Computer Security Symposium 2017
23 - 25 October 2017

DBIを用いたPDFに含まれる悪性コード抽出手法の提案

今 健吾[1]　長瀬 智行[1]

概要：Adobe Reader は，PDF のファイル構造の曖昧性に関して寛容に作られており，形式の崩れた PDF ファイルであっても自動修復を行い開ける場合が多い．しかし，既存の悪性 PDF 分類器で使用される PDF パーサーは，形式の崩れた PDF の完全なパースには至っておらず，攻撃者によって巧みに細工された悪性 PDF を解析できないという問題がある．また，既存の悪性 PDF 分類器の多くはパーサーの問題について十分考慮されておらず，パースの失敗がシステム全体の精度に影響してしまう懸念がある．そこで本稿では，DBI (Dynamic Binary Instrumentation) を用いて Adobe Reader に直接解析コードを挿入し，メモリアクセストレースの取得及び解析を行うことで，JavaScript 実行関数を特定する手法を提案する．これにより，パーサーの性能に影響されることなく悪性 PDF から JavaScript コードを抽出でき，既存の JavaScript ベースの悪性 PDF 分類器に必要な JavaScript コード抽出工程を改善できる．

キーワード：悪性 PDF，埋め込み JavaScript，PDF parser confusion attack

A Malicious Code Extraction Method for PDF Files Based on DBI

Kon Kengo[1]　Nagase Tomoyuki[1]

Abstract: Adobe PDF Reader is able to adjust and view a PDF file which has an ambiguous PDF file’s structure, and it often performs automatic correction a malformed PDF file. However, the PDF parser that is used in existing malicious PDF classifiers doesn’t provide perfect features to extract data from malformed PDFs. Therefore, the recent advanced in creating malicious PDF files by attacker cannot be perfectly analyzed. Also, many existing malicious PDF classifiers do not fully consider the problem of the parser, and there is concern that the failure of the parsing method may affect the accuracy of the whole system. In this paper, a method for identifying a JavaScript execution function is proposed based on inserting a code directly into Adobe Reader using DBI (Dynamic Binary Instrumentation) to obtain a memory access trace log for analysis purposes. This method enables to extract JavaScript code from malicious PDF without being affected by the performance of the parser. It improves the JavaScript code extraction process that is required for existing JavaScript based malicious PDF classifier.

Keywords: Malicious PDF, Embedded JavaScript, PDF parser confusion attack

1. はじめに

1.1 背景

近年，特定の組織や個人を対象に情報窃取などを目的として行われる様々なサイバー攻撃が大きな社会問題となっている．とりわけ，攻撃用のファイルを添付したメールを特定の相手へ送信する標的型メール攻撃や，不正に改ざんされた Web サイトを閲覧したクライアント PC にマルウェアをダウンロードさせる Drive-by Download 攻撃がその典型である．これらの攻撃に悪用されるファイル形式の一つに PDF (Portable Document Format) がある．PDF は電子文書のフォーマットとして広く定着しており，Adobe Reader に代表される閲覧ソフトが多くのクライアント PC にインストールされている．そのため，閲覧ソフトの脆弱性を狙った攻撃の影響範囲は極めて大きいものとなる．現在も Adobe Reader やその後継製品の脆弱性が多数報告さ

[1] 弘前大学大学院理工学研究科
Graduate School of Science and Technology, Hirosaki University

© 2017 Information Processing Society of Japan

```
function spary() {
var shellcode = unescape("%uc92b%u1fb1%u0cbd%uc
garbage = unescape("%u9090%u9090%u9090%u9090%u9
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-a
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');
}
spary();
```

図 1: 悪性 PDF に埋め込まれた JS コードの例

れており，2016 年には 200 件以上の脆弱性が発見されている [1], [2]．また，個人や組織へのマルウェア感染を目的としたばらまき型メール攻撃において，PDF の悪用が近年再び増加しているという報告もある [3]．このような状況から，悪性 PDF による脅威は現在も続いており，その対策が求められている．

