

API操作ログ取得による 難読化 JavaScript コード解析支援システム

上川 先之¹ 山内 利宏^{1,a)}

概要：Web を介するサイバー攻撃に使用される JavaScript コードは、目的の隠蔽や検知回避のために、難読化などの解析妨害が施されていることがある。迅速な対応が求められるサイバー攻撃において、このようなコードの迅速な解析が課題となる。そこで、我々は、難読化 JavaScript コード解析支援システムを提案する。提案システムは、Web ブラウザの API 操作を捕捉し、難読化 JavaScript コードの挙動を把握できるように API 操作ログを解析者に提示する。API 操作の捕捉に Proxy オブジェクトを利用することで、既存手法で捕捉できなかった API 操作を捕捉する。また、再代入できないグローバル変数として提供される API に対しても、変数の参照を置き換える手法により、API 操作を捕捉する。本稿では、提案システムのコンセプト、実現方式、および評価について述べる。

キーワード：難読化 JavaScript, 解析支援システム, 動的解析

1. はじめに

Web を介するサイバー攻撃では、スクリプト言語である JavaScript が使用されることが多い [1]。JavaScript は、通常、Web 開発者が Web ブラウザを制御するために利用される。しかし、Web を介するサイバー攻撃においては、ユーザを攻撃ページへ誘導するために、または Web ブラウザや Web アプリケーションの脆弱性を悪用するために JavaScript が使用される。JavaScript を使用する攻撃として、Web アプリケーションに対するクロスサイトスクリプティング (XSS) 攻撃や、Web ブラウザに対するドライブバイダウンロード (DBD) 攻撃がある [2]。例えば、DBD 攻撃では、Web ブラウザの脆弱性を悪用してマルウェアを秘密裏にダウンロードさせるために、多くの場合 JavaScript が使用される [3]。また、XSS 攻撃や DBD 攻撃では、本来の攻撃を行う悪質な Web サイトで JavaScript が使用されるだけでなく、一般の Web サイトが改ざんされて、不正な JavaScript コードを埋め込まれる場合がある [4]。特に、DBD 攻撃では、悪質な Web サイトへ誘導する JavaScript コードが、一般の Web サイトに埋め込まれる [5]。

サイバー攻撃の対策を行うには、攻撃コードを解析し、どのような攻撃なのかを明らかにすることが重要である。しかし、Web を介するサイバー攻撃に使用される JavaScript

コードは、難読化などの解析妨害が施されていることがある [5]。特に、Angler や Rig などのエクスプロイトキットによる攻撃では、解析妨害が施されている場合が多い [6]。JavaScript コードに解析妨害が施されている場合、人間による手動での解析が難しくなり、解析にかかる時間が長くなる。このことは、迅速な対応が求められるサイバー攻撃において問題となる。このため、解析妨害が施された JavaScript コードの解析手法の研究が重要である。

既存の JavaScript コードの解析手法として、特定の API をフックして情報を得る手法と、API のエミュレーションにより Web ブラウザの環境を擬似的に再現する手法がある。しかし、これらの手法には課題がある。そこで、難読化 JavaScript コード解析支援システムを提案し、既存の解析手法における課題へ対処する。本稿では、難読化 JavaScript コード解析支援システムのコンセプト、実現方式、および評価について述べる。

本研究の貢献は、以下に示す通りである。

- (1) 難読化 JavaScript コードに対する Proxy オブジェクトを利用した新たな解析手法を提案する。これにより、既存の解析手法における課題へ対処する。著者らの調べた限りでは、Proxy オブジェクトを JavaScript コードの解析に用いた手法は、まだ提案されていない。
- (2) JavaScript における再代入のできないグローバル変数に対して、変数の参照を置き換える手法を提案する。これにより、一部の再代入のできない API に対して、

¹ 岡山大学 大学院自然科学研究科

^{a)} yamauchi@cs.okayama-u.ac.jp

Proxy オブジェクトを適用できる。

- (3) 提案する解析手法をシステムとして実現し、さらなる解析に有用な情報を解析者に提示することで、解析者による解析を支援する。本システムは、JavaScript コードを事前解析することなく、動的解析できる。

2. JavaScript コードにおける解析妨害技術と解析手法

2.1 解析妨害技術

2.1.1 難読化

難読化とは、あるプログラムを理解が困難な等価なプログラムに変換することである [7]。一般に、難読化は、プログラムを不正な解析から保護するために行われる [7]。プログラムの解析を困難にすることで、プログラムの改ざんやプログラムコードの剽窃を防ぐことができる。

一方、サイバー攻撃に用いられる攻撃プログラムは、攻撃者によって難読化される場合がある。攻撃者は、難読化によって攻撃プログラムの動作を解析されにくくし [5][8][9]、その攻撃プログラムの目的を隠蔽する [10][11]。また、ウイルス対策ソフトウェアやファイアウォールによる検知を回避するためにも難読化が行われる [5][11][12]。難読化により等価なプログラムが生成されることから、多様な亜種の作成を容易にすることも目的として挙げられる [8]。

