2013 46th Hawaii International Conference on System Sciences

De-obfuscation and Detection of Malicious PDF Files with High Accuracy

Xun Lu
Institute of Network Science and Cyberspace, Tsinghua University, China
University of Illinois, Urbana-Champaign, USA
xunlu2@illinois.edu

Jianwei Zhuge
Institute of Network Science and Cyberspace, Tsinghua University, China
zhugejw@cernet.edu.cn

Ruoyu Wang
Institute of Network Science and Cyberspace, Tsinghua University, China
fish.thss@gmail.com

Yinzhi Cao
Northwestern University, USA
yinzhicao2013@u.northwestern.edu

Yan Chen
Northwestern University, USA
ychen@northwestern.edu

Abstract

Due to its high popularity and rich functionalities, the Portable Document Format (PDF) has become a major vector for malware propagation. To detect malicious PDF files, the first step is to extract and de-obfuscate JavaScript codes from the document, for which an effective technique is yet to be created. However, existing static methods cannot de-obfuscate JavaScript codes, existing dynamic methods bring high overhead, and existing hybrid methods introduce high false negatives.

Therefore, in this paper, we present MPScan, a scanner that combines dynamic JavaScript de-obfuscation and static malware detection. By hooking the Adobe Reader's native JavaScript engine, JavaScript source code and op-code can be extracted on the fly after the source code is parsed and then executed. We also perform a multilevel analysis on the resulting JavaScript strings and op-code to detect malware. Our evaluation shows that regardless of obfuscation techniques, MPScan can effectively de-obfuscate and detect 98% malicious PDF samples.

1. Introduction

Since launched in 1993, the Portable Document Format (PDF) has become the de facto standard for electronic file exchange. The ubiquitous-ness of PDF over the Internet has rendered PDF as a major vector for malware distribution. The 2010 Symantec Security Report[1] shows that PDF files were the most successful attacking vectors to serve malicious content on the Web. Besides being served on rogue website in a drive-by-download attack[2], malicious PDF documents can also be served via a variety of ways with the most notorious method being Spear Phishing[3]. By applying some social engineering techniques in the spam email (e.g. News stories of the latest Presidential

1530-1605/12 $26.00 © 2012 IEEE
DOI 10.1109/HICSS.2013.166

IEEE computer society

Campaign), users are solicited to open the malicious PDF attachment and get infected.

Beside the popularity of PDF file format, the other important reason that accounts for the proliferation of PDF malware is the complexity of rich features allowed by Adobe Reader (the most widely used PDF viewer), notably its support for JavaScript. JavaScript codes embedded inside PDF files are executed in Adobe's own JavaScript engine. This feature boosts the functionality of PDF document in the means of allowing PDF to perform sophisticated tasks such as form validation and calculation. However, it also bestows upon attackers the power to run arbitrary code by exploiting vulnerabilities in the Adobe JavaScript engine. Furthermore, most of JavaScript codes embedded in malicious PDFs are extensively obfuscated to the extent that it hinders code analysis, and thus anti-virus applications are not able to cope with even the most well-known PDF vulnerability.

In this paper, we present the design and implementation of MPScan (Malicious PDF Scanner), a scanner that de-obfuscates and detects malicious JavaScript code embedded in PDF files. Through dynamically hooking Adobe Reader's JavaScript Engine, MPScan can extract de-obfuscated JavaScript source code as well as Op-code stream (an intermediate code generated while parsing), and then statically analyze them for malware detection. MPScan can reliably de-obfuscate JavaScript code because no matter how much a piece a code is obfuscated, it has to be transformed to the de-obfuscated form for execution. So as long as the code is executed, the hook points in the JS engine will deliver the de-obfuscated JavaScript source code. MPScan's multi-level detection consists of Shellcode/Heapspray detection based on Strings in the extracted JavaScript code as well as the Op-code Signature Detection based on the extracted Op-codes stream. The de-obfuscated JavaScript source code extracted by MPScan also provides the understandable materials for forensic analyzer. A preliminary evaluation result shows that MPScan can accurately detect a wide range of Malicious PDF, regardless of the obfuscation techniques.