攻撃に利用するために悪意を持って作成された PDF ファイル (以下，悪性 PDF) では，不正な JavaScript(以下，JS) コードをファイル内に埋め込んだものが多く利用されている [4]．PDF の仕様 [5] では，文書の外観や機能性，インタラクティブフォームによる対話性の向上のために JS コードの埋め込みが許されており，悪性 PDF の多くは Adobe Reader に内蔵された JS 実行エンジンの脆弱性を悪用している．図 1 は悪性 PDF に埋め込まれた JS コードの例である．一般的にこのような JS コードの抽出は，悪性 PDF の解析や検知において重要な工程であり，既存の悪性 PDF 分類器の多くは，独自の PDF パーサーや，レンダリングエンジンの Poppler[6] をベースに作成された libpdfjs など，サードパーティの PDF パーサーを導入している場合が多い．しかし，攻撃者が巧みに細工することで，特定のパーサーでは解析不可能な悪性 PDF が作成可能であることが指摘されている [7]．加えて，既存の悪性 PDF 分類器の多くはパーサーの堅牢性について十分な配慮がされておらず，パースの失敗が検知システム全体の精度に影響してしまう可能性がある．このような状況により，悪性 PDF から JS コードを確実に抽出するための方法が必要とされている．

1.2 貢献

本稿では，DBI (Dynamic Binary Instrumentation) フレームワークである DynamoRIO[8] を使用することで，Adobe Reader に直接解析コードを挿入し，メモリアクセストレースの取得及び解析，関数フックポイントの特定を通じて，悪性 PDF に含まれる不正な JS コードを抽出する手法を提案する．これを利用することでパーサーの性能に影響されることなく悪性 PDF から JS コードを抽出することが可能になり，悪性 PDF 検知を行う多くの既存手法に必要な JS コード抽出工程の改善が期待できる．本稿の主要な貢献は以下のとおりである．

- 実行中の Adobe Reader に直接コードを挿入することで，PDF パーサーに依存しない JS コード抽出手法を提案した．
- Adobe Reader のバージョンに依存しない汎用的な手法を提案した．
- リバースエンジニアリングを行うことなく JS 実行関数を特定する手法を提案した．

2. 悪性 PDF の解析妨害技術

マルウェアに代表されるような，不正なファイルやコードは，解析を遅らせるために様々な解析妨害機能を備えている．悪性 PDF も例外ではなく，解析を遅らせる目的で解析妨害の仕組みを備えている場合が多い．例えば，改ざんされた Web サイトに埋め込まれる JS コードと同様に，PDF 内に埋め込まれた JS コードに対しても難読化を施すことが可能である．さらに，注釈や特定のページに情報を隠蔽する PDF 独自の仕組みを利用した難読化も存在しており，このような難読化が含まれていた場合 JS コードの抽出はさらに困難になる．

また，難読化以外の解析妨害手法として，悪性 PDF の解析や検知に使用される PDF パーサーの実装の不備を利用することで，PDF パーサーの正常な動作を妨害する，PDF parser confusion attack による問題が指摘されている [7]．以下では，PDF parser confusion attack の詳細について述べる．

2.1 PDF parser confusion attack

PDF parser confusion attack は論文 [7] で体系的に調査された悪性 PDF による解析妨害手法であり，PDF を細工することでパーサーの正常な動作を妨害する．この攻撃の目的は，悪性 PDF 分類器に含まれる PDF パーサーの処理を失敗させることである．現状の悪性 PDF 分類器の多くは PDF パーサーの能力に完全に依存しているが，PDF パーサーの堅牢性に関する十分な議論は行われていない．そのため，PDF パーサーの実装不備につけ込んだ攻撃が行われており，今後さらに増えることが懸念されている．

PDF parser confusion attack が有効な攻撃手法である背

© 2017 Information Processing Society of Japan

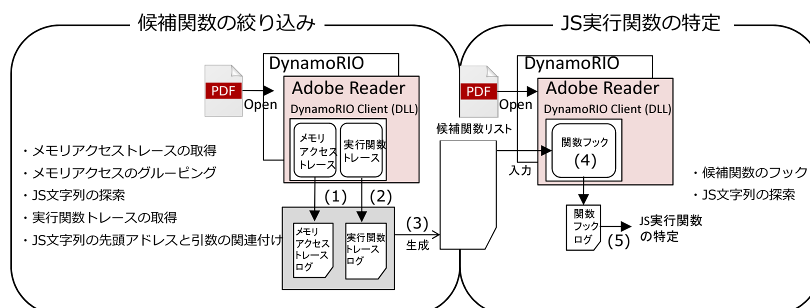

図 2: 提案手法の概要図

景には以下のような要因がある．

- PDF 仕様の一部に曖昧な定義が存在する
- 仕様から逸脱した PDF であっても Adobe Reader によって開ける場合が多い
- 既存の PDF パーサーは Adobe Reader に内蔵された PDF パーサーの動作を完全に再現できていない