Web を介するサイバー攻撃においても、使用される JavaScript コードが難読化されていることがある [5][10]。DBD 攻撃の例では、リダイレクト先である悪質な Web サイトの URL の隠蔽や、リダイレクトすること自体の隠蔽のために、JavaScript コードが難読化される。

難読化は、大きく以下の 3 種類に分けられる [13]。

(1) レイアウト難読化

識別子名、コメント、およびインデントなどのプログラムの実行に不要な情報を置き換える変換である。

(2) データ難読化

データ構造やデータ表現を変える変換である。例えば、文字列のエスケープや、数値をそれと同等な演算式に置き換える変換が挙げられる。

(3) 制御フロー難読化

制御フローを変える変換である。例えば、無駄なループ文の挿入や、関数のインライン展開が挙げられる。

JavaScript コードにおいては、特に、コードを圧縮するためにレイアウト難読化が施されることが多い。また、JavaScript には、文字列を JavaScript コードとして実行することのできる `eval` 関数があり、容易にコード全体を難読化できることから、`eval` 関数に渡される文字列にデータ難読化が施されることも多い。

2.1.2 クローキング

攻撃プログラムが、クライアントの環境によって動作を

```
eval(unescape("%6c%6f%63%61%74%69%6f%6e%2e%68%72%65%66%3d%22%68%74%74%70%3a%2f%2f%77%77%77%2e%65%78%61%6d%70%6c%65%2e%63%6f%6d%2f%22%3b")))
```

(a) エンコードされたコード、デコーダ、およびコード実行部

```
location.href="http://www.example.com/";
```

(b) デコードされた JavaScript コード

図 1 難読化 JavaScript コードの例

変えることがある。このことをクローキングと呼ぶ。例えば、Web ブラウザが特定の脆弱性を持つバージョンである場合のみ攻撃を行い、そうでない場合は何もしないといったように、動作を変える攻撃が存在する。クローキングには、クライアントへの応答を変えるサーバ側クローキングと、ブラウザ側でプログラムの処理流れを変えるクライアント側クローキングの 2 つがある。クローキングを行う攻撃プログラムでは、Web ブラウザのバージョンや使用しているプラグインの情報などのクライアントの環境情報を取得する。このことをフィンガープリンティングと呼ぶ。クライアント側クローキングにおけるフィンガープリンティングでは、JavaScript を用いて、Web ブラウザから提供される環境情報を取得する。

クローキングを行う攻撃プログラムの動的解析を行う場合、通常は、攻撃プログラムが対象としている環境でなければ攻撃が行われない。このため、攻撃プログラムが対象としている環境で攻撃プログラムを実行するか、または何らかの方法で攻撃を行わせるように細工を施す必要がある。このように、攻撃プログラムがクローキングを行う場合、攻撃を実行する条件を解析する手間が増えるため、クローキングは解析妨害の一種とみなせる。

2.2 既存の JavaScript コード解析手法

2.2.1 API Hooking

難読化 JavaScript コードは、エンコードされた JavaScript コード、デコーダ、およびコード実行部の 3 つからなるものが多く見られる。図 1 の例では、URL エスケープされた JavaScript コードを `unescape` 関数でデコードし、得られた (b) の JavaScript コードを `eval` 関数で実行している。このような難読化 JavaScript コードでは、デコードした JavaScript コードをコード実行部で文字列として `eval` などの関数に渡す。このため、どれほど複雑にエンコードされていても、図 2 のように `eval` などの関数をフックすることで、デコードされた JavaScript コードを得ることができる。このように、特定の関数をフックする手法を API Hooking [5][14] と呼ぶ。

文献 [5] では、コードを直接実行できる `eval` 関数や `setTimeout` 関数の他に、間接的にコードを実行できる `document.write` 関数など 10 種類の API を挙げ、これら

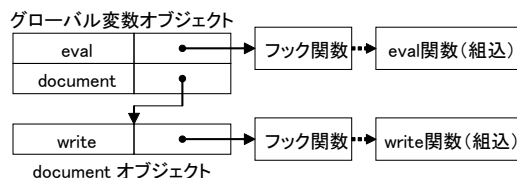


図2 コード実行APIのフック

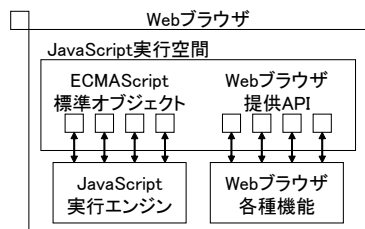


図3 WebブラウザにおけるJavaScriptのAPI

をフックすることで、デコードされたJavaScriptコードの取得を実現している。さらに、フィンガープリンティングやリダイレクトに用いられるAPIをフックすることで、フィンガープリンティングの検出と環境偽装、およびリダイレクト先URLの抽出を実現している。

2.2.2 ブラウザAPIのエミュレーション

JavaScriptコードの解析ツールとして、jsunpack-n[15]とJSDetox[16]がある。これらのツールは、Webブラウザを用いずにJavaScriptコードの動的解析を行っている。

