆弱性を検出」はソースコードが ESLint のルール内で問題点を検出した際に波線を出力することで、「オペレータ」は ESLint が問題点を修正するための手段を選定することで、「修正案の提示」は ESLint が修正案をダイアログで提示することでそれぞれ実現した。また、ESLint は修正をした後、検証をソースコード全体で再度行うことから「精度のテスト」を実現しているとも言える。

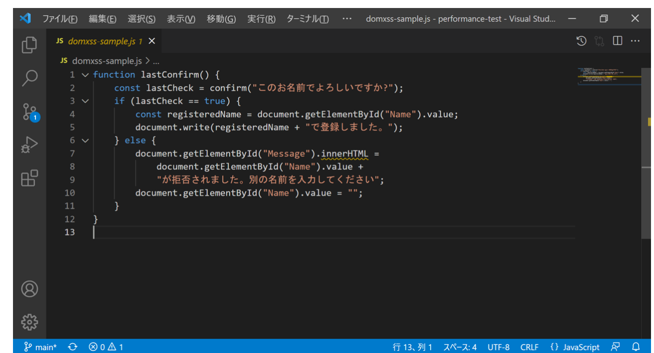
4.1.1 対応する DOM Based XSS 脆弱性

まず、ESLint を用いて防ぐ DOM Based XSS 脆弱性について述べる。本研究では、DOM Based XSS 脆弱性の基本的な原因である innerHTML プロパティと document.write メソッドがソースコードから検出された際に、DOM Based XSS 脆弱性が含まれないよう修正を加える ESLint のルールを実装した。innerHTML プロパティと、document.write メソッドはそれぞれ HTML タグなどを有効にしてしまうため、DOM Based XSS 脆弱性が含まれる。

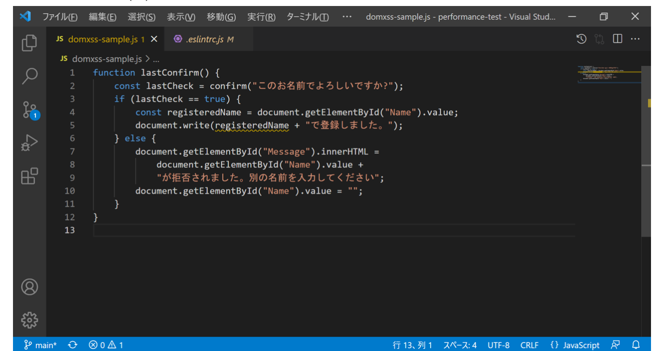
DOM Based XSS を防ぐための対策として、innerHTML プロパティでは textContent に置き換えることで HTML タグなどを文字列として出力することが挙げられる。document.write メソッドでは代替となる手段が使用目的に応じて異なる。そのため、出力テキストをエスケープすることにより HTML タグなどを文字列として出力する。これにより、一貫して DOM Based XSS を防ぐことができる。

4.1.2 ルールの内容

本研究ではそれぞれ、innerHTML プロパティを textContent プロパティに置き換え、document.write メソッドで出力するテキストをエスケープすることで DOM Based XSS を対策するための、ESLint のルールを作成した。



(a) innerHTML プロパティによる脆弱性



(b) document.write メソッドによる脆弱性

図 3: 脆弱性の検出、波線の出力

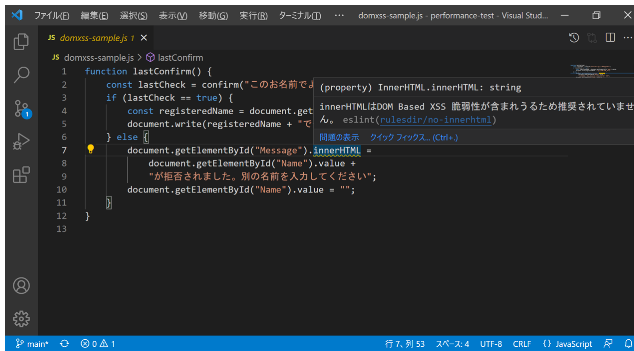
ESLint は、ソースコードをパースしてできた AST を探索し、各ノードを分析する。ルールで指定した、特定のノードに到達した際にそのノードを問題箇所として波線を出力する。提案したルールでは、AST の探索で *Identifire* type を持つノードに到達した際、その name が *innerHTML* である場合を問題とする。そして、修正案として、innerHTML プロパティを textContent プロパティに置き換えることを開発者に提案する。同様に、document.write メソッド内の変数に対して、エスケープをする関数を呼び出し、戻り値にエスケープされたテキストを返す。また、エスケープする関数がソースコード内に存在しない場合、新たに関数を宣言する。