In summary, this paper provides the following contributions:

- Designing a novel approach to de-obfuscate JavaScript code embedded in PDF by hooking the native JS execution engine. This approach is robust against even previously unknown obfuscation techniques.
- Designing a Multi-level malware detection scheme to monitor for both Shellcode/ Heapspray in strings and malicious behavior demonstrated via Op-code, thus providing a more reliable detection.
- Combining dynamic JavaScript code de-obfuscation with static malware detection in an effort to balance the detection effectiveness and performance overhead.

2. Background and Related Works

In this section, we summarize the main features of PDF standard. Then we present an overview of existing works of detecting malicious PDF.

2.1. PDF in A Nutshell

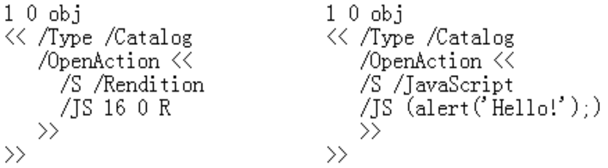
According to PDF specification[5], each valid PDF file has four main sections:

1. **Header**: One line statement containing "%PDF" followed by the version number;
2. **Body**: PDF objects that make up of the document content. Embedded files are also included in this section;
3. **Cross-reference Table**: Offsets of each indirect PDF objects within the file.
4. **Trailer**: Offsets of the cross-reference table and certain special objects.

The parsing of PDF file starts by first checking the version number in the header section, then it retrieves the offsets of the Cross-reference table and some special objects such as the *catalog* object from the trailer section. The body of a PDF document is constructed as a hierarchy of objects linked together in a meaningful way to describe pages, form, annotations, etc. Objects in PDF body are assigned a unique identifier in the form of “1 0 obj” where the first number indicates the object number, the second number indicate the generation number and the“obj” indicate that the identifier represent an object. This object can be referenced by “1 0 R”, the “R” character tells the viewer that this is an *indirect reference*. There are eight basic type of objects in PDF standard: Boolean, Integer, Strings, Names, Arrays, Dictionaries, Streams, Null. Note that Dictionaries are collections of key-value pairs with keys being names and values being any type of PDF object. Streams are dictionary objects followed by a sequence of bytes enclosed between keywords **stream** and **endstream.** These bytes can be encoded or compressed to represent large objects.

2.2. JavaScript in PDF

Even though inclusion of JavaScript in PDF can be achieved in various ways, these scripts all come down to be the value of the /JS keyword in some object’s dictionary. The value of /JS keyword can be a literal string containing JavaScript codes as well as an indirect reference pointing to another object containing the literal JavaScript codes. In the latter case, the codes can be compressed or encrypted in a stream of the referenced object.



```
1 0 obj                          1 0 obj
<< /Type /Catalog                << /Type /Catalog
   /OpenAction <<                   /OpenAction <<
     /S /Rendition                    /S /JavaScript
     /JS 16 0 R                       /JS (alert('Hello!');)
   >>                               >>
>>                               >>
```

Figure 1. Sample constructs of JavaScript in PDF

Before execution, JavaScript in a PDF document has to be included in an *action dictionary*. Such dictionary has the /S keyword that may have the value /JavaScript and /Rendition, both of which are also dictionaries themselves that have the keyword /JS. The /JavaScript and /Rendition keywords can be found at the following locations:

- The Catalog dictionary’s /AA entry may define an additional action specified by a JavaScript action dictionary.
- The Catalog dictionary’s /OpenAction entry may define an action to be taken after a document is opened.
- The document’s name entry may contain an entry ‘JavaScript’ that maps name strings to document–level JavaScript action dictionaries for execution after a document is opened.
- The document’s outline hierarchy may contain references to JavaScript action dictionaries.
- Pages, file attachments and forms may contain references to JavaScript action dictionaries.

Besides being embedded within PDF document, JavaScript codes may also reside on a remote location and can be retrieved by the /URI or /GoTokey directives.

In a survey that we conducted using the CVE database about the techniques that attackers use to exploit vulnerabilities in PDF, almost 96% of exploitations involved JavaScript to various extents. The vulnerabilities in Adobe Reader that are related to JavaScript can be classified into two categories. The first class of vulnerabilities arises from bugs in the implementation of the Adobe JavaScript API, and they account for 33% of all JavaScript related PDF exploitations. The second class of vulnerabilities is triggered in non-JavaScript features in PDF but it requires JavaScript to prepare the environment for exploitation (e.g. Heapspray). Our Op-code signature matching detection component can address the former class of vulnerabilities, and the latter class can be handled by our Shellcode/Heapspray detection component. Therefore, MPScan has a broad detection range that covers all kinds of malicious JavaScript in PDF.