すなわち，根本から解決するためには，Adobe Reader に内蔵された PDF パーサーを完全に再現する必要があるが，PDF 仕様の複雑さと，定期的に行われる Adobe Reader のアップデートを考慮すると現実的ではない．

2.2 PDF parser confusion attack の具体例

PDF parser confusion attack は，PDF 仕様と Adobe Reader による実装の矛盾点や，PDF パーサーの実装不備を利用しており，非常に多くのパターンが考えられるため，全てを把握することは困難である．そのため，以下では一部の例を示す．2.2.1 項，2.2.2 項は PDF 仕様の曖昧性を利用しており，2.2.3 項，2.2.4 項は PDF パーサーの実装の不備を利用している．また，これらの例を含めた PDF parser confusion attack の手法は個々では非常に単純なものではあるが，複数を組み合わせた場合，正常に動作する悪性 PDF 分類器は現状少ないのが実態である．

2.2.1 相互参照テーブルの除去

PDF 仕様では，ファイル内の各オブジェクトへのバイトオフセットを格納した，相互参照テーブルが全ての PDF ファイルに含まれると記載されている．しかし，PDF ファイルに相互参照テーブルがない場合，Adobe Reader は自動でこれを修復し再構築を行うという独自の処理を実装しているため，相互参照テーブルを元にオブジェクトをたどる PDF パーサーでは正常な動作が行われない．

2.2.2 不正なキーワードの利用

不正なキーワードに関するものとしては，一部の悪性 PDF が使用する `objend` キーワードが典型的な例である．PDF 仕様では，個々のオブジェクトは `obj` キーワードと `endobj` キーワードに囲まれるものと記載されているが，一部の悪性 PDF では不正な `objend` キーワードを用いており，この場合も Adobe Reader によって正しく開けることが知られている．

2.2.3 コメントの挿入

PDF 仕様では，%から始まる 1 行はコメントとして解釈され，文書構造に影響を与えないものとして扱われる．しかし，一部の PDF パーサーでは，トレーラや辞書内に挿入されたコメントを正しく処理できずに動作が停止してしまう．

2.2.4 多層のストリーム圧縮

PDF ファイルのオブジェクトでは，特定の文字列やバイナリデータを格納するためのストリームと呼ばれる領域が使用でき，その多くはフィルタによって圧縮されている．PDF 仕様では圧縮のための様々なエンコーディングがサポートされているが，使用頻度の高い一部のエンコーディングのみのサポートに留まっている PDF パーサーが多く存在している．そのため，ストリームに対して複数のエンコーディングを多層に適用された場合に正常に解析できないことがある．

3. DBI フレームワークを用いた JS 実行関数の特定手法の概要

3.1 概要

本研究では，JS コードを含む PDF を Adobe Reader で開くときに呼ばれる，JS 実行関数を自動で特定するシステムを開発した．JS 実行関数を特定することで，関数フックによる JS コードの抽出が可能になる．JS 実行関数は次のような定義とする．

- 引数の一つに，JS コードが文字列として格納されたメモリ領域の先頭アドレスを含む
- その JS コードを実行する機能を有する

特定のバージョンの Adobe Reader（または Adobe Reader にリンクされるライブラリ）に含まれる JS 実行関数は，一度特定できると，その後は関数フックによって JS コードの抽出を繰り返し行うことができる．また，関数フックは様々な方法で容易に実現可能であることから，JS 実行関数の特定が最も重要な作業であり，本稿ではこの作業を自動化するために有効な手法を提案する．

© 2017 Information Processing Society of Japan

```
[Format]
<pc>,<r/w>,<data size>,<data addr>,<data>
0x77d80970,w,4,0x002ffe48,0x003f12c7 |..?.|
0x77d80974,w,4,0x002ffe4c,0x0086a000 |....|
0x77d70ffe,r,4,0x0086d018,0x0086d000 |....|
0x77d71005,r,4,0x002ffe34,0x002ffe38 |8./.|
0x77d71008,r,4,0x77e1e64c,0x77d92f05 |./.w|
0x77d71010,w,4,0x002ffe3c,0x77d92f05 |./.w|
0x77d7101d,w,4,0x0086d000,0x002ffe38 |8./.|
0x77d70f99,r,4,0x002ffe4c,0x0086a000 |....|
0x77d70f9c,r,4,0x002ffe48,0x003f12c7 |..?.|
0x77d92108,r,4,0x002ffe30,0x0000001c |....|
0x77d9210c,w,4,0x002ffe30,0x002ffe40 |@./.|
0x77d92119,r,4,0x77e242d4,0x54b5160a |...T|
0x77d92123,w,4,0x002ffe14,0x549ae83a |:..T|
0x77d92127,w,4,0x002ffe18,0x002ffdf4 |../.|
```