WebブラウザにおけるJavaScriptのAPIについて説明する。WebブラウザにおけるJavaScriptでは、ECMAScript標準オブジェクト（以降、標準API）とWebブラウザ提供API（以降、ブラウザAPI）の2種類のAPIが、組み込みオブジェクトとして提供される（図3）。例えば、標準APIとして、配列を扱うためのArrayや文字列を扱うためのStringなどがある。一方、ブラウザAPIには、JavaScriptからHTTP通信を行うためのXMLHttpRequestやHTML要素の操作を可能にするHTMLElementなどがある。

上記解析ツールは、動的解析のために、JavaScript実行エンジンを利用している。このため、JavaScript実行空間には、Webブラウザと同様に標準APIが提供される。しかし、JavaScript実行エンジンはWebブラウザの機能を持たないため、ブラウザAPIは提供されない。そこで、上記解析ツールでは、一部のブラウザAPIを独自に定義し、ブラウザAPIのエミュレーションを行っている。

ブラウザAPIのエミュレーションでは、解析者が自由にAPIを定義できる。このため、例えば、ブラウザAPIにより提供される環境情報を自由に変えることができ、フィンガープリンティングを行うJavaScriptプログラムの動的解析に有効である。

2.3 課題

解析妨害が施されたJavaScriptコードに対する既存の解

析手法には、以下の課題がある。

（課題1）コード実行APIを使用しない種類の難読化 JavaScriptコードの解析

コード実行APIを使用する種類の難読化JavaScriptコードについては、API Hookingにより、デコードされたJavaScriptコードを容易に取得できる。しかし、コード実行APIを使用しない種類の難読化が施される例が存在する。このような難読化JavaScriptコードについては、既存手法では、より可読性の高いコードを得られない。このため、コード実行APIを使用する種類の難読化と比べて、解析が難しい。

（課題2）解析者の想定していないAPIへの操作の捕捉

API HookingやブラウザAPIのエミュレーションでは、解析者の想定していないAPIへの操作を捕捉できない。例えば、解析者が環境情報としてWebブラウザ名と言語設定の2つを想定し、APIをフックまたはエミュレートしている場合、プラグイン情報の取得操作が行われても、そのことに気づかない可能性がある。このことは、従来と異なる環境情報を取得するJavaScriptコードを解析する際に問題となる。

3. 難読化JavaScriptコード解析支援システム

3.1 目的と要件

本稿では、難読化JavaScriptコード解析支援システム（以降、提案システム）を提案する。提案の目的は、2.3節で述べた既存の解析手法における課題への対処である。また、解析に有用な情報を提示し、解析者による解析を支援することも目的の一つである。

前述の目的を達成するために、以下の要件が提案システムに求められる。特に（要件1）と（要件2）は、それぞれ（課題1）と（課題2）への対処である。

（要件1）可読化の難しいコードの解析

難読化JavaScriptコードに対し、より可読性の高いコードが容易に得られない場合においても、解析に有用な情報が十分に得られることが望ましい。

（要件2）解析者が想定していないAPIの操作の捕捉

解析者が想定していないAPIの操作を捕捉できるようにすることで、API操作に関する幅広い情報を得る。これにより、例えば、未知のフィンガープリンティングが行われ、想定していなかったAPIへのアクセスがあったことに気づくことができるようになる。

（要件3）Webブラウザを用いた正確な実行

ブラウザAPIのエミュレーションによる動的解析では、ブラウザAPIのエミュレーションが不完全である可能性があり、本来のWebブラウザでの挙動が再現できない可能性がある。このため、実際のWebブ

ブラウザを用いて解析できることが望ましい。

3.2 考え方

Web ブラウザにおける JavaScript には、コンソールプログラムにあるような標準入出力がない。また、イベント駆動型プログラミングを想定した設計になっている。このため、何らかの目的を持った JavaScript プログラムは、Web ブラウザから何らかのイベントによる入力を受け、その実行結果を Web ブラウザに反映する。つまり、JavaScript プログラムは、入出力インタフェース相当のブラウザ API を操作することにより、プログラムの目的を達成する。このことから、ブラウザ API の操作を捕捉し、そのログを取得することにより、JavaScript プログラムの挙動の把握でき、また解析に有用な情報を得ることができる。

以上のことを踏まえて、3.1 節で述べた要件を満たし、難読化 JavaScript を自動で動的解析するシステムを、以下のコンセプトに基づき実現する。

(1) 実環境で JavaScript コードを正確に実行するため、実際のブラウザを用いた動的解析を可能とする。これにより、目的とするブラウザとそのバージョンの環境において、詳細な情報を取得できる。

(2) ブラウザ API に対するすべてのプロパティ参照とメソッド呼び出しを漏れなく捕捉するために、Proxy オブジェクトを用いた手法を提案する。この手法を用いることで、ブラウザ API 操作をすべて捕捉でき、その内容をログに出力することで、対象の JavaScript コードが何を実行したのかを解析することを支援できる。また、捕捉するために行う処理は、ブラウザ API 呼び出し処理で、Proxy オブジェクトを呼び出すように変更する 200 行程度の JavaScript コードを、解析対象の JavaScript コードの前に 1 度挿入するだけでよい。この方法の利点は、解析対象毎に挿入するコードを修正する必要がなく、動的解析前に JavaScript コードを解析する必要がないことである。