2.3. Related Works

A number of approaches and tools have been proposed in recent years to de-obfuscate and detect malicious PDF document. We will briefly introduce the most relevant ones and compare them to our work.

Fully Static Method:

Fully static method was used in the early era of PDF malware analysis, and it features W.j.Li, et al.[6] and Z.shafiq et al.[15]'s research. But since malicious PDFs are nowadays extensively obfuscated, these approaches can hardly work. The most recent fully static work is PjScan[4] that takes the idea a step further by analyzing the token stream generated while the code is executing. However, it retrieves the token stream by hooking the SpiderMonkey[7] JS engine instead of the native Adobe JS engine; therefore it may not be able to deal with certain JavaScript method that existed only in the native environment. Since we hooked the native JavaScript engine in Adobe Reader, we have controls of all the methods.

Fully Dynamic Method:

CWSandbox[8] is the most prominent tool in this category. It literally launches the Adobe Reader to load the suspected PDF document in an emulated runtime environment, and then it detects malicious behavior by monitoring system calls and modifications. The problem with CWSandbox and dynamic tools in general is that an attack can be detected only if the vulnerable component targeted by the exploit is installed and correctly activated on the detection system. In addition, extra overhead is incurred to revert the sandbox environment to clean state.

Hybrid Emulated Method:

This category combines the advantages of both dynamic and static methods, and it is gradually becoming the mainstream method for malicious PDF analysis.

Major works in this category include the Honeynet Project's PDFphoneyC[9] and MDScan[10]. They both statically parse PDF document and retrieve JavaScript code. Then they feed JavaScript codes into an instrumented SpiderMonkey JS engine for malware detection. The problem with their approaches is that both their static code extraction and dynamic execution are performed in an emulated environment, which lacks some proprietary feature in the native Adobe environment. Thus it may lead to some undesirable outcome such as abrupt termination of Adobe Reader. Our work is built on idea of hooking the native JS engine in Adobe Reader, so we can avoid such troubles.

3. Design and Implementation

The overall architecture of document scanning in MPScan is shown in Figure 2.

Generally, it consists of the dynamic JavaScript code extraction module and the static multilevel malware detection module. The JavaScript extraction module retrieves the JavaScript source code and op-code from the PDF file during execution. The resulting source code and op-code are used as input to the malware detection module. The malware detection module is further divided into the Shell-code/Heapspray detection component that scans JavaScript Strings and the Op-code Signature Matching component that searches in the JavaScript op-code the signature of malicious JavaScript. If either of the two detection components identifies the PDF as malicious, it will be reported as malicious. We describe detailed description of design and implementation of each component as follow.

