Advanced Detection Tool for PDF Threats

Quentin Jerome, Samuel Marchal, Radu State, and Thomas Engel

SnT - University of Luxembourg
4 rue Alphonse Weicker
L-2721 Luxembourg, Luxembourg
{quentin.jerome,samuel.marchal,radu.state,thomas.engel}@uni.lu
http://wwwen.uni.lu/snt

Abstract. In this paper we introduce an efficient application for malicious PDF detection: ADEPT. With targeted attacks rising over the recent past, exploring a new detection and mitigation paradigm becomes mandatory. The use of malicious PDF files that exploit vulnerabilities in well-known PDF readers has become a popular vector for targeted attacks, for which few efficient approaches exist. Although simple in theory, parsing followed by analysis of such files is resource-intensive and may even be impossible due to several obfuscation and reader-specific artifacts. Our paper describes a new approach for detecting such malicious payloads that leverages machine learning techniques and an efficient feature selection mechanism for rapidly detecting anomalies. We assess our approach on a large selection of malicious files and report the experimental performance results for the developed prototype.

Keywords: PDF files, malware detection, machine learning

1 Introduction

Targeted attacks remain among the highly relevant persistent threat vectors. The past year has seen a dramatic rise in targeted attacks using PDF files as propagation vector[1][2]. Exploiting several zero-day vulnerabilities against popular readers (primarily from Adobe)[3], these attacks are difficult to mitigate for two main reasons. The first is related to users not perceiving the opening of PDF files as dangerous. Browsers plugins that automatically render PDF files make drive-by contamination even easier, since the user merely needs to visit a malicious site in order to get compromised. The second reason is the complex structure of PDF files, which makes their parsing quite challenging. Obfuscation techniques can thwart most of the available PDF parsing libraries, while still allowing error-tolerant readers to parse the file and thus compromise the system. Sometimes,

[1] http://thehackernews.com/2013/02/chinese-malware-campaign-beebus-target.html

[2] http://www.securelist.com/en/blog/774/A_Targeted_Attack_Against_The_Syrian_Ministry_of_Foreign_Affairs

[3] http://www.securelist.com/en/analysis/204792255/Kaspersky_Security_Bulletin_2012_The_overall_statistics_for_2012

even a slight change in a PDF file can make it unreadable to most libraries and still produce a working exploit for proprietary readers.

In this paper, we consider the mitigation of this attack, which has as major contributions, the followings:

- We propose an n-gram-based application to detect and mitigate attacks leveraging PDF vulnerabilities;
- This method does not rely on semantic parsing and thus is not prone to vulnerability exploitation found in PDF parsing libraries;
- We evaluate the performance of our system on a comprehensive dataset and report very good results for performance, speed and accuracy;
- We compare our tool with academic work;
- We provide a web service implementation of our approach.

Our paper is structured as follows: we start out in section 2 with an overview of malicious PDF files and highlight some of the recent vulnerabilities exploited by this threat. Section 3 details the overall architecture of our system. Section 4 presents the dataset used for the tuning of our approach. We describe the experiments performed and validation in section 5. In section 6, we compare our tool to previous work and we introduce our web service implementation. Section 7 discusses relationships with prior work and we conclude the paper and discuss future work in the section 8.

2 Malicious PDF

In this section, we introduce some lesser-known facts about the PDF language. We first present the basis of the PDF language. We next show some general ways used by rogue authors to craft malicious files. Finally we justify the challenges that our work must address by pointing out the analysis difficulties concerning PDF files.

2.1 PDF, a Programming Language

The PDF language is a PDL (Page Description Language). This type of language was created to avoid dependencies between documents and hardware. Thus, when someone wants to open a PDF document, it has only to own an application known as a Reader in order to interpret and understand the PDF language. This feature provides high portability because the resultant document is not hardware or OS dependant. This interesting property has made PDF an attractive alternative to platform-specific documents like Microsoft Word files. PDF files are now widely used on the Web and unfortunately have also become an attractive vector for malware propagation. Before showing how malicious PDF documents are crafted, we explain the basis of this uncommon language.

We can consider PDF as a collection of various types of object. According to the PDF reference [1], we can enumerate these different types:

- Boolean: Number (integer or real) and String values
- Names
- Array: collection of objects
- Dictionary: collection of objects indexed by their names (type Name)
- Streams: contain encoded data as text or images of the document
- Null objects

To be correctly interpreted and displayed, a PDF must contain some basic parts, ordered as follows.