(3) 再代入のできないグローバル変数として提供される一部の API は、単純な再代入による Proxy オブジェクトへの差し替えができない。そこで、変数への再代入と同等の効果を得られる変数の参照置き換え手法を提案し、再代入のできない API の操作捕捉を可能にする。変数の参照置き換えにより API 操作を捕捉するために行う処理は、解析対象の JavaScript コードに対し、50 行程度の JavaScript コードを挿入するだけでよい。これにより、グローバル変数へのアクセスや API Hooking で捕捉できなかった API の操作などを把握可能となり、より詳細な解析結果を得ることができる。

以上の対処により、ブラウザ API 操作の捕捉処理追加を自動化でき、ブラウザ API に対する操作を漏れなく捕捉できる。また、API Hooking で捕捉できない一部の API に対する操作を捕捉できる。

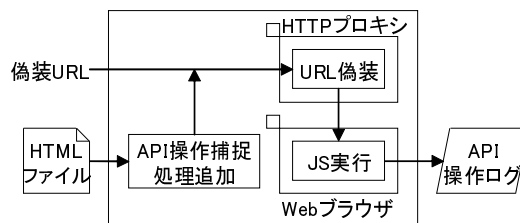


図 4 提案システムの全体図

表 1 主要 Web ブラウザにおける Proxy オブジェクトの実装状況

Web ブラウザ	実装バージョン
Internet Explorer	未実装
Microsoft Edge	12 以降
Mozilla Firefox	18 以降
Google Chrome	49 以降

3.3 提案システム

3.2 節で述べた考え方を踏まえて設計した提案システムの全体図を図 4 に示す。提案システムは、Web ブラウザでの動的解析をベースとしたシステムであり、解析者は、JavaScript コードを含む HTML ファイルおよび偽装したページ URL を入力し、出力としてブラウザ API 操作ログを得る。また、提案システムに HTML ファイルが入力されると、その HTML ファイルに API 操作捕捉処理を追加する。API 操作捕捉処理を加えた HTML ファイルを Web ブラウザで読み込み、JavaScript コードを実行することで、Web ブラウザのコンソールに API 操作ログが出力される。

Web ブラウザは、HTTP サーバとしての機能を持たせた HTTP プロキシを介して HTML ファイルを読み込む。HTTP プロキシを利用する理由は、ページ URL の偽装のためである。攻撃プログラムがページ URL をもとにクロッキングを行う可能性が考えられる。しかし、API Hooking では、自身のページ URL を提供している API をフックできないため、ページ URL を偽装することができない。ページ URL をもとにクロッキングを行う攻撃プログラムを動的解析するために、ページ URL の偽装が可能となるように提案システムを設計した。

提案システムは、Web ブラウザでの動的解析をベースとすることにより、(要件 3) を満たす。また、ブラウザ API の操作を捕捉し、そのログを取得して解析者に提示することにより、(要件 1) と (要件 2) を満たす。これにより、3.1 節で述べた要件が全て満たされる。

Proxy オブジェクトは、JavaScript の仕様を定める ECMAScript において、ECMAScript 2015[17] から策定された標準 API である。このため、Proxy オブジェクトが実装された Web ブラウザ (表 1) を使用する。

4. 実現方式

4.1 ブラウザ API の概要

3.2 節で述べた通り、提案システムでは、ブラウザ API

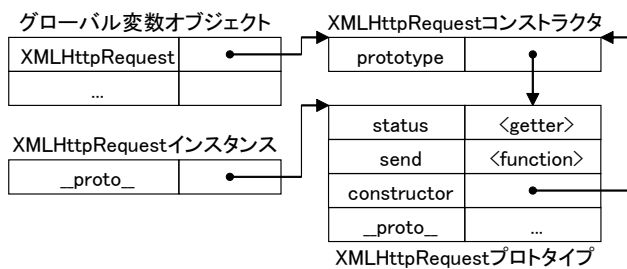


図 5 ブラウザ API の例 (XMLHttpRequest)

```

1 var request = new XMLHttpRequest();
2 request.open("GET", "http://www.example.com/", false);
3 request.send();
4 if (request.status === 200)
5     console.log(request.responseText);

```

図 6 XHR の使用例

の操作を捕捉する。JavaScript は、プロトタイプベースオブジェクト指向言語である。C++や Java のようなクラスベースオブジェクト指向言語と異なり、クラス概念が存在しない。このため、本稿では説明上、特定の操作に関する一連の API 群をクラスと呼ぶ。

ブラウザ API の一つである XMLHttpRequest (図 5) を例に、ブラウザ API の概要および関連する言語仕様について説明する。JavaScript から HTTP 通信を行うためのクラスとして、XMLHttpRequest (XHR と略記) がある。ブラウザ API として提供されるクラスは、基本的に、そのクラスのインスタンスを生成するためのコンストラクタ (関数)、そのクラスのインスタンスの雛形であるプロトタイプ、および各種メソッド (関数) の 3 つから成る。例えば、XHR クラスは、XHR コンストラクタ、XHR プロトタイプ、およびメソッド (open や send など) から成る。コンストラクタは、prototype プロパティを持ち、そのクラスのプロトタイプを値として持つ。

