University of Crete
Computer Science Department

Combining Static and Dynamic Analysis for the Detection of Malicious Documents

Zacharias Tzermias

Master's Thesis

October 2011
Heraklion, Greece

University of Crete
Computer Science Department

Combining Static and Dynamic Analysis for the Detection of Malicious Documents

Thesis submitted by

Zacharias Tzermias

in partial fulfilment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Zacharias Tzermias

Committee approvals: _____
Evangelos P. Markatos
Professor, Thesis Supervisor

_____
Sotiris Ioannidis
Principal Researcher at FORTH-ICS

_____
Maria Papadopouli
Assistant Professor

Departmental approval: _____
Angelos Bilas
Associate Professor, Chairman of Graduate Studies

Heraklion, October 2011

Abstract

The widespread adoption of the PDF format for document exchange has given rise to the use of PDF files as a prime vector for malware propagation. Cybercriminals can use a specially crafted document as well as social engineering tacticts to target specific victims, like large enterprises or military related organizations. Targeted attacks using this type of documents have been increased in the last years. As vulnerabilities in the major PDF viewers keep surfacing, effective detection of malicious PDF documents remains an important issue. In this thesis we present MDScan, a standalone malicious document scanner that combines static document analysis and dynamic code execution to detect previously unknown PDF threats. Our evaluation shows that MDScan can detect a broad range of malicious PDF documents with a small runtime overhead, even when they have been extensively obfuscated. We also state that despite antivirus software making efforts towards malicious PDF detection, they still cannot effectively detect unknown PDF threats. Moreover, we seek for the presence of malicious documents, on both a popular social network and on large spam mail corpuses.

Supervisor: Professor Evangelos Markatos

Περίληψη

Η διαδεδομένη υιοθέτηση των αρχείων PDF για την ανταλλαγή εγγράφων, έχει αυξήσει την χρήση τους ως πρωτεύων μέσο για την διάδοση κακόβουλων αρχείων (malware). Κακόβουλοι χρήστες μπορούν να χρησιμοποιήσουν ειδικά κατασκευασμένα έγγραφα καθώς και τεχνικές social engineering για να επιτεθούν σε συκγκεριμένους στόχους, όπως μεγάλες επιχειρήσεις ή οργανισμούς στρατιωτικού περιεχομένου. Καθώς τρωτά σημεία στις πιο διάσημες εφαρμογές προβολής εγγράφων PDF ανακαλύπτονται ολοένα και περισσότερο, η αποτελεσματική ανίχνευση των κακόβουλων εγγραφων PDF παραμένει ένα σημαντικό ζήτημα.

Σε αυτή την εργασία παρουσιάζουμε το MDScan, μια αυτόνομη εφαρμογή σάρωσης κακόβουλων εγγράφων που συνδυάζει την στατική ανάλυση του εγγράφου καθώς και την δυναμική εκτέλεση κώδικα που περιέχεται σε αυτό, για να εντοπίσει άγνωστες απειλές από κακόβουλα έγγραφα. Η πειραματική αξιολόγηση μας, δείχνει ότι το MDScan μπορεί να εντοπίσει μια μεγάλη γκάμα από κακόβουλα έγγραφα PDF με μικρό κόστος χρόνου, ακόμα και όταν σε αυτά έχουν εφαρμόστει εκτεταμένες τεχνικές απόκρυψης της επίθεσης. Επίσης, ερευνούμε την παρουσία κακόβουλων εγγράφων σε ένα δημιφιλές κοινωνικό δίκτυο καθώς και σε μεγάλες συλλογές από μηνύματα spam.

Επόπτης: Καθηγητής Ευάγγελος Μαρκατος

Acknowledgments

I am deeply grateful to my supervisor Prof. Evangelos Markatos, for his valuable advice and guidance during my studies. Many thanks to Dr. Sotiris Ioannidis for his advice and support. I am also grateful to Michalis Polychronakis for his constant support and encouragement as long as his advice, playing a crucial role to my studies. A big thanks to my friend and colleague George Sykiotakis who contributed at the implementation of the very first version of MDScan. I would also like to thank Marco Cova for providing a collection of malicious documents during the evaluation stage of MDScan.