1. A header, which contains the PDF language version number
2. The document body, which contains all objects
3. A cross-reference table containing offsets of all objects (whether currently in use or deleted by an incremental update) and version number of those objects
4. A trailer containing the cross-reference table offset.

The purpose of the cross-reference table is to retrieve objects efficiently. As mentioned above, it contains versions of objects which can be modified by updates. Once an object is deleted, its current version in the cross-reference table is modified to become the next generation number (version) of this object. Updated objects are appended to the end of the file with a corresponding cross-reference table modification.

Here we enumerate some existing ways to craft malicious documents. Indeed, by design PDF provides a large range of possibilities to create malicious documents. The most-abused features are, for instance, additional features such as JavaScript. Other features proper to a rich language can be used by rogue authors as well. Indeed, because of its popularity, PDF embeds an increasing number of features which offer new possibilities and flaws within the source code. We notice two large categories of attacks relying on PDF language: feature-based and exploit-based attacks. Feature-based attacks leverage only language features such as \OpenAction, which allows a task to be executed when someone opens the document. In [2] the authors illustrate this by crafting phishing attacks relying on such features. Exploit based attacks are more nasty since those rely on vulnerability exploitation. If such an attack succeed, the victim's machine becomes compromised and the attacker can control it remotely. This kind of attacks is even more critical for companies when a workstation becomes infected. We distinguish three families of attacks exploiting vulnerabilities in common PDF readers:

1. Attacks based only on JavaScript, which rely on a flaw in the JavaScript API and need JavaScript to be exploited;
2. Attacks relying on JavaScript only for payload delivery. For instance, before exploitation, a former step of heap spraying [3] can be performed. This technique aims at preparing the heap to control memory allocation and increase the success rate of jumping into the landing zone that the attacker wants;

3. Attacks that do not need JavaScript at all. For instance classical stack based buffer overflow flaws could directly allow arbitrary code execution without a previous heap preparation and thus do not need JavaScript.

Adobe Reader version(s)	Target	Flaws	CVE-ID
9.1, 8.1.4, 7.1.1 and earlier	JavaScript API getAnnots()	Resource Management Errors (CWE-399)	CVE-2009-1492
8.1.2 and earlier	JavaScript API util.printf()	Stack-based buffer overflow (CWE-119)	CVE-2008-2992
8.1.1 and earlier	JavaScript method in EScript.api	Code Injection (CWE-94)	CVE-2007-5663
10.1.1 and earlier on Windows and Mac OS, 9.x through 9.4.6 on UNIX	U3D	Unknown (probably heap overflow)	CVE-2011-2462
9.x through 9.1.2	authplay.dll	Code injection (CWE-94)	CVE-2009-1862
9.0 and earlier	J2BIG	Heap pointer corruption (CWE-119)	CVE-2009-0658
8.x before 8.3.1, 9.x before 9.4.6, 10.x before 10.1.1	CoolType.dll	Stack-based buffer overflow (CWE-119)	CVE-2011-2441
8.x before 8.2.5 and 9.x before 9.4	ActiveX	Input validation (CWE-20)	CVE-2010-2888
9.x before 9.3.2, and 8.x before 8.2.2	Unspecified	Buffer overflow (CWE-119)	CVE-2010-0198

Table 1. Vulnerabilities

In Table 1, we illustrate some real examples of the vulnerabilities cited previously. The information was gathered from the Metasploit[4] database and from the NIST National Vulnerability Database[5]. The CWE – Common Weakness Enumeration – references were retrieved from the MITRE[6] database. We can see in this table that several Reader versions and OSs can be targeted, increasing the attractiveness of such attacks.

2.2 Challenges

As shown in table 1, several Readers are vulnerable to exploitation. Hence it is very difficult to perform classical dynamic analysis. All the vulnerable software should be grouped in a common monitoring environment in order to cover the whole range of threats. Whence choosing a static detection approach is sound

[4] http://www.metasploit.com/
[5] http://nvd.nist.gov/
[6] http://cwe.mitre.org/data/slices/2000.html

in that particular case. However we must face other problems, specific to static analysis.

The PDF language offers its own obfuscation facilities like representing *Name* or *String* objects in hexadecimal form. In addition, another way to obfuscate PDF documents is to use cascade filters to encode Streams objects. In this fashion, the attacker is able to encode the same stream twice or even more with different algorithms. This technique is straightforward to use for an attacker and difficult to reverse engineer. For instance, according to the version 1.7 of the PDF documentation [1], the PDF language provides about 10 filters, including one to encrypt streams.