ここで、XHR を使用する JavaScript コードの例を図 6 に示す。1 行目で、XHR コンストラクタにより XHR インスタンスを生成し、request 変数に格納している。このとき、まず XHR の名前解決が行われ、グローバル変数オブジェクトから XHR コンストラクタを得る。次に、new 構文により、XHR コンストラクタに対して内部関数 [[Construct]] が呼び出される。この結果、XHR インスタンスとして空のオブジェクトが生成され、その._proto_プロパティに XHR コンストラクタの prototype プロパティの値である XHR プロトタイプが設定される。

2 行目と 3 行目では、request 変数に格納された XHR インスタンスに対し、open メソッドと send メソッドを呼び出している。このとき、send メソッドの場合、まず XHR インスタンスに対して内部関数 [[Get]] を用いて send の名前解決が行われる。XHR インスタンスは、send プロパ

ティを持っていないため、._proto_プロパティが指すオブジェクトに対して再帰的に名前解決が行われる。この結果、XHR プロトタイプに send プロパティが見つかり、その send プロパティが持つ関数 (メソッド) を得る。最終的に、得た関数 (メソッド) に対して内部関数 [[Apply]] が呼ばれ、その関数 (メソッド) の処理が実行される。4 行目では、同様に内部関数 [[Get]] を用いて status の名前解決が行われ、XHR プロトタイプの status プロパティの値を得る。この時、status プロパティにはゲッター関数が設定されており、この関数を実行した結果が status プロパティの値として得られる。

4.2 Proxy オブジェクトによるブラウザ API 操作の捕捉

提案システムでは、Proxy オブジェクトを利用することにより、ブラウザ API 操作の捕捉を実現する。具体的には、ブラウザ API の操作をブラウザ API (オブジェクト) に対する内部関数呼び出しと定義し、これらの内部関数を捕捉する。Proxy オブジェクトは、JavaScript のオブジェクトに対する基本的な操作を内部関数呼び出しレベルで捕捉することができる。

ブラウザ API 操作を捕捉するために、以下を実施し、クラスを構成するコンストラクタ、プロトタイプ、および各種メソッド (関数) へのアクセスを、すべて Proxy オブジェクトを介するようにする。

- (1) 以下に挙げるコンストラクタへの参照をあらかじめすべて Proxy オブジェクトに差し替える。
 - (a) グローバル変数オブジェクトの該当プロパティ
 - (b) プロトタイプの constructor プロパティ
- (2) 以下に挙げるプロトタイプへの参照をあらかじめすべて Proxy オブジェクトに差し替える。
 - (a) コンストラクタの prototype プロパティ
 - (b) 他プロトタイプの._proto_プロパティ
 - (c) あらかじめインスタンスとして提供されるオブジェクトの._proto_プロパティ
- (3) Proxy オブジェクトが内部関数呼び出しを捕捉したとき、以下を実施する。
 - (a) 対象オブジェクトに対する操作情報として、内部関数の情報を出力する。
 - (b) 返却値が関数 (メソッド) であれば、その関数 (メソッド) に対する Proxy オブジェクトを返却する。
 - (c) 返却値が捕捉すべきクラスのインスタンスであれば、そのインスタンスの._proto_プロパティを Proxy オブジェクトに差し替える。

Proxy オブジェクトを XMLHttpRequest (XHR) に適用した例を図 7 に示し、図 6 のコードを用いて説明する。まず、1 行目では、XHR の名前解決の結果、XHR コンストラクタの代わりに Proxy オブジェクトを得る。これによ

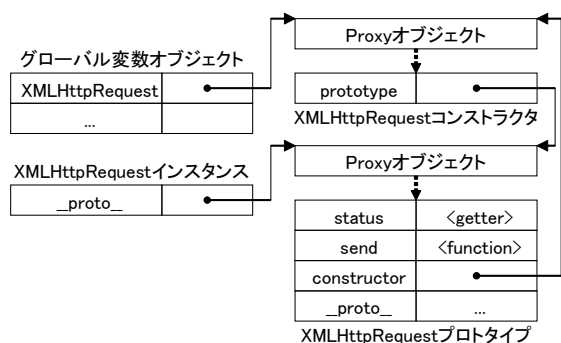


図 7 XMLHttpRequest に Proxy オブジェクトを適用した例

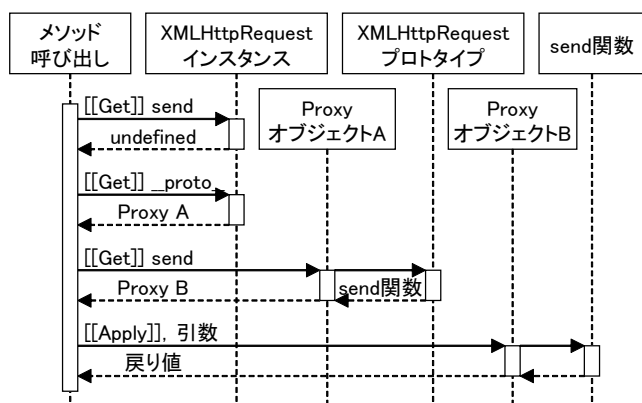


図 8 Proxy オブジェクト適用時の send メソッド呼び出しの流れ

り、new 構文でインスタンスを生成する際に、Proxy オブジェクトに対して内部関数 `[[Construct]]` が呼び出される。この Proxy オブジェクトでは、呼び出された内部関数の情報をログとして出力し、本来の XHR コンストラクタに対して内部関数を呼び出す。また、生成された XHR インスタンスの `__proto__` プロパティには、XHR コンストラクタの `prototype` プロパティに設定されている Proxy オブジェクトが設定される。