Many thanks to all former and present members of the Distributed Computing Systems Lab at FORTH-ICS Antonis Papadogiannakis, Spiros Ligouras, Giorgos Kontaxis, Eleni Gessiou, Nikos Tsikoudis, Lazaros Koromilas, Spiros Antonatos, Elias Athanasopoulos, Giorgos Vasiliadis, Iasonas Polakis, Demetris Antoniades, Apostolis Zarras, Giorgos Chinis, Harris Papadakis, Antonis Papaioannou, Giorgos Saloustros, Antonis Krithinakis, Christos Papachristos, Manolis Stamatogiannakis that contributed for a pleasant and productive environment all these years in the lab.

A sincere thank you to my friends Tzortzina Velegraki, Thanasis Petsas, Manolis Stylianakakis, Nikos Dritsos, Petros Tsakoumakopoulos, Kostas Kefalakis, Pavlos Efthymiou, Stratos Kyriakakis, Michalis Tsiknakis and many others that I do not mention by name, for their constant encouragement and support since the early days of my Master studies.

I would never made it through without the encouragement of my family. I would especially thank my mother and my brother, for their support, endless encouragement and patience throughout these years. Also, I owe a big thank you to my father who brought me into contact with the world of computers, at a very early age.

It ain't just the power. It's all about balance.
Keiichi Tsuchiya

Στον πατέρα μου

To my father

Contents

1 Introduction **1**
- 1.1 From malicious executables to malicious documents 1
- 1.2 Contributions 2
- 1.3 Thesis Outline 3

2 Background **5**
- 2.1 Portable Document Format 5
- 2.2 Targeted attacks 7

3 Design and Implementation **9**
- 3.1 Document Analysis 9
  - 3.1.1 File Parsing 10
  - 3.1.2 Emulation of the JavaScript for Acrobat API 12
  - 3.1.3 JavaScript Code Extraction 13
- 3.2 Code Execution and Shellcode Detection 14

4 Experimental Evaluation **17**
- 4.1 Data Set 17
- 4.2 Detection Effectiveness 17
- 4.3 Runtime Performance 20

5 Case Studies **23**
- 5.1 Case Study: Twitter 23
  - 5.1.1 Design and Implementation 24
  - 5.1.2 Results 25
  - 5.1.3 Proof of Concept attack 25
- 5.2 Case Study: Spam Corpus 25

6 Limitations **27**

7 Related Work **29**

8 Conclusion **31**

List of Figures

2.1 Structure of a PDF document. 6
2.2 Sample targeted attack message. 7

3.1 Overall architecture of MDScan. 10
3.2 A malformed (missing cross-reference table, an `endobj` keyword, and `%%EOF`) PDF document that is rendered normally by Adobe Reader. 11
3.3 An obfuscated version of the document shown in Figure 3.2 that is still rendered normally by Adobe Reader. Note that a part of the original JavaScript code has been stored into a non-code object, and that no PDF filters are used. 12

4.1 Cumulative fraction of the virus scanners of VirusTotal that detected a set of 11302 malicious PDF samples. 18
4.2 Cumulative fraction of virus scanners of VirusTotal that detected a set of 197 malicious PDF files at February and September 2011. . 19
4.3 Percentage of virus scanners of VirusTotal that detected obfuscated versions of malicious PDF files generated using Metasploit at February 2011. 20
4.4 Percentage of virus scanners of VirusTotal, that detected obfuscated versions of malicious PDF files generated using Metasploit at September 2011. 20
4.5 Evolution of detection rate for sample 9, at VirusTotal 21
4.6 Cumulative distribution of the processing time for malicious and benign PDF samples. 22
4.7 Average processing time for malicious and benign samples. 22

5.1 Architecture of the infrastructure used to download URLs from Twitter 24

1 Introduction

1.1 From malicious executables to malicious documents

The Portable Document Format (PDF) is one of the most popular file formats for document exchange. Created by Adobe Systems [3] in 1993 as an independent document representation format. Over the years, PDF has become extremely popular and is now used by a wide variety of environments—from households, schools to large enterprises—as the primary document exchange format.