Object reference can be used to increase the obfuscation level as well. This allows calling an object which is defined elsewhere in the document. It is worth noting that the physical representation of the file does not matter, only the logical structure is important for the Reader.

Another technique often used in malicious files is JavaScript string obfuscation, which has been heavily abused in the past in web based attacks. For instance, using the *eval()* function on an encoded part of the script complicates static analysis.

These aforementioned obfuscation methods are massively used within malicious documents and makes the reverse engineer of files harder. A comprehensive study is performed in [4] and is a relevant example of analysis issues. To avoid time consuming and prone to error desobfuscation, we were motivated to use n-grams as features to represent documents. In addition, extracting n-grams does not require semantic parsing as done in previous works [5,6]. Since the Poppler library used in these approaches can be vulnerable[7], using it for detecting malicious documents is not relevant.

3 Tool Description and Architecture

The ADEPT architecture, presented in this section, is designed to provide both accuracy and performance. A well-known technique for automated detection is machine learning-based classification. This method leverages features associated with malware and benign files and uses them to build a model. The latter is used to classify files depending on their features. Figure 1 shows the architecture of the tool.

3.1 Feature Selection

The first step is to choose relevant features representing both regular and malicious PDF. We choose n-gram to achieve this. N-grams are substrings of length n extracted by sliding over the input character by character. Formally, a n-gram sequence denoted E by :

[7] `http://www.securityfocus.com/archive/1/526364/30/0/threaded`

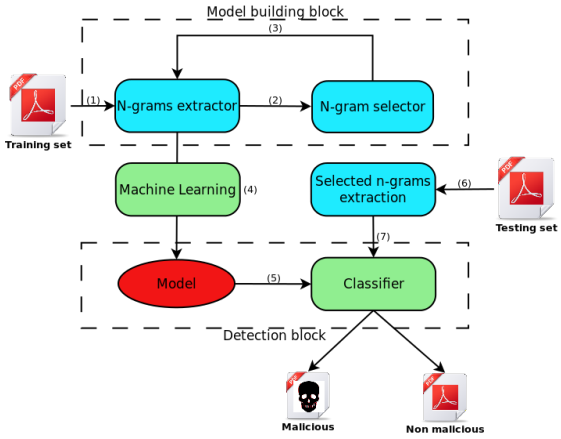

Fig. 1. System architecture

$\forall s \in S^*$, where S^* is the word set of a given alphabet S, $\forall n \in \mathbb{N}^*$, where n is the n-gram length :

$$E = \{s[i; i+n], i \in [0, L-n] \cap \mathbb{N}\}$$

where $s[i; j]$ denotes the substring extracted from s containing characters between indexes i and j. L denotes the length of the string s.

For example, in the string *malicious* and for $n = 3$ we extract the following n-grams set $\{'mal';' ali';' lic';' ici';' cio';' iou';' ous'\}$. The number of n-grams is $N = L - (n-1) \approx L$ and constitutes an upper bound to the number of distinct n-grams that we could extract from a given file.

3.2 Model Building Block

This module extracts relevant features to build the final model used for malware detection. This task is time consuming due to the intensive process of n-grams extraction. However, as the model is built only once, its processing time is not an issue.

N-grams Extractor. This entity parse PDF documents and extracts all n-grams from them. It also permits to gather n-grams for collection of documents.

N-grams Selector. The selector identifies the most relevant n-grams (or features) to include in the model. While, in theory, all features can be used, fast processing of PDF files is possible only when a subset of all possible n-grams is used. Thereby, only the most frequent n-grams are selected among our document corpus. During a preliminary study of our dataset, we noticed that malicious documents are shorter than benign files. To make our approach size independent,

we opted for a binary count of n-grams. Hence, each document is represented by a binary vector where each component stands for a n-gram, set to one if the n-gram appears in the document, zero otherwise.

Once features are selected, we need to retrieve the binary occurrence of these features in the initial dataset (step 3 in figure 1). In the end of step 3 we get a matrix where each line corresponds to a file and each column to one selected n-gram. This data is ready to train a machine learning algorithm in step 4. We discuss the learning algorithm embedded in our tool in section 5.2.

3.3 Detection Block

Model. This entity results from the machine learning algorithm which builds a model based on data gathered in the preceding block. This model is used now as a classifier input. In order to find the best classifier, we performed the experiments detailed in section 5.

Classifier. Classifier is strongly related to the learning algorithm because it takes a model previously learned as an input parameter for further comparison (step 5). At the same time, it takes a feature vector extracted from a file that we want to classify (step 7). The classifier determines if the feature vector extracted from a file fits a benign or a malicious profile, according to the model that has been learned previously.