4.1.3 ルールの実行

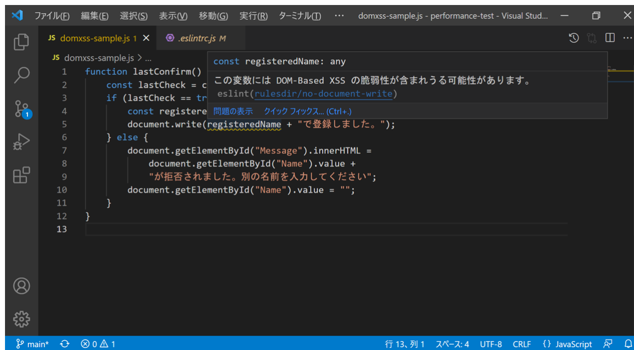
ESLint が修正を加える流れを図 3、4、5、6 に示す。図 5 で提示されている複数の修正案に関して、1 番上に提示されている案を選択することで文字列変換を行い修正することができる。2 番目を選択することでこの波線に対して修正を行わないで無視することができ、3 番目を選択することでこのファイルに対して修正を行わないことができる。最後の 4 番目に提示されている案を選択するとこで修正案の詳細を表示する Web ページに遷移することができる。

また、図 7 が ESLint を実行した後の Web ページの動き

ⓒ 2022 Information Processing Society of Japan

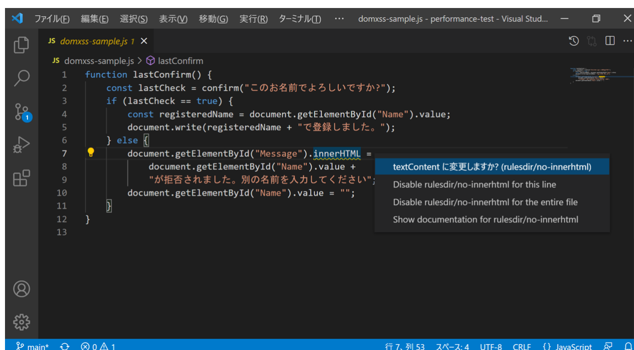


(a) innerHTML プロパティによる脆弱性

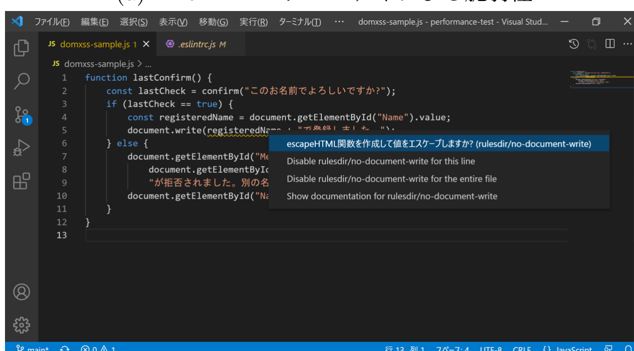


(b) document.write メソッドによる脆弱性

図 4: 波線部にカーソルを合わせる、エラーメッセージの出力

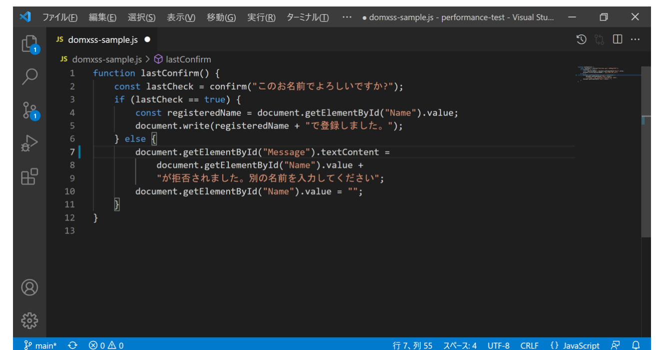


(a) innerHTML プロパティによる脆弱性

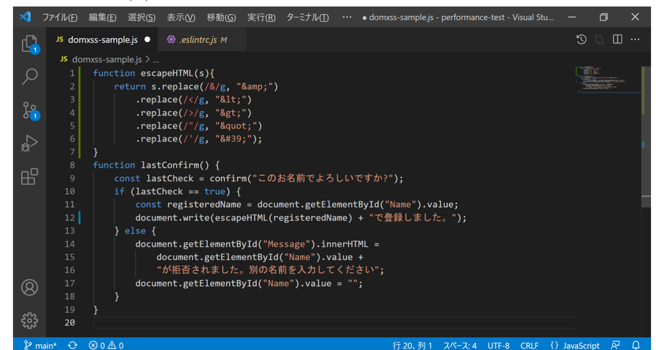


(b) document.write メソッドによる脆弱性

図 5: クイックフィックスを選択、修正案の提示

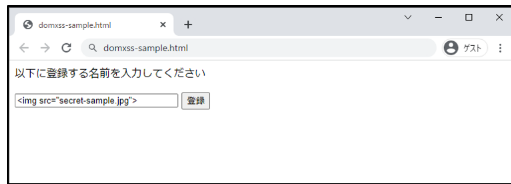


(a) innerHTML プロパティによる脆弱性

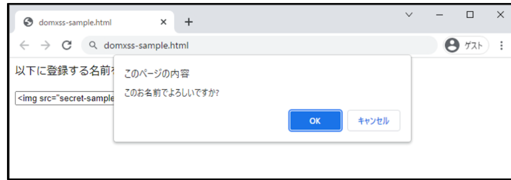


(b) document.write メソッドによる脆弱性

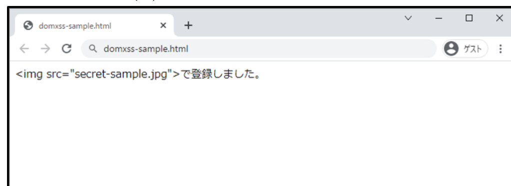
図 6: 脆弱性が含まれないよう修正