図 3: メモリアクセストレースの例

3.2 提案手法の流れ

図 2 に本手法の概要図を示す．本手法は以下のような流れになる．

(1) 事前に作成した JS コードを含む良性 PDF を Adobe Reader で開き，この時のメモリアクセストレースを取得する．

(2) Adobe Reader(または Adobe Reader にリンクされるライブラリ) によって，呼び出される全ての関数のアドレスとその引数 (以下，実行関数トレース) を取得する．

(3) (1) のメモリアクセストレース及び，(2) の実行関数トレースのログを元に JS 実行関数の候補 (以下，候補関数) を絞り込み，候補関数リストとして保存する．

(4) 事前に作成した JS コードを含む良性 PDF を Adobe Reader で開き，(3) で生成された候補関数リストに含まれる関数をフックする．この時，引数で渡されたメモリアドレスの参照先を関数ごとに文字列で保存する．

(5) (4) で出力されたログから，開いた PDF ファイルに含まれた JS コードを探索し，見つかったログに対応する関数を JS 実行関数とする．

4. 提案手法の詳細

本手法は大きく，Adobe Reader の動的解析を伴う以下の二つの工程で構成されている．

- 候補関数を絞り込む ((1)，(2)，(3))
- 候補関数の中から JS 実行関数を特定する ((4)，(5))

本章では，それぞれの工程の詳細について説明する．

4.1 候補関数の絞り込み

4.1.1 メモリアクセストレースの取得

候補関数を絞り込むためにまず，メモリアクセストレースを取得する．JS コードが埋め込まれた良性 PDF を Adobe Reader で開き，そこで実行されるメモリアクセス命令を補足することで，メモリアクセストレースの取得を行う．メモリアクセストレースの例を図 3 に，図 3 に含まれる各項目の説明を表 1 に示す．

表 1: メモリアクセストレースの各項目

項目	意味
pc	メモリアクセス命令のアドレス
r/w	読み込み/書き込み
data size	読き込み/書き込みされるデータのサイズ
data addr	読み込み/書き込みされるデータのアドレス
data	読み込み/書き込みされるデータ

メモリアクセストレースを取得する目的は，これを元に JS 文字列が格納されていたメモリアドレスを推測するためである．このメモリアドレスを推測するためには次に，`data` が JS 文字列である，連続したメモリアクセスを特定する必要があり，メモリアクセスのグルーピングによってこれを実現する．

4.1.2 メモリアクセスのグルーピング

文字列処理などの連続したメモリ領域に格納されたデータへのアクセスは，アセンブラレベルであっても連続的に処理される場合が多い．そのため，Adobe Reader による JS 文字列に関係する処理も，連続したメモリアクセスを伴うものと考えられる．例えば，`app.alert("qqqq...");` で始まる JS コードが含まれる PDF ファイルを，Adobe Reader で開いた場合のメモリアクセスログの一部を図 4 に示す．`r/w` が w で，`data size` が 4，`data addr` が連続であることから，連続的なメモリ領域への書き込みを 4 バイト単位で行っている部分であることが分かる．しかし実際は，JS 文字列に関係するメモリアクセス命令の間に別のメモリアクセス命令が実行されることもある．例えば図 5 の場合，`pc` が 0x5f6cdec0 のメモリアクセスを見ると，`r/w` が w で，`data size` が 1，`data addr` が連続であり，`data` が `app.alert` になっていることが分かる．しかし，間に JS 文字列とは無関係なメモリアクセスが挟まっている．このような場合を考慮するため，`data` を対象に単純な文字列探索を行う前に，`pc` と `data addr` を元に，連続したメモリアクセスのグルーピングを行う．グルーピングは，`data addr` が連続でかつ `pc` が同じメモリアクセスを一つのグループとする．

4.1.3 JS 文字列の探索

グルーピングを終えたら，グループごとに JS 文字列の探索を行う．JS 文字列が見つかった場合，その時の `data addr` から先頭アドレスを取得する．同一の JS 文字列が複数箇所に格納されていることも考えられるため，ここで取得される JS 文字列の先頭アドレスも複数あると想定できる．例えば，図 4 の場合，0x0a4ec20d が JS 文字列の先頭