4 Dataset Introduction

This section presents the preliminary study made on the dataset detailed in table 2. In order to compare our findings and benchmark our approach with respect to previous studies we use the datasets introduced in [5]: D1,D2 and D3. The dataset D4 is used further in this paper to evaluate our approach.

Our whole dataset was provided by VirusTotal[8]. For each dataset we have a set of detected files and a set of undetected ones. Files were labelled as detected by Virustotal if at least one anti-virus package among 42 reported an alert. In contrast, all files labelled as undetected passed through anti-virus packages without raising any alert.

While we analyzed some PDF documents manually – with the PDFTool[9] toolkit – we observed that many files had the same structure and almost the same size, but a different hash code. The physical and logical structure of these files were indeed very similar. We assumed that many malicious documents have been generated by exploit kit like BlackHole[10] or Metasploit[11]. Indeed, two hashing values calculated can be very different if only one byte differs between the two

[8] https://www.virustotal.com/

[9] http://blog.didierstevens.com/programs/pdf-tools/

[10] `http://en.wikipedia.org/wiki/Blackhole_exploit_kit`

[11] `http://www.metasploit.com/`

	D1		D2		D3		D4	
	det.	undet.	det.	undet.	det.	undet.	det.	undet.
Date of collection	2010-11-03		2011-01-19		2011-02-17		2012-12-21	
Number of files	7,592	7,768	6,465	9,993	11,634	22,490	3892	3474
Dataset size	873MB	13GB	429MB	13GB	1.5GB	29GB	223MB	3.5GB
Average file size	118KB	1.8MB	67KB	1.4MB	129KB	1.4MB	57KB	1MB
Number of different files	476	unknown	367	unknown	822	unknown	173	unknown

Table 2. Datasets introduction

files. However this high degree of similarity produces misleading results during the calibration phase of the tool. The reason is that if we have different files with small dissimilarities, the likelihood of taking into account these dissimilarities is low. As a consequence, if we want to evaluate our tool with common machine learning assessment techniques, we would probably test our tool on previously seen instances. Moreover, similarities do not contribute to model building. Therefore we assume that files are different on the PDFID output basis. PDFID is part of the PDFTool toolkit and reflects the internal structure of a given file.

5 Experiments

Here, we present the methodology we follow to find the best classification algorithm for our feature set. To achieve this, we used Weka [7], a well-known machine learning toolkit.

5.1 Experimental Description

In order to find out the best combination of n-gram/classifier, we ran a ten-fold cross-validation test on a labelled – benign or malign – document corpus. We chose to experiment with several well-known classification methods that range from tree and rule based classifiers to Support Vector Machine (SVM). To make an n-fold cross-validation, we firstly partition our dataset into n subsets. We then take $(n-1)$ subsets to train a machine learning algorithm and the remaining one for testing. In order to test each instance available in the dataset we do this n times. Ten folds are most frequently used to obtain significant results[8]. What we want is a classifier that gives the best prediction capabilities. To reach this goal, we must deal with the file similarity problem that we mentioned in section 4. To overcome this issue, we use only different files on the PDFID output basis. By doing this, we provide a *worst case scenario* that gives us a lower bound on detection capability for our tool.

Following we enumerate the settings used for those experiments:

- 843 malicious files gathered from datasets D1 and D2. This is the sum of different files in each dataset;
- 843 regular files randomly chosen in D1 and D2;

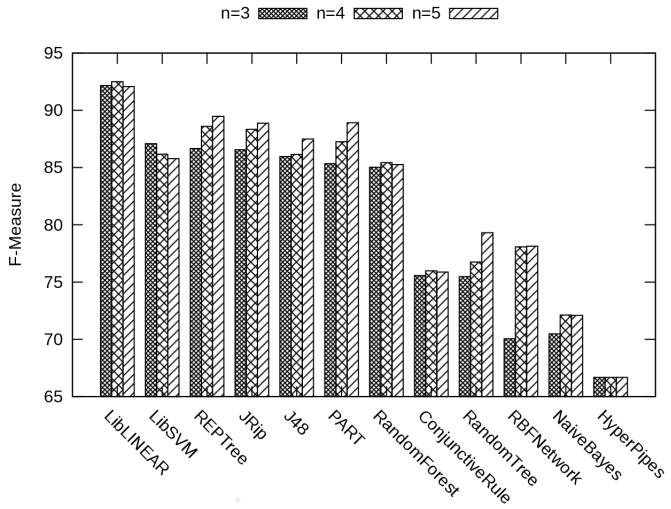

Fig. 2. Classification results for several classifiers

– We selected the 10,000 most frequent features in order to build the model. We determined that n-grams occurring less did not contribute to the model significantly.