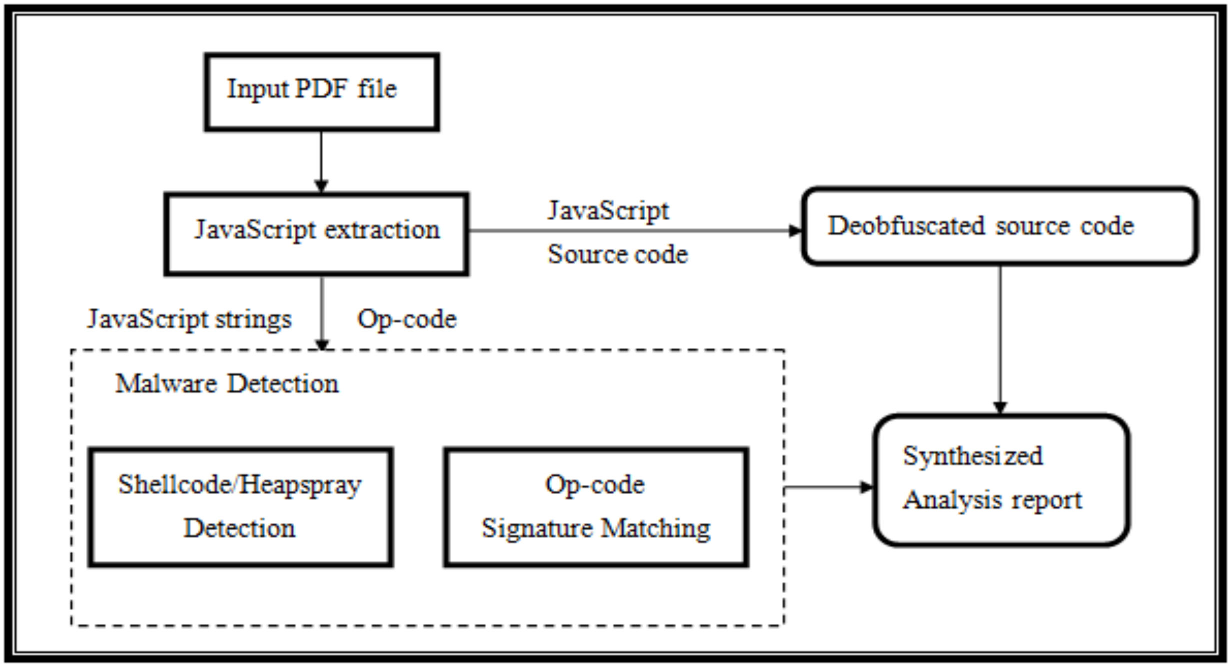


Figure 2. System architecture

3.1. JavaScript Extraction

An accurate and effective extraction method for JavaScript source code and op-code is the cornerstone for the success of MPScan since it relies on the extraction results to perform the multilevel malware detection. The main challenge for JavaScripts extraction is that they are extensively obfuscated, especially by those techniques that take advantage of the complexities and ambiguities provided by the PDF specification. Following are some common PDF oriented obfuscation techniques:

- Because Adobe Reader tries to render malformed PDF document that does not follow PDF standard strictly, attackers have some scope to use the subtleties to obfuscate the structure of the PDF file, thus hindering malware analysis.
- The rich JavaScript APIs provided by Adobe Reader can be used to access document specific objects, properties and methods. Therefore, attackers can hide some portions of JavaScript code or the data they use into PDF objects or dictionaries that are accessible through the Acrobat JavaScript API. These missing parts can be easily retrieved when the malicious code is executed.
- The stream object in PDF can store JavaScript source code and data. Multiple layers of different encoding method such as LZW, FlateEncode and CCITTFax can be applied to the stream. Therefore static decoding of the stream is difficult.

Many existing works such as PDFHoneyC and MDSCan take a static approach to the obfuscation problem by constructing a PDF document parser that searches for embedded JavaScript. However, this approach can hardly cover all PDF oriented obfuscation techniques due to the huge amount of Acrobat JavaScript API it has to simulate. And even if JavaScript source code segments were retrieved this way, they have to be put back in the right sequence before analyzed, which is very challenging for a static document parser. In case that JavaScript execution requires runtime user interaction, the static approach will have no way to put the pieces of JavaScript back together.

In light of the limitations of the static JavaScript extraction methods, we decided to retrieve the source code dynamically by hooking Adobe Reader's native JavaScript engine. By doing so, we also save the trouble of converting JavaScript source code to op-code, since the Op-code will be generated in the engine as the JavaScript executes and we only need output it. Figure3 shows how JavaScript in PDF is processed.

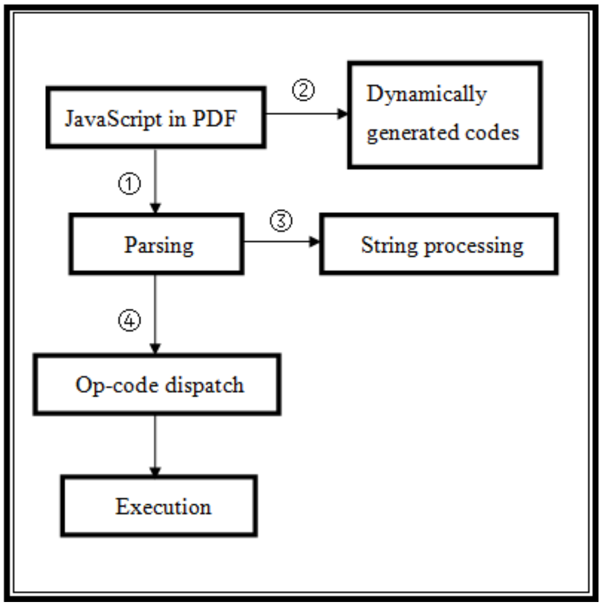


Figure 3. Process of JavaScript in PDF

Point① in Figure3 is the starting point of parsing where all JavaScript source code must go through before execution. Point② is used to process source codes that are dynamically generated by methods such as app.eval() and new function(). Hooking result of point① and point② combined will provide the complete de-obfuscated JavaScript source code.