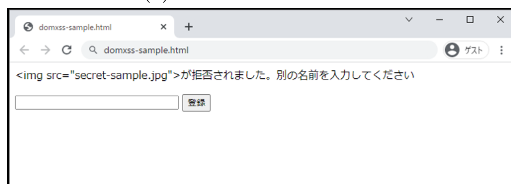
(a) スクリプトの入力



(b) 登録ボタン押下、確認



(c) はいを選択した場合



(d) いいえを選択した場合

図 7: ESLint で DOM Based XSS 脆弱性を防いだ後の Web ページの画面遷移

を示す。スクリプトを入力しているのにもかかわらず、テキストとして出力されていることがわかる。

4.2 評価

実現可能性を測る評価として、ESLint 用いた際にユーザへ与える影響を調査する。 評価手法は ESLint のルールごとの解析時間を分析することによりコーディングの処理速度にどの程度影響を与えるかを 1 つの指標として評価する。個々のルールが解析に掛かる時間は ESLint の公式ホームページに記されている方法を用いる [13]。環境変数に *TIMING* を設定することで、個々のルールの実行時間とルール処理時間の合計に対する実行時間の割合がルールごとに表示される。また、*TIMING* の値は 1 を代入した場合、最も長く実行されている 10 個のルールを表示し、10 より多い数のルールを表示させたい場合は表示させたい数の値を代入する。

今回はこのコマンドを 100 回行い、*no-innerhtml* と *no-document-write* の 2 つのルールにかかった実行時間の平均値を算出し、ユーザへ与える影響を考察する。

4.2.1 実験環境の構築

実験で利用するコンピュータの環境を表 1 に示す。

表 1: 評価に用いた環境

項目	詳細
OS	Windows 10 Home 21H2
メモリ	16GB
CPU	11th Gen Intel (R) Core (TM) i5-1135G7 @ 2.40GHz 2.42 GHz
CUI	PowerShell 7.2.1

ESLint のプラグインは npm で公開されているパッケージをインストールして使用することが一般的だが、本研究ではローカルなテスト用プラグインを VSCode に設定するために *eslint-plugin-rulesdir* を用いて実装した [14]。