We use a balanced dataset, as recommended by the machine-learning community [9] to avoid over-fitting and under-fitting issues.

Figure 2 depicts classification performances for each combination of n-gram/classifier that we experimented. On the y axis, we plot the F-Measure also known as F-1 score. The F-Measure is defined as follows:

$$\text{F-measure} = \frac{2(Recall \cdot Precision)}{Recall + Precision}$$

where Recall $= \frac{\text{Instances in class i classified as belonging to class i}}{\text{Instances in class i}}$

and Precision $= \frac{\text{Instances in class i classified as belonging to class i}}{\text{Instances classified as belonging to class i}}$

We used this metric to assess our tool since it evaluates both the retrieving capability of the tool through the *Recall* metric as well as the prediction capability through the *Precision*. Based on the results depicted in figure 2, we choose LibLINEAR as classification algorithm coupled with 4-grams to build our detection mechanism since this combination of feature bring the best results with F-Measure = 92.50%.

5.2 Classifiers Details

This section describes how LibLINEAR [10] works in details. It is an implementation of a support vector machine (SVM) classifier. The aim of SVM classification is to calculate the equation of the boundary between two sets of labelled

instances (PDF files in our case) characterized by n features. In this n dimensional problem, it must find the equation of an hyperplane. The shape of the hyperplane can be linear, polynomial, radial or sigmoidal and is determined by a kernel function. The kernel maps data into a space, in which it can be separated by an hyperplane. LibLINEAR is faster than LibSVM because it does not map instances into higher dimensional space, but instead it tries directly to separate instances in the initial vector space [10]. Thus, the computational complexity of the algorithm grows linearly with the number of instances.

More formally, by doing a LibLINEAR classification, the learning algorithm solves the following optimization problem:

$$\begin{gathered}\min_{w,b}\{\tfrac{1}{2} \parallel w \parallel^2 + C \sum_{i=1}^{n} \xi(w; x_i, y_i)\} \\ \text{subject to } y_i(w \cdot x_i - b) \geq 1 \ \forall \ 0 \leq i \leq n\end{gathered}$$

where y_i is the class of instance i
where x_i represents an instance i
and w is the normal to the hyperplane

The C parameter represents margin rigidity: the higher it is, the softer are margins. This means that we allow misclassified instances to contribute to the model. In contrast, when margins are more rigid, we do not allow those instances to be part of the model. We have to be careful in choosing this parameter because it can lead to under-fitting or over-fitting problems. While the former would represent more our training sample rather than the instance population, the latter would be too general. To find the good value for C, a grid search is usually performed. This consists in varying the parameter and doing a cross validation for each variation. Parameter offering the best cross-validation result is adopted. After a grid search we were able to determine that $C = 0.03125$ is the optimum value. The $\xi(w; x_i, y_i)$ term is the loss function, which approximates the misclassification degree of instance i.

6 Evaluation and Use-case

We present in this section a real-life use-case for such a detection tool. We firstly define what we mean by such a scenario and then we evaluate our tool. Lastly, we compare our approach with PJScan, another tool aiming at detecting malicious PDF files.

6.1 Real-life Use-case

Before going further into this evaluation, we propose to define what we mean by a real-life use-case. Following we point out two points of interest for running our experiments:

1. We must use a training set older than the files we want to detect;
2. We must assess our tool on a realistic dataset of malicious files.

In order to satisfy the first condition, we use the three older datasets introduced in section 4 to train the tool. As a result, we do the training with 24,327 files in both classes extracted from D1,D2 and D3. For the evaluation set, we use the most recent dataset, namely D4. In using these settings for the experiments, we also assess the viability of our approach regarding the threat evolution since the training set contains files two years older than files used for testing. Concerning the second requirement, we do not pay attention to the similarity problem between learning and testing. This makes sense since the tool must be able to find threats present in its knowledge base. Moreover, it is possible that some files in the testing set share similarities between them. To quantify the similarity between learning and testing, we found that 50 have the same hashing signature and 416 have a similar PDFID fingerprint. For this experiment we use the settings defined in the previous section.

	class. as mal.	class. as .reg	Recall	Precision	F-Measure
Malicious	3667	224	97.00%	96.85%	96.92%
Undetected	8	3466			

Table 3. Realistic scenario evaluation

We can see immediately in table 3 that the results are better than for the previous experiments. This can be partly explained by the fact that we used many more files than for our evaluation with Weka. Another reason is that in this test case, we did not filter the initial dataset as we did in our first experiment. This allows us to assess the real capabilities of the tool in terms of prediction and identifying known threats. We can also note the low false positive rate – benign files incorrectly classified as malicious – of 0.23% as well as a very good classification accuracy of 96.85%. This is a valuable attribute in a detection tool since only few false alarms are raised. Furthermore, we point out that files having the same hashing signature or similar PDFID fingerprint that files in the model were all well classified.