2, 3, 4, および 5 行目では、プロパティの名前解決のために `__proto__` プロパティを参照し、XHR プロトタイプではなく、Proxy オブジェクト A に対して内部関数 `[[Get]]` を呼び出す。Proxy オブジェクト A では、呼び出された内部関数の情報をログとして出力し、本来の XHR プロトタイプに対して内部関数を呼び出す。特に 2, 3 行目のメソッド呼び出しでは、内部関数 `[[Get]]` は、本来の関数（メソッド）の代わりに Proxy オブジェクト B を返す。Proxy オブジェクト B に対し、関数実行のために内部関数 `[[Apply]]` が呼び出され、Proxy オブジェクト B は、同様に情報をログとして出力する。3 行目の `send` メソッド呼び出しについて、Proxy オブジェクトを適用した際のシーケンス図を図 8 に示す。以上のようにすることで、ブラウザ API 操作を捕捉し、そのログを取得できるようになる。

4.3 Proxy オブジェクト適用の限界

4.2 節で述べたブラウザ API 操作の捕捉手法には、限界がある。Web ブラウザにより提供される `window` オブジェクトは、その `__proto__` プロパティを書き換えられない。このため、一部のブラウザ API (`window.addEventListener` メソッドなど) を捕捉できない。この問題への対処として、API Hooking の併用が有効である。

また、インスタンスとして提供されるブラウザ API である `window` オブジェクトと `location` オブジェクトは、それぞれのプロトタイプが持たないプロパティを各インスタンスが持つ。`window` オブジェクトは、グローバル変数オブジェクトとしての役割を持ち、そのプロパティ参照を捕捉することで、グローバル変数への参照を捕捉できる。`location` オブジェクトは、現在のページ URL に関する情報を提供しており、特にリダイレクトのために利用される。しかし、これらのオブジェクトを指すグローバル変数である `window` 変数と `location` 変数は、再代入が禁止されており、Proxy オブジェクトに差し替えられない。また、グローバル変数オブジェクトそのものの置き換えもできない。グローバル変数に再代入できない問題は、4.4 節で説明する変数の参照置き換え手法により対処する。

4.4 変数の参照置き換え

再代入のできない API 操作の捕捉を可能にするため、変数の参照置き換えにより、変数への再代入と同等の効果を得る。

JavaScript における変数の名前解決について説明する。JavaScript の実行コンテキストには、関数の外側のグローバル実行コンテキストと、呼び出された関数ごとの関数実行コンテキストがある。各実行コンテキストは、それぞれ変数オブジェクトを持ち、グローバル変数や関数内変数を管理する。例えば、グローバルコンテキストで関数 A を呼び、関数 A 内で変数 `window` が参照されたとする。このとき、`window` の名前解決のため、現在の実行コンテキストが持つ関数 A の変数オブジェクトから `window` を探す。関数 A の変数オブジェクトに `window` がなければ、上位の実行コンテキストであるグローバル実行コンテキストの変数オブジェクト、つまりグローバル変数オブジェクトから探す。

4.3 節で説明した通り、グローバル変数として提供される `window` 変数と `location` 変数は、再代入ができない。これらの変数に Proxy オブジェクトを適用するために、前述した変数の名前解決の仕様を利用する。具体的には、解析対象の JavaScript コードをまるごと 1 つの関数内に入れ、その関数で関数内変数として `window` 変数と `location` 変数を定義する。これにより、関数内で例えば `window` 変数が参照された場合、先にその関数の変数オブジェクトから探索されるため、関数内変数の `window` 変数の値が得られる。この手法で、図 9 のように、`window` 変数と `location` 変

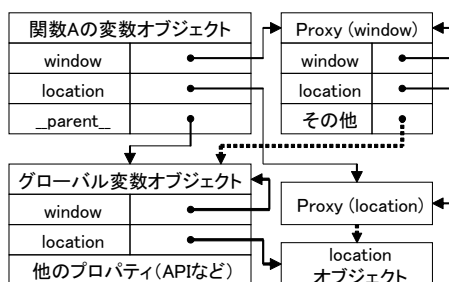


図 9 変数の参照置き換えによる Proxy オブジェクト適用

数の値にそれぞれ Proxy オブジェクトを代入する。

Web ブラウザの JavaScript においては、window オブジェクトがグローバル変数オブジェクトとしての役割を果たす。このため、通常、window オブジェクトの window プロパティは、グローバル変数 window と同じく window オブジェクト自身を指す。同様に、window オブジェクトの location プロパティは、location オブジェクトを指す。このことを考慮し、window オブジェクトに対する Proxy オブジェクトは、window プロパティや location プロパティを参照されたとき、各 Proxy オブジェクトを返すようにする。以上により、window 変数と location 変数が指していた window オブジェクトと location オブジェクトに対する操作を捕捉できるようになる。