From July 3, 2000, PDF specification was available by Adobe, and 2008 it was released as an open standard [8], offering the opportunity to software companies to develop tools and viewers that can manipulate this document format. Moreover, at PDF version 1.3, Adobe introduced JavaScript actions helping document authors to enrich their documents with interactive features such as form validation, attachment of multimedia contents etc. Due to several vulnerabilities discovered on PDF viewers as long as on the specification itself, PDF also attracted malware authors.

As the focus of attackers has recently shifted from server-side to client-side attacks, which target client applications like web browsers, media players, and document viewers [41], the universal adoption of the PDF format has rendered PDF documents a prime vector for malware distribution [47]. A key aspect of this increased attractiveness of the PDF format from the side of the attackers is the complexity of the feature-rich Adobe Reader for Windows—probably the most widely used PDF viewer—which has led to the discovery of many exploitable vulnerabilities. In many cases, other PDF viewers also suffer from the same or similar weaknesses.

In contrast to drive-by download attacks [27], besides being served by rogue web sites, malicious PDF files can also be distributed through other means, such as file-sharing networks, removable media, or as attachments to email messages. The

latter method has lately been particularly effective in combination with some social engineering tactics for targeted attacks against individual organizations. In these attacks, potential victims receive fake messages seemingly coming from a well respected colleague, which carry a legitimately looking but malicious PDF document. Due to the widespread use of PDF documents in corporate environments, PDF files are rarely blocked or face other restrictions even under strict security policies. This gives a great opportunity to attackers to use PDF document as a means of entering into corporate environments and even worse to military related targets. [32–34] According to F-secure, targeted attacks that leverage PDF-based exploits as a means of infection have been increased the last three years. [20, 22].

With the rapid evolvement of smartphones, allowing the rendering PDF documents, the scope of potential victims is constantly increasing. Flaws discovered to Apple iOS while rendering PDF documents, led to the creation of specially crafted PDF documents that can jailbreak an iOS-running device remotely. [21, 23, 36, 37] These documents are freely available. [1] Vulnerabilities in such devices, could attract attackers on compromising them, with minimum effort.

In essence, malicious PDF documents can be thought of as the rebirth of the macro viruses that plagued Microsoft Office and other productivity suites from the mid-1990s to the early 2000s [38]. One of the factors that led to the extinction of macro viruses was the additional security measures and protections that were gradually being applied to newer versions of the affected applications. Similarly, Reader X, the most recent version of Adobe Reader, comes with security features such as sandboxing and isolation, which significantly reduce the risk of full system compromise.

However, until the current vast user base of older versions of the most popular PDF viewers diminishes significantly, the effective detection of existing PDF threats will remain an important issue. For example, as we demonstrate, antivirus applications do not provide adequate detection coverage even for well-known PDF threats. while the use of simple obfuscation techniques can decrease the detection rate even further.

1.2 Contributions

In the above section we mentioned the growth of malicious PDF files and the need for an effective detection of this type of files.

In this thesis we are targeting to the detection of malicious PDF documents by developing a standalone detector, that combines both static analysis of the document and dynamic analysis of the embedded code. Key contributions in this thesis are the following:

- We present the design and implementation of MDScan, a standalone malicious document scanner that analyzes individual PDF documents and detects any embedded malicious code. Through the combination of static analysis of the document format representation, and dynamic analysis of the embedded

script code, MDScan can detect PDF documents that exploit even previously unknown vulnerabilities in PDF viewers. The autonomous design of MDScan allows it to be easily incorporated as a detection component into existing defenses, such as intrusion detection systems and antivirus applications.