© 2017 Information Processing Society of Japan

```
[Format]
<pc>,<r/w>,<data size>,<data addr>,<data>
0x77d845a3,w,4,0x0a4ec20c,0x70706120 | app|
0x77d845a3,w,4,0x0a4ec210,0x656c612e |.ale|
0x77d845a3,w,4,0x0a4ec214,0x22287472 |rt("|
0x77d845a3,w,4,0x0a4ec218,0x71717171 |qqqq|
0x77d845a3,w,4,0x0a4ec21c,0x71717171 |qqqq|
0x77d845a3,w,4,0x0a4ec220,0x71717171 |qqqq|
0x77d845a3,w,4,0x0a4ec224,0x71717171 |qqqq|
0x77d845a3,w,4,0x0a4ec228,0x71717171 |qqqq|
0x77d845a3,w,4,0x0a4ec22c,0x71717171 |qqqq|
```

図 4: JS 文字列が含まれるメモリアクセストレースの例

```
[Format]
<pc>,<r/w>,<data size>,<data addr>,<data>
0x5f6cdec0,w,1,0x002fc791,0x00000061 |a...|
0x5f6d96a4,w,4,0x03e2602c,0x0402bb6e |n...|
0x5f6cdec0,w,1,0x002fc792,0x00000070 |p...|
0x5f6d96a4,w,4,0x03e2602c,0x0402bb6f |o...|
0x5f6cdec0,w,1,0x002fc793,0x00000070 |p...|
0x5f6d96a4,w,4,0x03e2602c,0x0402bb70 |p...|
0x5f6cdec0,w,1,0x002fc794,0x0000002e |....|
0x5f6d96a4,w,4,0x03e2602c,0x0402bb71 |q...|
0x5f6cdec0,w,1,0x002fc795,0x00000061 |a...|
0x5f6d96a4,w,4,0x03e2602c,0x0402bb72 |r...|
0x5f6cdec0,w,1,0x002fc796,0x0000006c |l...|
0x5f6d96a4,w,4,0x03e2602c,0x0402bb73 |s...|
0x5f6cdec0,w,1,0x002fc797,0x00000065 |e...|
0x5f6d96a4,w,4,0x03e2602c,0x0402bb74 |t...|
0x5f6cdec0,w,1,0x002fc798,0x00000072 |r...|
0x5f6d96a4,w,4,0x03e2602c,0x0402bb75 |u...|
0x5f6cdec0,w,1,0x002fc799,0x00000074 |t...|
```

図 5: 断続的なメモリアクセログの例

アドレスであると言える．

4.1.4 実行関数トレースの取得

本手法では，メモリアクセストレースの取得と同時に実行関数トレースの取得も行う．全ての関数の実行前に引数を取得するコードを挿入することで，呼び出される全ての関数のアドレスとその引数の記録を行う．実行関数トレースの例を図 6 に示す．取得する引数の数は任意の値を指定できるが，6 章で述べる評価では 3 つの引数で行った．

4.1.5 JS 文字列の先頭アドレスと引数の関連付け

最後に，JS 文字列の先頭アドレスを引数に持つ関数を実行関数トレースから抽出し，候補関数リストを生成する．ここまでで，JS 実行関数の候補となる関数を絞り込める．

4.2 JS 実行関数の特定

4.2.1 候補関数のフック

JS 実行関数を特定するために，再び JS コードが埋め込まれた良性 PDF を Adobe Reader で開き，与えられた候補関数を動的にフックするコードを Adobe Reader に挿入する．これにより，フックされた関数は本来の処理を始める前に，引数で渡されたメモリアドレスの参照先を文字列で保存する処理を行う．

```
[Format]
<func addr>,<arg1>,<arg2>,<arg3>
0x77d55b40,0x00000000,0x00000000,0x002fc778
0x77d55d20,0x00000018,0x00000000,0x00000018
0x5fa216b0,0x002fc7ac,0x08b7761a,0x00000002
0x5fa203c0,0x002fc7ac,0x6e9e1320,0x03e9fd48
0x5f675a20,0x6e9e13cc,0x08c5bbc0,0x08c8dd30
0x5f646850,0x0000001f,0x6e9e13cc,0x08c5bbc0
0x5f645920,0x0000001f,0x6e9e13cc,0x08c5bbc0
0x77d55b00,0x006b0000,0x00000000,0x0000001f
0x77d55b40,0x00000000,0x00000000,0x001801c8
0x77d55d20,0x0000001f,0x00000000,0x0000001f
0x5f646af0,0x08c63e60,0x08c5bbc0,0x0000001f
0x5fa21780,0x08c8dd44,0x08b7761a,0x00000002
0x5f646850,0x00000004,0x6e9e13f4,0x08c5bbc0
0x5f645920,0x00000004,0x6e9e13f4,0x08c5bbc0
```