本節で述べた変数の参照置き換え手法は、実行コンテキストが変わることによる副作用がある。具体的には、解析対象の JavaScript コードのうち、本来グローバル実行コンテキストで実行されるコードが、関数実行コンテキストで実行される。このため、元々グローバル変数として宣言していた変数が関数内変数になり、JavaScript コードによってはプログラムの挙動が変わる。このことを踏まえ、解析対象の JavaScript コードを修正したり、本手法を適用せずに解析するなど、場合に応じて解析する必要がある。

また、JavaScript には、window 変数を参照せずにグローバル変数オブジェクトの参照を得る方法が存在する。このため、そのようなコードを含む JavaScript プログラムは、window オブジェクトと location オブジェクトに対する操作を漏れなく捕捉することができない。

5. 評価

5.1 評価内容

提案システムの有用性を示すために、可読化の難しいコードでもプログラムの挙動を把握できること、およびさらなる解析に有用な情報を得られることの2点を確認する。

評価方法として、実際の攻撃で用いられた難読化 JavaScript コードを提案システムに入力し、解析結果について考察する。JavaScript コードを提案システムで解析する際、可読化の難しいコードでもプログラムの挙動を把握できることを確認するために、JavaScript コードを事前に

手動解析しない。また、解析結果について、得られる情報がさらなる解析に有用かどうかに着目して考察する。

評価では、Web ブラウザとして、Google Chrome 60.0.3112.101 (64 ビット, Windows 版) を使用した。

5.2 評価結果

難読化 JavaScript コードを含む HTML ファイルを D3M データセット [18] のパケットキャプチャデータ*1から一つ抽出し*2, この HTML ファイルを提案システムに入力した。Web ブラウザで動的解析を行い、ブラウザコンソールに出力されたブラウザ API 操作ログを図 10 に示す。得られたログを見ると、特定の要素 ID の HTML 要素へのアクセス、HTML 要素が持つ情報の取得、および object 要素を含む HTML の書き出しが行われていることを一目で確認できる。ブラウザコンソールを操作してログを詳細表示すると、object 要素で SWF ファイルを読み込もうとしていることも確認できる。また、環境情報として、プラグイン情報 (特に Silverlight プラグイン) と User-Agent を取得していることも容易に確認できる。

別の難読化 JavaScript コードとして、Malware-Traffic-Analysis.net[6] から Magnitude EK によるサンプル*3を取得し、提案システムに入力した。得られたログを図 11 に示す。この例では、window[gjrbulhg(...)] が関数ではないというエラーにより、プログラムが動作を停止している。しかし、エラーメッセージから window[gjrbulhg(...)] が何であるかを読み取ることができず、本来は、さらなる解析のために JavaScript コードを手動解析する必要がある。そこで、エラーメッセージ直前の提案システムが出力したログを見ると、window.ScriptEngineBuildVersion へのアクセスがあり、その値が未定義であることがわかる。つまり、本評価に用いた Web ブラウザでは、window.ScriptEngineBuildVersion 関数が定義されていないということを推察できる。この情報を元に、この攻撃プログラムがどのような環境を想定しているかを分析したり、独自に関数を定義して攻撃プログラムの挙動を解析したりなど、さらなる解析につなげることができる。

以上より、提案システムによって、可読化の難しいコードでもプログラムの挙動を把握できることを示した。また、取得された環境情報の種類や未定義のブラウザ API へのアクセスを確認できることから、環境偽装による動的解析などのさらなる解析に有用な情報を得られることを示した。

6. おわりに

既存の JavaScript コードの解析手法における課題を述べ、難読化 JavaScript コード解析支援システムを提案し、

*1 MD5 ハッシュ値: 2563d0cfed67550a5ca940c74d91dd4a

*2 ファイル名: VF9QAxxhUUBkG.html

*3 ファイル名: 2016-07-25-Magnitude-EK-landing-page.txt

```

[[Get]] HTMLDocument.getElementById val: ▶ [f]
[[Call]] HTMLDocument.getElementById args: ▶ ["tTnfG"]
[[Get]] HTMLDocument.innerHTML val:
▶ ["27245m515i2q534t512q3p2qGGM3c00D3n2724272459 56 34...45585o53355r5e5952345a5f355t3n27245t27245t27CDF24"]
[[Get]] HTMLDocument.getElementById val: ▶ [f]
[[Call]] HTMLDocument.getElementById args: ▶ ["tTnfG"]
[[Get]] HTMLDocument.title val: ▶ ["LR2eKS3UX9vV1AZ04aDG4HZ06LAF2"]
[[Get]] HTMLDocument.write val: ▶ [f]
[[Call]] HTMLDocument.write args:
▶ ["<object classid='clsid:d27cdb6e-ae6d-11cf-96b8-444...!-[if !IE]>--</object><!--<![endif]>--</object>"]
[[Get]] Navigator.plugins val: ▶ [PluginArray]
[[Get]] PluginArray.Silverlight Plug-In val: ▶ [undefined]
[[Construct]] ActiveXObject ▶ ["AgControl.AgControl"]
[[Get]] ActiveXObject.isVersionSupported val: ▶ [undefined]
[[Get]] Navigator.userAgent val:
▶ ["Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit...L, Like Gecko) Chrome/60.0.3112.101 Safari/537.36"]