- We test MDScan against real and generated malicious documents, as well as benign PDF files, and show that MDScan can accurately detect a broad range of malicious documents, even when they have been highly obfuscated, with a reasonable runtime processing overhead.

- We perform a case study at a popular social network and at spam mail corpuses, to seek for the presence of malicious activity via specially crafted malicious documents.

1.3 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 makes an overview of the PDF document format and the features supported. In Chapter 3 we describe in detail the design and the implementation of the MDScan. Chapter 4 presents the experimental evaluation of MDScan in terms of detection effectiveness and runtime performance. Chapter 5 presents case studies for the detection of malicious documents, in a popular social-network and in large spam mail corpuses respectively. We discuss limitations in Chapter 6. Finally, Chapter 7 summarizes related work, and Chapter 8 concludes the thesis.

2
Background

2.1 Portable Document Format

PDF, created by Adobe Systems, has become the de facto file format for the distribution of printable documents. According to its authors, PDF is a file format for document representation that is independent of the application, software, hardware and operating system used to create or open them.

As shown in Figure 2.1, a file adhering to the PDF specification has four main sections:

- A one-line header with the version number of the PDF specification.
- The body of the document, which consists of PDF objects which they define different elements of the document such as text, images, fonts, annotations, or even other embedded files.
- A cross-reference table with the offsets of the PDF objects within the file to allow random accesses to them.
- A trailer for quick access to the cross-reference table and other special objects.

The fundamental building block in PDF documents, is the PDF object. A PDF object can describe the appearance of page, define an element (e.g. annotations, text, images, fonts), or even contain an embedded file. Usually, a PDF document can be seen as a hierarchical collection of objects that are connected together, containing the required information.

Besides static data, PDF objects can also contain code written in JavaScript. This allows document authors to incorporate sophisticated features such as form

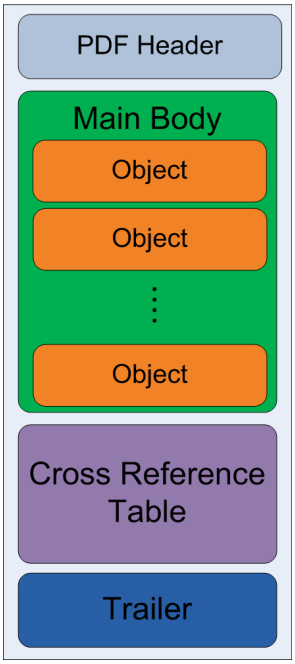

FIGURE 2.1: Structure of a PDF document.

validation, multimedia content, or even communication with external systems and applications. Moreover, Adobe provides a JavaScript API that allows document authors to create documents with additional functionality. The JavaScript API can be seen as a DOM-like representation of the whole document in JavaScript address space. Using this API and its methods, document authors can access various components of the document like page contents, metadata etc.

Unfortunately, attackers can also take advantage of the versatility offered by both JavaScript and the Adobe JavaScript API for the exploitation of arbitrary code execution vulnerabilities in the PDF viewer application. Through JavaScript, the attacker can achieve two crucial goals: trigger the vulnerable code, and divert the execution to code of his choice. Depending on the vulnerability, the first is achieved by calling the vulnerable API function or otherwise setting up the necessary conditions. Then, through heap-spraying [27] or other memory manipulation techniques, the flow of control is transferred to the embedded shellcode, which carries out the final step of the attack, e.g., dumping on disk and then launching an embedded malware executable.