図 6: 実行関数トレースの例

4.2.2 JS 文字列の探索

最後に，候補関数のフックによって得られたログから，開いた PDF ファイルに含まれた JS コードを探索し，見つかったログに対応する関数を JS 実行関数とする．

5. DynamoRIO による実装

本研究では，DynamoRIO を使用することで実行中のアプリケーションに対して解析コードの挿入を行った．DynamoRIO は動的なコード挿入を行うための API を複数提供しており，これらの API を利用することで DynamoRIO クライアントと呼ばれる独自の拡張機能を作成することができる．DynamoRIO クライアントは DLL にコンパイルされ，DynamoRIO によって解析対象アプリケーションのメモリ空間にマッピングされる．

本手法では，メモリアクセストレース及び実行関数トレースを行うプログラムと，与えられた複数の関数をフックするプログラムを DynamoRIO クライアントとして実装した．

5.1 メモリアクセストレースの取得

メモリアクセストレースは，メモリ参照を伴う mov や movs といった命令の直前に，参照アドレスを取得するコードを挿入することで実現している．例えば，`mov eax, [edx+0x30]` という命令の場合は `lea ebx, [edx+0x30]` のような命令を直前に挿入することで参照アドレスを取得できる．また `data` は，`mov ebx, [ebx];`

© 2017 Information Processing Society of Japan

表 2: JS 実行関数の特定結果

バージョン	JS 実行関数特定の成否
Adobe Reader 11.0.08	-
Adobe Acrobat Reader DC 2017.012.20095	-

表 3: JS 実行関数の特定結果 (再実行)

バージョン	JS 実行関数特定の成否
Adobe Reader 11.0.08	○
Adobe Acrobat Reader DC 2017.012.20095	○

`mov [mem_data_addr], ebx` というような一連の命令を挿入することで取得できる．このような命令の挿入は場合によってはクラッシュの原因になるため，適切にレジスタの退避や復元を行う必要がある．

5.2 実行関数トレースの取得

実行関数トレースでは，実行される全ての call 命令の直前に，呼び出し先のアドレスを求めるコードを挿入する．呼び出し先のアドレスは，call 命令のオペランドを取得し算出する．新たな call 命令が実行されるたびに呼び出し先のアドレスを算出し，そのアドレスに対応する関数をフックする．フック済みの関数が呼ばれた後は，引数の取得を行い，関数のアドレスと引数の値をログに出力する．これにより，最終的に呼び出された全ての関数のアドレスと引数が取得できる．

5.3 候補関数のフック

候補関数のフックは，フック対象の関数が含まれるモジュールがロードされたタイミングで候補関数リストを元にフックを行う．ここでは，与えられた関数をフックするだけであるため，必ずしも DynamoRIO を使用する必要はなく，Detours[9] などの汎用フックライブラリによって代用可能である．

6. 評価 1: JS 実行関数の特定

JS 実行関数の実装やモジュール内オフセットの値は，Adobe Reader のバージョンアップごとに変更される可能性がある．そのため，リバースエンジニアリングによって JS 実行関数を特定する場合は，バージョンアップごとに繰り返し行う必要がある．また，Adobe Reader のような大規模で複雑なアプリケーションの解析には多くの作業時間を必要とする．したがって JS 実行関数の特定は，Adobe Reader の特定のバージョンに依存せず行えることが理想である．

そこで評価 1 では，Adobe Reader のバージョン 11.0.08 と Adobe Acrobat Reader DC のバージョン 2017.012.20095 を対象に提案手法を適用することで有効性を検証する．

表 4: JS コードの抽出結果

全ての JS コードの抽出成功	70 検体
一部のみ JS コードの抽出成功	1 検体
全ての JS コードの抽出失敗	0 検体

6.1 評価環境

評価用 PC には，Intel Core i5-3320M (2.60GHz)，メモリ 8GB，OS は Windows 10 Pro を用いた．DynamoRIO はバージョン 7.0.0 を用いた．