Point③ is where JavaScript strings are created and manipulated. By hooking it, the JavaScript strings can be directly extracted.

Point④ is the execution point of op-code where each op-code is processed in a structure similar to switch(). By hooking it, we get the op-code flow.

Due to the fact that Adobe Acrobat is close-sourced, we resorted to reverse-engineering technology to locate these hooking points.

In this way, JavaScript source code, strings and op-code are extracted on the fly while the PDF embedded JavaScript executes. And the resulting source code and op-code are in the correct execution sequence.

3.2. Malware Detection

Having obtained the JavaScript source code and op-code, MPScan proceeds to malware detection. In order to achieve a broader range of detection, we take a multilevel detection scheme that detects shellcode/heapspray strings at the source code level and matches malicious op-code signature at the op-code level.

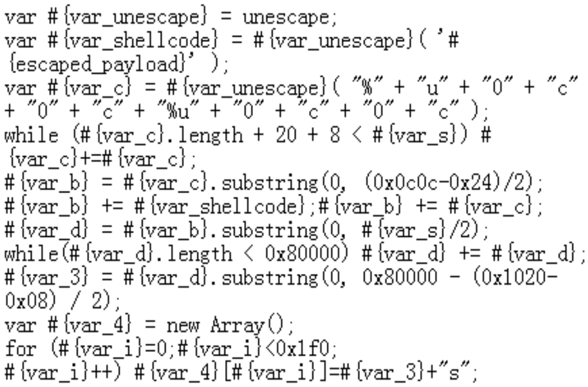
3.2.1. Shellcode/Heapspray Detection

The heapspray technique is widely used in malicious PDF to manipulate memory heap. Coupled with heap overflow, the malware can transfer the flow of control to embedded shellcode. The String data type is often used to carry shellcode/heapspray codes because in JavaScript it's the only data type that will not be garbage-collected even if it's not referenced.

To effectively detect shellcode/heapspray, we first divide the JavaScript strings into two groups by length. Strings that are between 32Bytes and 64Kbytes are checked for shellcode because 32Bytes is the shortest length for a known functioning shellcode and shellcodes longer than 64Kbytes are conspicuous thus not suitable for remote transferring. Strings longer than 64Kbytes are checked for Heapspray then.

The shellcode is detected using Libemu[11], which is a C library that detects shellcode using GetPC heuristics. Heapspray is detected by calculating the entropy of the strings. Since heapspray is consisted mostly of repeated characters, its entropy should be much lower than normal string. Zhijie Cetal[12]showed that setting entropy threshold to 1 would yield the best detection result. Therefore in MPScan, the entropy threshold is 1, which means any string with entropy less than 1 is flagged as heapspray.

As a proof of concept, we apply the Shellcode/Heapspray detection component to CVE-2010-3654, which exploits Flash embedded in PDF via crafted SWF content. It used heapspray to manipulate the heap as shown in Figure4.



```
var #{var_unescape} = unescape;
var #{var_shellcode} = #{var_unescape}( '#
{escaped_payload}' );
var #{var_c} = #{var_unescape}( "%" + "u" + "0" + "c"
+ "0" + "c" + "%u" + "0" + "c" + "0" + "c" );
while (#{var_c}.length + 20 + 8 < #{var_s}) #
{var_c}+=#{var_c};
#{var_b} = #{var_c}.substring(0, (0x0c0c-0x24)/2);
#{var_b} += #{var_shellcode};#{var_b} += #{var_c};
#{var_d} = #{var_b}.substring(0, #{var_s}/2);
while(#{var_d}.length < 0x80000) #{var_d} += #{var_d};
#{var_3} = #{var_d}.substring(0, 0x80000 - (0x1020-
0x08) / 2);
var #{var_4} = new Array();
for (#{var_i}=0;#{var_i}<0x1f0;
#{var_i}++) #{var_4}[#{var_i}]=#{var_3}+"s";
```