Besides exploiting some vulnerability in the PDF viewer, attackers have exploited advanced PDF features such as the `/Launch` option, which automatically launches an embedded executable, or the `/URI` and `/GoTo` options [28], which can open external resources from the same host or the Internet. Although in both cases the application first asks for user authorization, such features are quite haz-

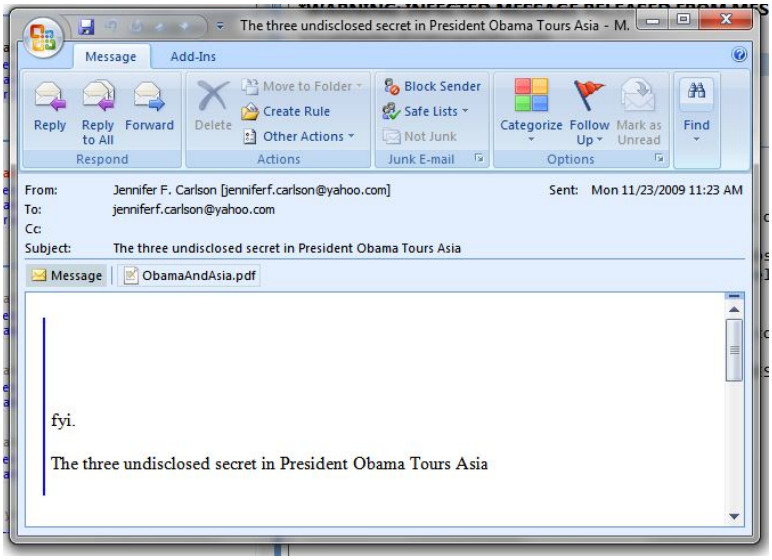

FIGURE 2.2: Sample targeted attack message.

ardous [52], and after the public exposure of their security implications they were promptly mitigated.

2.2 Targeted attacks

Nowadays, cybercriminals shift their focus from massive attack campaigns to target specific users [35]. In contrast to widespread attack campaigns, targeted attacks focus on a limited amount of victims. Tied with social engineering tactics, an e-mail is sent, from the address of a friend or co-worker of the victim (which is spoofed), trying to convince the victim to open a document.A real sample of a targeted attack e-mail is shown in Figure 2.2. Through social engineering tactics the user is convinced that the mail is not counterfeit, since sensitive information about the victim (e.g information about a project, on his current work) are referred, thus is mislead. As soon as the attachment is opened, it exploits a known vulnerability to the document reader and attacker's code is executed.

Targeted attacks, are a serious threat, because the attacker points to specific users, and tries to extort sensitive information. This would include classified documents from corporate or military environments etc. In comparison to widespread attacks where the attack would eventually trigger a honeypot or any other detection mechanism deployed by antivirus and antispam companies, targeted attacks could not trigger such mechanisms as they are destined to a limited amount of victims.

Attachments in this type of attacks, are usually files that are handled by productivity suites like Microsoft Office or Adobe Acrobat. Users usually have the false belief that these type of documents contain only static text and no executable code, thus could not potentially harm its system. Thus they deceived in opening

the attachment. F-secure states that the most common file type used in such types of attacks is PDF documents. [20] The acceptance of PDF documents in corporate environments and the ever-growing list of vulnerabilities in Adobe Acrobat Reader has led cybercriminals to abandon Microsoft Office document formats in favor of PDF format.

3
Design and Implementation

In this chapter we analyze how MDScan is constructed in order to detect malicious documents.

The mere presence of JavaScript code in a PDF file is not an indication of malicious intent, even if the code has been highly obfuscated. [40] Besides hindering malicious code analysis, code obfuscation is legitimately used for preventing reverse engineering of proprietary applications. To be resilient against highly obfuscated code, MDScan analyzes any embedded code by actually running it on a JavaScript interpreter. During execution, if some form of shellcode is revealed in the address space of the JavaScript interpreter, then the input document is flagged as malicious.

Document scanning in MDScan consists mainly of two phases. In the first phase, MDScan analyzes the input file and reconstructs the logical structure of the document by extracting all identified objects, including objects that contain JavaScript code. In the second phase, any JavaScript code found in the document is executed on an instrumented JavaScript interpreter, which at runtime can detect the presence of embedded shellcode. The overall design of MDScan is presented in Figure 3.1. In the following, we describe its main components and the details of our detection method.