To have a better idea of which files remain undetected by the tool we extracted some pertinent information from misclassified files. The first point of interest is that among these 224 files there are 67 different PDFID outputs. We verified that no file with any of these outputs was in our training data. We can assume that these files are new attacks that our tool did not have in its training set. We also found that 399 files in our evaluation set had one of these PDFID signature. This means that we correctly identified 175 of these files while 224 were misclassified. If these 399 files are really totally new threat, relatively to the tool knowledge, we can not hope detecting new threats with 100% accuracy.

For sake of space, we do not present the detailed throughput assessment of this detection mechanism. However, it is worth noting that we can process around hundred regular documents per minute on a desktop computer with an Intel Core I5 processor and 8GB of RAM. We mean by *regular documents* documents that

we are used to deal with on a daily basis. Since malicious files are lightweight, the tool performs faster detection on these but a scenario containing only rogue files seems to be unlikely.

6.2 Comparisons

In this section we compare our approach with previous academic work in this area. Therefore, we compare our results against the detection capabilities of PJScan [5]. To make the comparison as fair as possible we use exactly the same scenario we defined in the previous section .

Comparison with Academic Work. Here we summarize the experiments that we ran in order to compare our tool to PJScan. We chose to compare our tool to PJScan because it is well documented and its source code is open[12]. We do not present a run-time performance comparison with PJScan because we ran it on a virtual machine due to compatibility issues. Before use, PJScan needs to be trained on a malicious training sample. As PJScan uses One Class SVM classification, the model has to be built using only one class. To satisfy this requirement we fed the model with the malicious files that we used in our training set. We expected very different results from our own because PJScan deals only with documents containing JavaScript.

	class. as mal.	class. as .reg	Recall	Precision	F-Measure
Malicious	209	49	88.89	68.67	77.48
Undetected	3	31			

Table 4. PJScan evaluation

Table 4 summarizes the results provided by PJScan. The tool was only able to process 3.9% of the test set. According to the output of the tool, the remaining files were skipped because no information was found in them. In this situation, the tool is unable to classify files and thus does not take any decision. However, in reality we need to take decisions regarding unknown files. Thus, we can conclude that this tool does not fit well with a real-life scenario. To be fair we compare both approaches according to the file that PJScan is able to process. From this comparison, we can conclude that our approach outperforms PJScan since it has a better accuracy, 96.85% for ADEPT against 82.19% for PJScan.

6.3 Web Service Implementation

We briefly present here a web interface implementation of our tool. We developed a front end that makes the tool more user-friendly than the command line

[12] `http://sourceforge.net/p/pjscan/home/Home/`

version. The service is hosted at `http://www.secan-lab.uni.lu/pdfchecker` and provides a VirusTotal-like graphical interface. The user can choose either to scan a file or try to retrieve a scan result for a previously scanned file by providing its SHA-256 signature. The result provides useful information to the user by showing the first and the last submission date. In addition, we also provide a PDFID-like output from another tool that we have developed. This tool actually extends the work done for PDFID and does it faster, up to 30 times for big files (>25.0MB). This tool also warns the user when the file he scanned contains dangerous features. This extra-feature is particularly useful when the tool misclassified a document.

7 Related Work

In this section we present the related work concerning PDF analysis and automated PDF detection. We also mention some relevant papers about the general topic of malware detection that drove us to build such a detection mechanism.

7.1 PDF Analysis

Several approaches exist for PDF analysis. We can find both static and dynamic, or even hybrid methods. While our tool addresses the problem of malicious file detection we cite here some analysis tools, helpful when we must deal with unknown threats. Outputs from analysis tools are often used as input for a detection mechanism.

PdfTools[13], developed by Didier Stevens, is an analysis toolkit consisting of PDFID and PDF-PARSER. The former gives statistics about potentially malicious features which are embedded in a PDF while the second is a parser that displays the PDF code in a readable format. Another static tool, PDF Structazer, is presented in [2]. It can be used to analyse, create or modify PDF files. In the same paper, the authors show the power of the PDF language by implementing phishing attacks using only language features. Itext[14] is an open source and free library providing ways of automating PDF creation and modification. Almost all features provided by the PDF language are supported; [11] provides an introduction and practical guide.