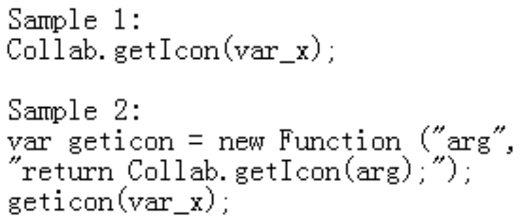
Figure 4. JavaScript in CVE-2010-3654 exploit

We submit a sample PDF of this exploit to our Shellcode/Heapspray detection component. String "var_4" with 200MBs size goes to the heapspray check routine and the result is positive. Thus even though this piece of JavaScript contains no exploitation of vulnerable Adobe JavaScript API, MPScan is still able to identify it as malicious based on the appearance of heapspray strings.

3.2.2. Op-code Signature Matching

Op-code is an intermediate instruction set generated by JavaScript engine for efficient execution. Because op-code is at a lower level than the source code, it reflects the actual behavior of the malware. No matter how malicious JavaScript is constructed at the source code level, they should have some distinctive behavior (e.g. exploiting vulnerabilities, retrieving files from remote locations). Therefore, the op-code stream of malicious JavaScript should have patterns that match malware op-code signature, which is a strong signal for identifying malicious PDF.

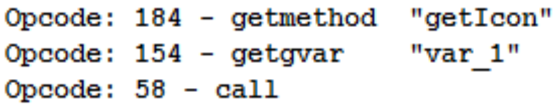
Op-code detection is especially useful in situations where different JavaScript codes would trigger the same vulnerabilities. For example, the two pieces of JavaScript code in Figure 5 both trigger CVE-2009-0927 that exploits the getIcon() method through stack-based buffer overflow. At the text level the codes look different, but they share the common op-codes showing in Figure 6.



```
Sample 1:
Collab.getIcon(var_x);

Sample 2:
var geticon = new Function ("arg",
"return Collab.getIcon(arg);");
geticon(var_x);
```

Figure 5.Different samples triggering the same Vulnerability



```
Opcode: 184 - getmethod  "getIcon"
Opcode: 154 - getgvar    "var_1"
Opcode: 58 - call
```

Figure 6. Common op-codes of the two samples

Then we can construct a deterministic finite automaton based on these op-codes to depict and match this exploit. And the automaton is the signature, as demonstrated in Figure 7.

Following the automaton transitions shown in Figure 7, malicious op-code signature can be easily matched.

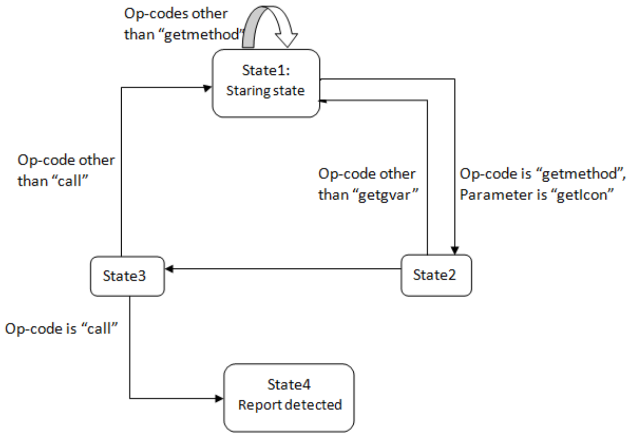

Op-codes other than "getmethod"
State1: Staring state
Op-code other than "call"
Op-code other than "getgvar"
Op-code is "getmethod", Parameter is "getIcon"
State3
State2
Op-code is "call"
State4 Report detected

Figure 7. Signature for CVE-2009-0927 exploit

4. Experimental Evaluation

In this section we present the experimental evaluation result of our prototype implementation. We have collected 198 various kinds of malicious PDF samples from Internet and malware repositories as well as individual sources. Combined with 9 distinctive malicious PDF samples generated from the Metasploit Framework[13], we obtained a testing set of 207 PDF documents that covered the majority types of PDF malware today.

4.1. Effectiveness

First, we tested the effectiveness of MPScan using these samples.