3.1 Document Analysis

Upon reading the input document, MDScan analyzes its structure and extracts all identified objects, which are then organized in a hierarchical structure. The complexity and ambiguities [47,54,56] of the PDF specification [11] make this process a non-trivial task. In addition, most PDF viewers, including Adobe Reader, attempt

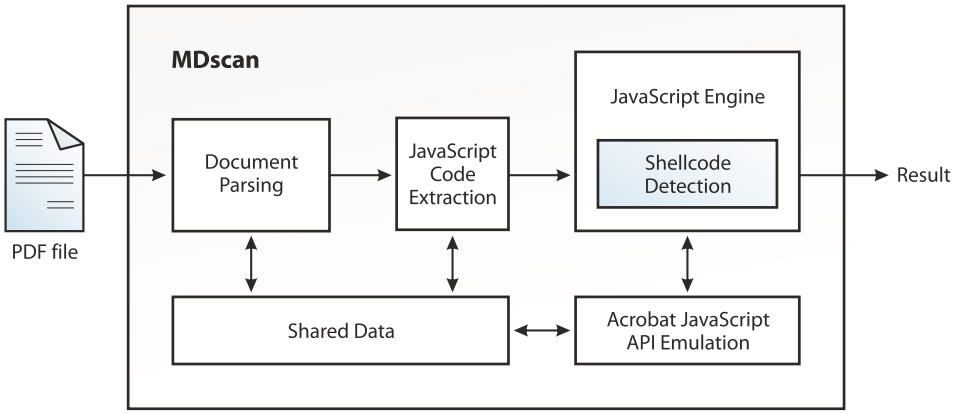

FIGURE 3.1: Overall architecture of MDScan.

to render even malformed documents, and generally do not strictly follow the PDF specification. This gives attackers even more room to hinder document analysis, by taking advantage of these intricacies to obfuscate the structure of malicious PDF files.

3.1.1 File Parsing

File parsing begins with the extraction of all objects found in the body of the document, including objects that have been deliberately left out from the cross-reference table. In fact, the cross-reference table can be omitted altogether, as in the document shown in Figure 3.2, along with other required (according to the PDF specification) elements, such as the `endobj`, `endstream`, and `%%EOF` keywords. Similarly, although according to the specification a PDF file should begin with the 5-byte sequence `%PDF-` followed by a version number, in practice Adobe Reader will open any document that contains this sequence anywhere within the first 1024 bytes of the file [56], even without a proper version number (line 1 in Figure 3.3). file [53, 56], even without a proper version number. Attackers can also use seemingly incorrect but actually valid keywords, such as `objend` instead of `endobj`. In general, the parser is resilient on parsing errors, and attempts to extract as much information as possible in a best-effort manner, in accordance with the behavior of the most popular PDF viewers.

After all objects have been identified, the parser proceeds to a normalization step that neutralizes any further obfuscations, and extracts semantic information about each identified object. Probably the most common object-level obfuscation technique is the use of filters to transform the stream data of an object and conceal the embedded JavaScript code. Initially, filters were introduced to allow the compression of large portions of data to improve document portability.

The PDF format supports many different filters for the decompression of arbitrary data (`/FlateDecode`, `/LZWDecode`, `/RunLengthDecode`), the de-

```
1  %PDF-1.1
2  1 0 obj <<
3    /Type /Catalog
4    /Pages 1 0 R
5    /OpenAction <<
6      /S /JavaScript
7      /JS (app.alert({cMsg: 'Hello!'});)
8    >>
9  >>
10 endobj
11
12 2 0 obj <<
13   /Title (Malicious Document)
14 >>
15
16 trailer <<
17   /Root 1 0 R
18   /Info 2 0 R
19 >>
```

FIGURE 3.2: A malformed (missing cross-reference table, an `endobj` keyword, and `%%EOF`) PDF document that is rendered normally by Adobe Reader.

compression of images (`/JBIG2Decode`, `/CCITTFaxDecode`, `/DCTDecode`, `/JPXDecode`), or the decoding of arbitrary 8-bit data that have been encoded as ASCII text (`/ASCIIHexDecode`, `/ASCII85Decode`).