A dynamic approach is implemented in CWSandbox[12], an application which monitors malware execution in a sandboxed environment. Its dynamic analysis monitors features such as file modification, changes made to the Windows registry and processes created. Post execution, the application provides a detailed report directly readable by analysts who can take a decision concerning the file. This tool has been adapted for malicious PDF[15], which is certainly its main problem concerning PDF detection. However, as noted previously, some exploits run only on particular Adobe Reader version. Thus before running a malicious

[13] http://blog.didierstevens.com/programs/pdf-tools/
[14] http://itextpdf.com/itext.php
[15] http://honeyblog.org/archives/12-Analyzing-Malicious-PDF-Files.html

file in a sandboxed environment, we need to know which version is targeted. In [13] the authors introduce MIST, means of interpreting output from online platforms such as CWSandbox. The resulting instruction can subsequently be used for a machine learning based classification.

7.2 Malicious PDF Detection

A combination of both types of analysis is implemented in MDScan [14] in order to detect malicious PDF files. MDScan first detects malicious code by parsing the document. The extracted code is then monitored in an emulator emulating providing a subset of the functionalities available in the Adobe API. Because some API functions have not yet been implemented; the detection can be defeated if malicious file exploit an unimplemented function. Schmitt *et al.* present PDF Scrutinizer in [15]. The approach combines both static and dynamic analysis to detect malicious PDF files containing JavaScript.

In [5], *Laskov et al.* describe PJScan, a static detection based on machine learning. They focus on malicious PDFs containing JavaScript. They use a JavaScript extractor and then treat extracted code to transform it in a standard token representation. A learning algorithm is then applied to the tokenized sequence in order to detect malicious patterns. The authors leveraged approaches introduced in [16], where lexical analysis associated with a learning method is applied to detect drive-by download attacks.

A recent tool is proposed in [17] where the authors leverage several metadata extraction combined with machine learning in order to determine whether a file is likely to be malicious or not. Another approach, based on the hierarchical structure of PDF documents combined with machine learning has been presented in [6]. Although this approach seems to have good performances, it is still vulnerable to parser vulnerabilities since it uses the libpoppler library.

7.3 Malware Analysis

We have done previous work in machine learning techniques for security in [18,19,20], but focussed more on the network traffic monitoring and not the system level defines.

In [21] the authors present a detection solution that, like ours, is based on n-gram associated with learning techniques. Their experiments tested different classifiers and different values of n. They address malicious Windows binary (PE files) detection. While their approach targets malicious code in binary format, ours approach deals with ASCII encoded files. Additionally, while their approach leverages information gain in feature selection, ours uses most frequent features.

N-gram analysis is used in [22] in order to detect file types. This type of analysis can be used to tag unknown files or to detect files which try to disquise their content. To reach their goal, the authors firstly obtain n-gram distributions for various file types. Secondly, they compare n-gram distribution of a file under test with known values to determine its real type. Similar work appears in [23], where the authors present a way of detecting embedded files within documents.

This method consists in observing variation of n-gram distribution compared to the expected distribution for a given file format. This method can also be used to detect embedded files within PDF files.

Wei-Jen Li et al. [24] present a way to analyse malicious Word documents. Their method is based on static analysis coupled with a dynamic element. Byte distribution is analysed in an initial static analysis step. They further monitor malware behaviour using API hooking techniques. The file is ultimately classified as a result of these two steps.

In term of full dynamic analysis, TTAnalyze, presented in [25] aims to quickly identify malicious PE files. To achieve this, the tool monitors both the Windows API and native API hooks to catch even the stealthiest malware. It evades detection by the rogue program in avoiding both classic API hooking and breakpoint setting.

8 Conclusion

This paper describes an accurate detection tool for malicious payload detection. Our work was motivated by the lack of efficient approaches to mitigate an advanced persistent threat that has had significant impact recently. We proposed a method that leverages machine learning and sequence based features in order to detect malicious PDF files. We have assessed our approach on a very large set of data that was obtained through the courtesy of VirusTotal. The performance in both speed and accuracy are very good since it is able to process hundred files per minutes with 0.23% of false positives. We plan to extend this work by integrating additional pieces of information, such as entropy and multiple alignment scores. We are also considering to generalize this approach to a larger class of payload types, but obtaining ground truth datasets for each is a particularly challenging.