Table 1.MPScan detection results

Implementation	Detected	Undetected	Detection rate
Original implementation	186	21	89.9%
After implementing dummy functions for deprecated API	203	4	98%

As shown in Table 1, among the 207 PDF samples, 186(89.9%) were correctly identified as malicious. For the remaining 21 undetected malicious PDF: 3 of them try to exploit the flawed embedded TrueType font handling vulnerability (CVE-2010-0195) in Adobe Reader, which does not involve any JavaScript functionality; 1 of them does nothing else but extracting an embedded malicious PDF from within itself; The rest are due to the deprecation of some vulnerable Adobe JavaScript APIs in newer version of Adobe Reader (we hooked Adobe Reader 9.5.1, but some vulnerable API only exist in Adobe Reader versions older than 9.3.2), therefore their executions are terminated before JavaScript extraction is finished. After implementing dummy functions for the deprecated APIs, these samples are correctly classified as being malicious, thus improving our detection rate to 98%.

To test MPScan for false positive, we obtained 500 benign PDF documents by crawling the Alexa top 50 websites. This testing set has both PDF documents with and without JavaScript, and we have deliberately added obfuscation to some of the samples. It turns out that MPScan didn't make any misjudgment.

4.2. Performance

We measured the time MPScan takes to process PDF document. To get an idea about the impact that dynamic hooking has on performance, we measured both the processing time when the hooking is on and that when the hooking is off. We repeated each experiment five times and reported the average number. The result from the evaluation is shown in Table 2.

Table 2. Overhead measurement results

Situation	Average processing time for 207 samples
Not hooked	0.5s
Hooked	3.9s

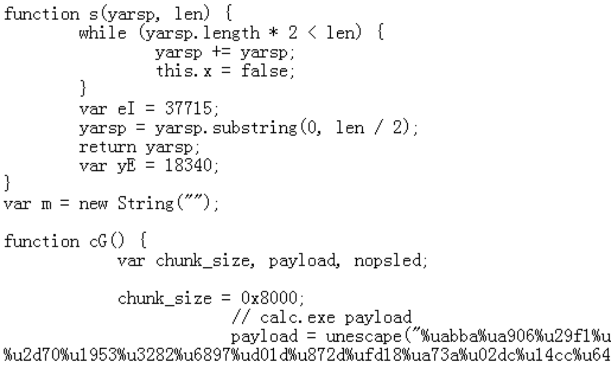
As we had expected the hooking of Adobe JavaScript engine has incurred significant overhead. However this overhead is comparable to other works that use static parsing instead of dynamic hooking. Given the superior extraction result that dynamic hooking can provide, MPScan strikes a balance between effectiveness and performance. And the analysis can be easily parallelized, which could further improve performance.

4.3. Application in Forensic Analysis

In the last part of this section, we examine MPScan's capability to assist forensic analysis of malicious PDF files. We take the challenge No.6 of the 2010 The Honeynet Project's Forensic Challenge[14] for example. In this challenge, contesters are asked to analyze a PDF document extracted from PCAP file. Some advanced tasks (worth more than 1 point) in the challenge are listed below:

1. Determine which object stream contains malicious content.
2. Find out which exploit is contained in the PDF file, and determine which one was actually triggered.
3. Locate the payload in the PDF file.

These tasks can be quite complicated if analyzed manually, but MPScan can handle it very well.



```
function s(yarsp, len) {
        while (yarsp.length * 2 < len) {
                yarsp += yarsp;
                this.x = false;
        }
        var eI = 37715;
        yarsp = yarsp.substring(0, len / 2);
        return yarsp;
        var yE = 18340;
}
var m = new String("");

function cG() {
            var chunk_size, payload, nopsled;

            chunk_size = 0x8000;
                        // calc.exe payload
                        payload = unescape("%uabba%ua906%u29f1%u
%u2d70%u1953%u3282%u6897%ud01d%u872d%ufd18%ua73a%u02dc%u14cc%u64
```

Figure 8. Part of deobfuscated JavaScript extracted from the PDF

MPScan's de-obfuscation module can correctly output the de-obfuscated JavaScript, from which forensic analyzer can gain insight of the exploitation. The exploitations and payloads are also detected by MPScan's multi-level detection module.

By reading the de-obfuscated JavaScript source code and the log of MPScan's detection module, analyzer can easily spot the vulnerable Adobe JavaScript APIs that have been triggered. And as shown in Figure 8, the payload of the exploitation is right in the JavaScript. By backtracking the flow of PDF object, forensic analyzer should be able to determine which PDF objects contain the malicious content. In this way, the three advanced tasks can all be effortlessly solved with the help of MPScan.