PDF specification allows the use of more than one filters for data compression. For instance, a typical use of these filters is to compress an image using JBIG2 compression, and then encode the compressed data using an ASCII hexadecimal representation. In practice, attackers can combine any number of filters to conceal the embedded malicious JavaScript code. [51] Special care is taken for the correct handling of filter abbreviations [50], such as the use of `/Fl` in place of `/FlateDecode`. Although it is straightforward to extract the encoded data by undoing each transformation, stream compression is very effective against simple detection methods such as pattern matching.

Another important aspect of the object normalization step deals with keywords that have been encoded using an ASCII hexadecimal representation. The PDF format allows the arbitrary use of hexadecimal numbers in place of ASCII characters in keywords, as shown in Figure 3.3 enabling constructs like `/F#6CateDe#63od#65` being treated as equivalent to `/FlateDecode`. Similarly, strings in objects can be represented using various other encodings, such as octal or hexadecimal representations with flexible character whitespace requirements [54]. Finally, version 1.5 of the PDF specification introduced the concept of *object streams*, which contain a sequence of PDF object definitions inside a single container object. Combining this feature with filters, gives an extra level of compression to the document. Sophisticated attackers have been using this feature for deeper concealment of PDF objects that contain malicious code by wrapping them inside object streams. MD-

```
..JUNKDATA..%PDF-x.y..JUNKDATA..
1 0 obj <</tYpE
/C#61t#41log /#50#61#67#65#73 1 0 R
/Open#41ction<<
  /S/JavaScript/JS(eval(
            this.\
info.author);)>>>>
..JUNKDATA..

6 0 obj <<
  /Title <4D61 6C   69636
          96F757320446F63756D656E 74>
  /Author(app.al\145rt(
          {cMsg: 'Hell\157!'});)
>>

trailer<</Root 1 0 R/Info 6 0 R>>

..JUNKDATA..
```

FIGURE 3.3: An obfuscated version of the document shown in Figure 3.2 that is still rendered normally by Adobe Reader. Note that a part of the original JavaScript code has been stored into a non-code object, and that no PDF filters are used.

Scan handles object streams by identifying objects with a `/Type` key that has the value `/ObjStm` in the object's dictionary.

3.1.2 Emulation of the JavaScript for Acrobat API

Adobe Reader provides an extensive API that allows authors to create feature-rich documents with a wide range of functionality. The JavaScript for Acrobat API is accessible as a set of JavaScript extensions that provide document-specific objects, properties, and methods. Unfortunately, attackers can take advantage of this versatile API to obfuscate further their malicious documents. This can be achieved by embedding parts of the JavaScript code, or actual data on which it depends, into objects or elements that are accessible only through the Acrobat API. The malicious code can then retrieve its missing parts or access any hidden data through the Acrobat API, and continue its execution.

For instance, some malicious PDFs use the `info` property of the Adobe JavaScript Document Object Model (DOM) to store parts of code or data, as shown in Figure 3.3. The `info` property provides access to document metadata such as the document title, author, copyright notice, and so on. Moreover, the `info` property can be accessed by JavaScript code via the use of the `this.info` object, defined in the Adobe JavaScript DOM. Other objects that can hold data supplied by the attacker include annotations, XML specifications for embedded forms [24], or even document pages themselves. Some malicious samples conceal the shellcode as text to the pages of the document. Then, by using the JavaScript API functions

`getPageNumWords` and `getPageNthWord`, they fetch the concealed shellcode to JavaScript address space and trigger the vulnerable function.