4.2.2 評価結果

評価の結果、ESLint を 100 回実行したときのルールごとの実行時間の平均は *no-innerHTML* が 0.109ms、*no-document-write* が 0.906ms であり、実行比率の平均は *no-innerHTML* が 2.01%、*no-document-write* が 16.63%であった。また、各実行時間の最大値は *no-innerHTML* が 0.235ms、*no-document-write* が 2.025ms であり、最小値は *no-innerHTML* が 0.091ms、*no-document-write* が 0.788ms であった。

5. 考察

本章では、第 5 章の評価結果から、ユーザへの影響、実現可能性に対する考察と今後の展望について述べる。

5.1 ユーザへの影響に対する考察

評価結果から、行数の短いシンプルなソースコードに対して ESLint を使用する際のユーザに対する影響は低いと確認できた。実装したヒューリスティックな文字列変換は規模も小さいため、すぐに修正が行われた。そのため、ユーザへのパフォーマンスに対する影響は少ないとわかる。また、コーディング時に自動で問題を検出することからユーザの負担も少ないと考えられる。

5.2 実現可能性に対する考察

ルールの作成から、ESLint は atomic-change オペレータとして十分に機能し、generate-and-validate アプローチをセキュリティへの応用が可能であることが示唆された。ESLint を VSCode を設定することに対しては、拡張機能のインストール数も 18,000,000 件を超えていることから、妥当性としては十分であると考えられる [15]。

5.3 今後の展望

本研究は、generate-and-validate アプローチをセキュリティへ応用させるための先駆けとして、ESLint を atomic-change オペレータとして用いる実証を行った。実現可能性を検討する議論を進めたが、実現するための十分なフレー

ⓒ 2022 Information Processing Society of Japan

ムワークが実装できていない。実現的なフレームワークを実装するための今後の展望をここで述べる。

まずルールの仕組みに対して本研究では、全ての innerHTML プロパティと document.write メソッド内のエスケープされていない全ての変数に対して波線を出力する単純な設計であるため誤検知が多いことが予想される。今後は分析を深め、DOM base XSS 発生防止に必要な部分と必要ではない部分などを適切に判別し、誤検知を減らすことが課題として挙げられる。しかし、仕組みが複雑になることで、検出するまでの時間が伸びる恐れがあることに加え、行数の多い複雑なソースコードに対して ESLint を実行する際に処理が多くなり、影響が増える可能性がある。仕組みを複雑化するに伴うユーザへの影響も並行して評価を進める。

また、本研究はユーザに対する影響を評価する指標として ESLint の実行時間のみを評価したが、他の指標も用いてユーザに対する有用性も評価することも今後の展望として挙げられる。たとえば、アンケートを用いて分析する量的なアプローチが挙げられる。あるいは半構造化インタビューなどを通した質的なアプローチも取りうる。本研究で用いた ESLint はコーディング時にリアルタイムで検出し波線を出力するが、果たしてユーザにとってどれほど有用であるかは、いずれのアプローチでも検証可能だろう。ユーザ実験としては、実験参加者に対して期間をあけて 2 週間後に同じタスクを課し、実験参加者が脆弱性に対する理解が深まったかや、コードの修正が適切に行われている度合いを検証することも考えられる。

さらに、今後仕組みを複雑にするにあたり、ESLint だけではなくほかのシステムも組み込み複合的に解析することで、今後ライブラリなどを含めた複数ファイルの分析を行えないか、また複合的に分析した結果をどのようにしてソースコードやシステム全体の修正を支援するかの具体性も課題に挙げられる。加えて、未知の脆弱性への対応も検討される。

本研究で用いた脆弱性を含む JavaScript ソースコードは著者が作成したものであるため、正確性の議論は行わなかった。今後は、実際に DOM based XSS が含まれる既存のソースコードを利用してその検知能力や修正能力を測ることが重要であると考える。しかし DOM Based XSS 脆弱性が含まれていると定められているオープンなデータセットは著者らの知る限り存在しない。そのため、StackOverflow や GitHub といったオープンな環境に掲載されているソースコード群に対する DOM based XSS 脆弱性を含むソースコードの調査や、それをもとにしたデータセットの作成なども課題として挙げられる。