Acknowledgment

The authors would like to thank Prof. Dr. Pavel Laskov for the support and dataset provided for our experiments. Special thanks also go to the VirusTotal team for giving us access to several datasets.

References

1. Adobe: PDF reference sixth edition, adobe portable document format, version 1.7 (2006)
2. Filiol, E., Blonce, A., Frayssignes, L.: Portable document format (PDF) security analysis and malware threats. Journal in Computer Virology (2007) 75–86
3. Daniel, M., Honoroff, J., Miller, C.: Engineering heap overflow exploits with JavaScript. In: Proceedings of the 2nd conference on USENIX Workshop on offensive technologies. WOOT'08, Berkeley, CA, USA, USENIX Association (2008) 1:1–1:6

4. Rahman, M.A.: Getting owned by malicious PDF - analysis. Global Information Assurance Certification Paper (2010)
5. Laskov, P., Šrndić, N.: Static detection of malicious JavaScript-bearing PDF documents. In: Proceedings of the 27th Annual Computer Security Applications Conference. ACSAC '11, New York, NY, USA, ACM (2011) 373–382
6. Šrndic, N., Laskov, P.: Detection of malicious pdf files based on hierarchical document structure. In: Proceedings of the 20th Annual Network & Distributed System Security Symposium. (2013)
7. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.: The WEKA data mining software: an update. ACM SIGKDD Explorations Newsletter (1) (2009) 10–18
8. Witten, I., Frank, E., Hall, M.: Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann (2011)
9. Akbani, R., Kwek, S., Japkowicz, N.: Applying support vector machines to imbalanced datasets. Machine Learning: ECML 2004 (2004) 39–50
10. Fan, R., Chang, K., Hsieh, C., Wang, X., Lin, C.: Liblinear: A library for large linear classification. The Journal of Machine Learning Research (2008) 1871–1874
11. Lowagie, B.: iText in action: Creating and manipulating PDF. Dreamtech Press (2006)
12. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using CWSandbox. IEEE Security & Privacy (2007) 32–39
13. Trinius, P., Willems, C., Holz, T., Rieck, K.: A malware instruction set for behavior-based analysis. Proc of Conference Sicherheit Schutz und Zuverlssigkeit SICHERHEIT (TR-2009-07) (2011) 1–11
14. Tzermias, Z., Sykiotakis, G., Polychronakis, M., Markatos, E.P.: Combining static and dynamic analysis for the detection of malicious documents. In: Proceedings of the Fourth European Workshop on System Security. EUROSEC '11, New York, NY, USA, ACM (2011) 4:1–4:6
15. Schmitt, F., Gassen, J., Gerhards-Padilla, E.: Pdf scrutinizer: Detecting javascript-based attacks in pdf documents. In: Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on, IEEE (2012) 104–111
16. Rieck, K., Krueger, T., Dewald, A.: Cujo: Efficient detection and prevention of drive-by-download attacks. In: Proceedings of the 26th Annual Computer Security Applications Conference, ACM (2010) 31–39
17. Smutz, C., Stavrou, A.: Malicious PDF detection using metadata and structural features. In: Proceedings of the 28th Annual Computer Security Applications Conference, ACM (2012) 239–248
18. François, J., Wang, S., State, R., Engel, T.: Bottrack: tracking botnets using netflow and pagerank. In: NETWORKING 2011. Springer Berlin Heidelberg (2011) 1–14
19. Wagner, C., Wagener, G., State, R., Engel, T.: Malware analysis with graph kernels and support vector machines. In: Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on, IEEE (2009) 63–68
20. Abdelnur, H.J., State, R., Festor, O.: Advanced network fingerprinting. In: Recent Advances in Intrusion Detection. Springer Berlin Heidelberg (2008) 372–389
21. Kolter, J., Maloof, M.: Learning to detect and classify malicious executables in the wild. The Journal of Machine Learning Research (2006) 2721–2744
22. Li, W., Wang, K., Stolfo, S., Herzog, B.: Fileprints: Identifying file types by n-gram analysis. In: Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC, Ieee (2005) 64–71

23. Stolfo, S.J., Wang, K., Li, W.J.: Fileprint analysis for malware detection. ACM CCS WORM (2005)
24. Li, W., Stolfo, S., Stavrou, A., Androulaki, E., Keromytis, A.: A study of malcode-bearing documents. Detection of Intrusions and Malware, and Vulnerability Assessment (2007) 231–250
25. Bayer, U., Moser, A., Kruegel, C., Kirda, E.: Dynamic analysis of malicious code. Journal in Computer Virology (1) (2006) 67–77