6.2 結果

本手法によって JS 実行関数の特定を行った結果を表 2 に示す．表 2 に記したとおり，どちらの場合も JS 実行関数の特定ができなかった．つまり，候補関数リスト内に JS 実行関数がなかったと考えられる．

そこで，Adobe Reader の内部で JS 文字列をマルチバイト文字で扱っている可能性を考え，JS 文字列の先頭アドレスと同一の値のみではなく，先頭アドレスより上位 2 バイトまでを候補関数の対象にして再び本手法を適用した結果を 3 に示す．表 3 に記したとおり，どちらのバージョンにおいても JS 実行関数を特定できた．この結果から，本システムをマルチバイト文字に対応させる必要があることが分かる．しかし，この変更は候補関数の絞り込み時に僅かな変更を加えるだけである．

7. 評価 2: 悪性 JS コードの抽出

評価 2 では，悪性 PDF に埋め込まれた JS コードの抽出において，評価 1 で得られた JS 実行関数のフックが有効であるかを検証するため，悪性 PDF を用いた評価を行った．評価に使用する悪性 PDF は，MWS Datasets 2016[10] に含まれる D3M から抽出した 71 検体を対象にした．評価では，検体それぞれから手動で抽出した JS コードと，評価 1 で特定した Adobe Acrobat Reader DC の JS 実行関数をフックすることによって抽出された JS コードの比較を行った．

© 2017 Information Processing Society of Japan

```
3 0 obj
<< /Type /Action /JS 6 0 R /S /JavaScript >>
endobj
6 0 obj
<< /Filter [/FlateDecode /ASCIIHexDecode /LZWDecode /ASCII85Decode /RunLengthDecode] /Length
    10736 >>
stream
...
endstream
```

図 7: 多層のストリーム圧縮

```
InitAnnot()
SetHTTPMethods()
SetRSSMethods()
eval(unescape("%66%75%6E%63%74%69%6F%6E%20%66%57%52%45%65%28%4C%6A%59%46%66%5A%57%2C%4C%6B
%44%52%6E%29%7B%77%68%69%6C%65%28%4C%6A%59%46%66%5A%57%2E%6C%65%6E%67%74%68%2A%32%20%3C%20
%4C%6B%44%52%6E%29%7B%4C%6A%59%46%66%5A%57%2B%3D%4C%6A%59%46%66%5A%57%3B%7D%4C%6A%59%46%66
%5A%57%3D%4C%6A%59%46%66%5A%57%2E%73%75%62%73%74%72%69%6E%67%28%30%2C%4C%6B%44%52%6E%2F%32
%29%3B%72%65%74%75%72%6E%20%4C%6A%59%46%66%5A%57%3B%7D%66%75%6E%63%74%69%6F%6E%20%4B%66%59
%59%68%4A%67%4F%28%29%7B%76%61%72%20%68%4B%7A%42%3D%75%6E%65%73%63%61%70%65%28%22%25%75%43
%30%33%33%25%75%38%42%36%34%25%75%33%30%34%30%25%75%30%43%37%38%25%75%34%30%38%42%25%75%38
```

図 8: 悪性 PDF から抽出した JS コード

7.1 評価環境

評価用 PC には，評価 1 と同様に Intel Core i5-3320M (2.60GHz)，メモリ 8GB，OS は Windows 10 Pro を用いた．評価は VM 上で行い，ゲスト OS は Windows 7，DynamoRIO はバージョン 7.0.0 を用いた．

7.2 結果

JS コードの抽出を行った結果を表 4 に示す．使用した 71 検体中 70 検体の JS コードを正確に全て抽出でき，1 検体は一部の JS コードのみ抽出できた．使用した数検体には 2 章で述べた PDF parser confusion attack を使用しているものも含まれていた．その一例を図 7 に示す．この例では，多層のストリーム圧縮による解析妨害が使用されている．図 8 は評価 1 で特定した JS 実行関数のフックにより抽出した JS コードであり，Adobe Reader によって多層のストリーム圧縮が解除された後に正しく抽出されているのが確認できる．したがって，JS 実行関数は正確に特定できていると言える．なお，図 8 の JS コードに含まれる上 3 行のコードは Adobe Reader によって自動で追加される初期化処理であると考えられる．

一部のみの抽出に留まった検体は，注釈に JS コードを格納し，別の JS コードから呼び出す解析妨害手法を用いた悪性 PDF であったが，何らかの理由で JS 実行処理を行わなかったと考えられる．