5. Conclusion and Future works

As PDF format becomes a major vector for malware propagation, effective tool that specifically tailored to de-obfuscate and detect malicious JavaScript embedded in PDF document has to be developed as a counter measure. We present MPScan, a dedicated PDF scanner that combined dynamic JavaScript source code de-obfuscation and extraction with static multilevel malware detection.

By hooking the Adobe Reader's native JavaScript engine, MPScan is robust against any kind of obfuscation including those that take advantage of the ambiguities and complexities of the PDF specification. Based on the accurate results provided by the JavaScript de-obfuscation module, the detection module will perform multilevel malware detection that covers a wide range of malicious PDF exploitation. The evaluation results have justified the effectiveness and high accuracy of MPScan. In addition, as we have demonstrated, MPScan can be well applied to assist forensic analysis.

For future work, we plan to add emulation functionality of user interaction to MPScan, so those JavaScript embedded PDF files that have to be triggered by user input can be automatically analyzed. Also we look forward to expanding the detection module of MPScan by adding yet another level of static detection, which is based on the AST node features in hope to expand detection coverage. Finally, we anticipate writing dummy functions for all deprecated Adobe JavaScript APIs, so more PDF documents can be correctly analyzed.

6. Acknowledgement

We thank Libo Chen and Yonggan Hou for providing highly valuable advices. We also thank anonymous reviewers for their comments. This work is partially supported by the National Natural Science Foundation Project (61003127), and the Huawei Company.

7. References

[1] Symantec. The Rise of PDF Malware http://www.symantec.com/connect/node/1473691 ,accessed June 2012

[2] M.Egele, P. Wurzinger, C. Kruegel, and E. Kirda.Defendingbrowsers against drive-by download:Mitigating heap-spraying code injection attacks. InProceedings of the 6th international conference onDetection of Intrusions and Malware, &VulnerabilityAssessment (DIMVA),2009

[3] http://searchsecurity.techtarget.com/definition/spear-phishing, accessed June 2012

[4] PavelLaskov and NedimŠrndić. 2011. Static detection of malicious JavaScript-bearing PDF documents. In *Proceedings of the 27th Annual Computer Security Applications Conference* (ACSAC '11). ACM, New York, NY, USA, 373-382

[5] Adobe Information Incorporated . PDF Reference, 6th edition. http://www.adobe.com/devnet/pdf/ pdf_reference.html, accessed June 2012

[6] W.-J. Li, S. Stolfo, A. Stavrou, E. Androulaki, andA. Keromytis. A study of malcode-bearing documents. InDetection of Intrusions and Malware & VulnerabilityAssessment (DIMVA), pages 231–250, 2007.

[7] MDN.SpiderMonkey. http://www.mozilla.org/js/spidermonkey/, accessed June 2012

[8] C. Willems, T. Holz, and F. Freiling. CWSandbox: Towardsautomated dynamic binary analysis. IEEE Security andPrivacy, 5(2):32–39, 2007

[9] PhoneyChttp://code.google.com/p/phoneyc/

[10] Z. Tzermias, G. Sykiotakis, M. Polychronakis, andE. Markatos. Combining static and dynamic analysis for thedetection of malicious documents. In European Workshop onSystem Security (EuroSec), 2011.

[11] P. Baecher and M. Koetter .libemu– X86 shellcode emulation . http://libemu.carnivore.it/, accessed June 2012

[12] Zhijie Chen, Chengyu Song, Xinhui Han, JianweiZhuge, Detecting Heap-spray in Drive-by Download Attacks Using Opcode Dynamic Instrumentation,In Proceedings of The 2nd Conference on Vulnerability Analysis and Risk Assessment(VARA'2009)

[13] Metasploit Framework http://metasploit.com/, accessed June 2012

[14] The Honeynet Project's Forensic Challenge http://www.honeynet.org/challenges/2010_6_malicious_pdf,accessed June 2012

[15] Z. Shafiq, S. Khayam, and M. Farooq. Embedded malware detection using markov n-grams. In *Detection of Intrusions and Malware* & *Vulnerability Assessment (DIMVA)*, pages 88–107, 2008.