```

図 10 ブラウザコンソールに出力されたログ（動的解析結果）

```

[[Get]] window.ScriptEngineBuildVersion val: ▶ [undefined]
▶ Uncaught TypeError: window[gjR8u1hg(...)] is not a function
at Proxy.<anonymous> (2016-07-25-Magnitude-EK-more-html.html:339)
at 2016-07-25-Magnitude-EK-more-html.html:339
at 2016-07-25-Magnitude-EK-more-html.html:340

```

図 11 window オブジェクトに対するプロパティ参照のログ

既存の解析手法における課題へ対処した。

提案システムでは、ブラウザ API 操作を捕捉し、そのログを解析者に提示することで、解析者による解析を支援する。ブラウザ API 操作捕捉を実現方式として、Proxy オブジェクトを利用する方法を提案した。また、再代入のできないグローバル変数への対処として、変数の参照置き換え手法を提案した。評価では、提案システムにより得られる API 操作ログが、さらなる解析に有用な情報を含むことを示し、提案システムの有用性を示した。

参考文献

- [1] Cisco: Cisco 2016 Annual Security Report (online), available from (https://www.cisco.com/c/m/en_us/offers/sc04/2016-annual-security-report/) (accessed 2017-08-08).
- [2] Trustwave: 2017 Global Security Report (online), available from (<https://www2.trustwave.com/2017-Trustwave-Global-Security-Report.html>) (accessed 2017-07-07).
- [3] 高田雄太, 秋山満昭, 針生剛男: ドライブバイダウンロード攻撃に使用される悪質な JavaScript の実態調査, 電子情報通信学会技術研究報告, vol.113, no.502, ICSS2013-72, pp.59-64 (2014).
- [4] Symantec: Symantec Global Internet Security Threat Report: Trends for 2008, Volume XIV, April 2009 (online), available from (<https://www.symantec.com/security-center/archived-publications>) (accessed 2017-07-07).
- [5] 柴田龍平, 羽田大樹, 横山恵一: Js-Walker: JavaScript API hooking を用いた解析妨害 JavaScript コードのアナリスト向け解析フレームワーク, コンピュータセキュリティシンポジウム 2016 論文集, vol.2016, no.2, pp.951-957 (2016).
- [6] Malware-Traffic-Analysis.net (online), available from (<http://www.malware-traffic-analysis.net/>) (accessed 2017-08-08).
- [7] 玉田春昭, 中村匡秀, 門田暁人, 松本健一: Java クラスファイル難読化ツール DonQuixote, ソフトウェア工学の基礎: 日本ソフトウェア科学会 FOSE2006, pp.113-118 (2006).
- [8] 高森健太郎, 岩本 舞, 小島俊輔, 中嶋卓雄: マハラノビス距離を用いた難読化マルウェア JavaScript の検出, 情報処理学会研究報告, vol.2014-DPS-161, no.17, pp.1-7 (2014).
- [9] Takata, Y., Akiyama, M., Yagi, T., Yada, T. and Goto, S.: Website Forensic Investigation to Identify Evidence and Impact of Compromise, Proc. SECURECOMM 2016, pp.431-453 (2016).
- [10] 西尾祐哉, 廣友雅徳, 福田洋治, 毛利公美, 白石善明: 悪性 Web サイトを分析するためのマルチ環境解析における通信ログ解析の効率化, コンピュータセキュリティシンポジウム 2016 論文集, vol.2016, no.2, pp.496-502 (2016).
- [11] Xu, W., Zhang, F. and Zhu, S.: JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code, Proc. CODASPY'13, pp.117-128 (2013).
- [12] Takata, Y., Akiyama, M., Yagi, T., Yada, T. and Goto, S.: Fine-Grained Analysis of Compromised Websites with Redirection Graphs and JavaScript Traces, IEICE Transactions on Information and Systems, vol.E100-D, no.8, pp.1714-1728 (2017).
- [13] Low, D.: Protecting Java Code via Code Obfuscation, Crossroads, vol.4, no.3, pp.21-23 (1998).
- [14] Pellegrino, G., TschürtzEric, C. and Rossow, B.: jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications, Proc. 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2015), pp.295-316 (2015).
- [15] Blake Hartstein: jsunpack — a generic JavaScript unpacker (online), available from (<http://jsunpack.jeek.org/>) (accessed 2017-07-07).
- [16] Relentless Coding: JSDetox (online), available from (<http://www.relentless-coding.org/projects/jsdetox/>) (accessed 2017-07-07).
- [17] Ecma International: ECMAScript® 2015 Language Specification (online), available from (<http://www.ecma-international.org/ecma-262/6.0/>) (accessed 2017-1-20).
- [18] 高田雄太, 寺田真敏, 村上純一, 笠間貴弘, 吉岡克成, 畑田充弘: マルウェア対策のための研究用データセット ~ MWS Datasets 2016~, 情報処理学会研究報告, vol.2016-CSEC-74, no.17, pp.1-8 (2016).