8. 関連研究

本章では，PDF パーサーを使用しない JS コードの抽出に関連する研究を示す．MPScan[11] は，悪性 PDF の JS コードにおいて，シェルコード及びヒープスプレーがよく使用されることを利用して検知を行う悪性 PDF 分類器である．MPScan では JS コードの抽出に，Adobe Reader の JS 実行エンジンを直接フックする方法を使用している．しかし，JS 実行のフックポイントはリバースエンジニアリングによって特定しており，具体的な作業量や要した時間については述べられていない．

reference JavaScript extractor[7] は，Adobe Reader の JS 実行関数，PDF レンダリングの正常終了ポイント，処理エラーのハンドリングポイントの三つを特定することで，ほぼ自動で JS コードを抽出する手法を提案している．正常終了ポイントとエラーハンドリングポイントを特定しているため，Adobe Reader によって開くことができない PDF が対象の場合でも自動排除が行え，既存の悪性 PDF 分類器との連携に非常に有効である．しかし，JS 実行関数の特定に関しては，一部手動による静的解析を行っており，完全な自動化は実現できていない．これに対して本手法では，リバースエンジニアリングを行うことなく，JS 実行関数の特定を実現しているため，自動化が可能である．

© 2017 Information Processing Society of Japan

9. おわりに

本稿では，DBI フレームワークである DynamoRIO を用いて Adobe Reader に直接解析コードを挿入することで，メモリアクセストレースの取得及び関数フックを行い，Adobe Reader の JS 実行関数を特定する手法を述べた．これによって，パーサーの性能に影響されることなく，悪性 PDF から JS コードを抽出することが可能であることを示した．一つ目の評価では，4.1.5 項で述べた，JS 文字列の先頭アドレスと引数の関連付けの工程で，多少の誤差を許容するための調整が必要であることが判明した．そしてこの調整を行うことで，Adobe Reader の JS 実行関数を正確に特定できることを確認した．二つ目の評価では，特定した JS 実行関数のフックによって悪性 PDF から JS コードを抽出できることを実証した．今後は，現在の手法では自動での抽出が難しい，インタラクティブ操作に関連づいた JS コードを自動で抽出する手法の実現と，既存の悪性 PDF 分類器との連携のしやすさを視野に入れたシステムの開発を目指す．

参考文献

[1] ”Adobe Acrobat Reader Dc : Security Vulnerabilities Published In 2016,” http://www.cvedetails.com/vulnerability-list/vendor_id-53/product_id-32069/year-2016/Adobe-Acrobat-Reader-Dc.html

[2] ”Adobe Reader : Security Vulnerabilities Published In 2016,” http://www.cvedetails.com/vulnerability-list/vendor_id-53/product_id-15364/year-2016/Adobe-Reader.html

[3] ショートレポート 2017 年 5 月マルウェア検出状況：「上位 5 種のマルウェアはすべてばらまき型メール攻撃によるもの」「PDF ファイルからランサムウェア感染を狙ったメールが急増」「WannaCryptor 国内影響は他国に比べ局所的」, https://eset-info.canon-its.jp/malware_info/malware_topics/detail/malware1705.html

[4] Lin, K. C. D. Y. D. (2014). Advanced Persistent Threat: Malicious Code Hidden in PDF Documents.

[5] ”PDF Reference,” http://www.adobe.com/devnet/pdf/pdf_reference.html, 2008.

[6] ”Poppler,” https://poppler.freedesktop.org

[7] Carmony, C., Hu, X., Yin, H., Bhaskar, A. V., & Zhang, M. (2016, February). Extract Me If You Can: Abusing PDF Parsers in Malware Detectors. In NDSS.

[8] ”DynamoRIO,” http://www.dynamorio.org

[9] Hunt, G., & Brubacher, D. (1999). Detours: Binaryinterception ofwin 3 2 functions. In 3rd usenix windows nt symposium.

[10] 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コンピュータセキュリティ（CSEC），Vol.2016-CSEC-74, No.17, pp. 1-8, (2016)

[11] Lu, X., Zhuge, J., Wang, R., Cao, Y., & Chen, Y. (2013, January). De-obfuscation and detection of malicious PDF files with high accuracy. In System sciences (HICSS), 2013 46th Hawaii international conference on (pp. 4890-4899). IEEE.

© 2017 Information Processing Society of Japan