6. まとめ

近年の情報技術の高度化に伴い、セキュリティの重要性が高まっているが、開発現場では技術の高度化への対応によりセキュリティの対応が後手に回っている。この問題を解決する手段として本論文では開発者が安全にソフトウェアを開発するための開発支援フレームワークを提案した。また、フレームワークを実装するためのアプローチの 1 つとして、ソフトウェア工学の研究分野においてバグの自動修復技術のアプローチとして用いられる generate-and-validate アプローチをセキュリティへ応用することを検討した。実装の先駆けとして、DOM Based XSS 脆弱性をソースコード内に含めないことを目的とするオペレータを ESLint を用いてシンプルな形で作成した。そして、作成した ESLint の実行時間を計測することでユーザに対する影響を評価し、実装した仕組みの実現可能性を検討した。結果として、generate-and-validate アプローチをセキュリティへ応用することが可能であることを明らかにするとともに、今後の展望をまとめた。

謝辞　本研究の一部は JSPS 科研費 JP19K11972 の助成を受けたものです。

参考文献

[1] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS ’13). ACM, New York, NY, USA, 73-84. DOI: http://dx.doi.org/10.1145/2508859.2516693

[2] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. 2016. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS ’16). ACM, New York, NY, USA, 711-722. DOI: https://doi.org/10.1145/2897845.2897896

[3] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. 2017. A Stitch in Time: Supporting Android Developers in WritingSecure Code. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS ’17). ACM, New York, NY, USA, 1065-1077. DOI: https://doi.org/10.1145/3133956.3133977

[4] Chen Liu, Jinqiu Yang, Lin Tan, Munawar Hafiz, ”R2Fix: Automatically Generating Bug Fixes from Bug Reports,” in 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 18-22 March, 2013 doi: 10.1109/ICST.2013.24.

[5] Dongsun Kim, Yida Tao, Sunghun Kim, Andreas Zeller, ”Where Should We Fix This Bug? A Two-Phase Recommendation Model,” in IEEE Transactions on Software Engineering, vol. 39, no. 11, pp. 1597-1610, 21 May, 2013 doi: 10.1109/TSE.2013.24.

[6] Jian Zhou, Hongyu Zhang, David Lo, ”Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports,” in 2012 34th International Conference on Software Engineering (ICSE), 2-9 June, 2012 doi: 10.1109/ICSE.2012.622710.

[7] L. Gazzola, D. Micucci and L. Mariani, ”Automatic Software Repair: A Survey,” in IEEE Transactions on Software Engineering, vol. 45, no. 1, pp. 34-67, 1 Jan, 2019.

ⓒ 2022 Information Processing Society of Japan

[8] K. F. Tómasdóttir, M. Aniche and A. Van Deursen, "The Adoption of JavaScript Linters in Practice: A Case Study on ESLint," in IEEE Transactions on Software Engineering, vol. 46, no. 8, pp. 863-891, 1 Aug. 2020, doi: 10.1109/TSE.2018.2871058.
[9] Rafnsson W., Giustolisi R., Kragerup M., Høyrup M., "Fixing Vulnerabilities Automatically with Linters," In: Kutyłowski M., Zhang J., Chen C. (eds) Network and System Security. NSS 2020. Lecture Notes in Computer Science, vol 12570. Springer, Cham. https://doi.org/10.1007/978-3-030-65745-1_13.
[10] Sebastian Lekies, Ben Stock, Martin Johns Authors, ",25 million flows later: large-scale detection of DOM-based XSS" in CCS '13: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, November 2013, Pages 1193-1204, doi: https://doi.org/10.1145/2508859.2516703 .
[11] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, Prateek Saxena, "Auto-patching DOM-based XSS at scale," in ESEC/FSE 2015: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering,August 2015, Pages 272-283, doi: https://doi.org/10.1145/2786805.2786821.
[12] Miao Liu, Boyu Zhang, Wenbin Chen, Xunlai Zhang, "A Survey of Exploitation and Detection Methods of XSS Vulnerabilities," in: IEEE Access (Volume: 7), pp. 182004-182016, 17 December. 2019, DOI: 10.1109/ACCESS.2019.2960449.
[13] ESLint. "Per-rule Performance" ESLint. https://eslint.org/docs/developer-guide/working-with-rules#per-rule-performance, (参照 2022-1-20).
[14] npm. "eslint-plugin-rulesdir" npm. https://www.npmjs.com/package/eslint-plugin-rulesdir, (参照 2022-1-20).
[15] 2022 Microsoft. "ESLint" Visual Studio Marketplace. https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint, (参照 2022-1-25).

ⓒ 2022 Information Processing Society of Japan