It is clear from the above that the proper execution of the code embedded in a malicious PDF file requires an environment that provides the functionality offered by the Acrobat API. Unfortunately, standalone JavaScript engines such as SpiderMonkey [16], which is the engine used in MDScan, do not support this API and are not aware of the Adobe JavaScript DOM, since both are proprietary. In MDScan, we resolve this issue by augmenting the JavaScript engine with our own implementation of the DOM parts and the API calls that are most frequently used in malicious PDF documents. We have followed an incremental approach, adding more functions according to the ones found in the samples that we have encountered so far. After the completion of the data parsing and normalization steps, MDScan analyzes the identified objects and reconstructs the hierarchical structure of the DOM objects needed for the emulation of the implemented API calls.

3.1.3 JavaScript Code Extraction

After all extracted objects have been analyzed, we need to identify the objects that contain JavaScript code, and reconstruct the entire code image that will be fed to the JavaScript engine for execution. According to the PDF specification, objects that contain JavaScript code are denoted by the keyword `/JS`. The code can be located either in the object itself, or in some other object linked to the parent object through an indirect reference (or a chain of indirectly linked objects).

At this point, we aim to recover only the initial JavaScript code that is set to run automatically when the document is opened. A common practice of malicious PDF authors is to scatter this code across many objects with the aim to hinder detection and analysis. However, no matter into how many objects the code has been split, in order for the original code to be executable when the document is opened, the respective objects (or their associated parent objects) should all have been marked as containing JavaScript code using the `/JS` key. Any parts of the code that have been concealed into other non-code PDF objects are not relevant at this stage, since they will be retrieved at runtime through the appropriate API calls.

Having located the objects with the initial code to be executed, a crucial next step is to identify the entry point of the code. This can be achieved by looking for objects with specific declarations that denote immediate execution of the object's content, such as `/OpenAction`, `/AA`, `/Names`, and others [19]. The code of these objects is placed at the very bottom of the whole reconstructed code, so that it follows any previous function or variable declarations.

Another important aspect of the code extraction phase is the order in which the code fragments are arranged before being loaded on the JavaScript engine. For example, an attacker can place a statement that assigns a variable with the string representation of the shellcode in a PDF object, and access that variable from code located in another object. If the code of the second object (variable access) precedes the code of the first object (variable definition), the JavaScript

interpreter will issue a reference error. In most cases, the correct order of the code chunks can be inferred from the inherent ordering of the PDF objects in the file, and the chains of indirect references. However, we also use some additional heuristics to identify any use-before-declaration conditions, and reorder the respective code chunks appropriately.

3.2 Code Execution and Shellcode Detection

Having extracted the embedded code found in the document, MDScan proceeds into the dynamic analysis phase, in which the code is executed on a JavaScript interpreter. In most malicious PDF files, the goal of the JavaScript code is to trigger a vulnerability in the PDF viewer, and divert the normal execution flow to the embedded shellcode. The shellcode can be initially concealed using multiple layers of encryption or transformations, such as UTF-encoded characters, `eval` chains, mapping tables, or other complex custom schemes. However, during execution, its actual binary code will eventually be revealed into a contiguous buffer referenced through a JavaScript string variable [27, 43].

Strings in JavaScript are immutable, and thus a modification to an existing string or a concatenation of multiple strings results to the allocation of a new memory buffer. This allows us to detect a PDF document that contains malicious JavaScript code by scanning each newly created string for the presence of shellcode—a benign document would never execute JavaScript code that carries any form of shellcode. To that end, we have instrumented SpiderMonkey to scan the memory area of each allocated string using Nemu [39], a shellcode detector based on binary code emulation. The runtime heuristics of Nemu can identify the most widely used types of Windows shellcode, and their polymorphic or metamorphic variants, including egg-hunt shellcode [39], a tiny piece of code that scans the address space of the exploited process to locate and execute a second, much larger payload. which has been widely used in malicious PDFs [57].

Our approach is analogous to the one used by Egele et al. [27] for the detection of drive-by download attacks. In their system, the JavaScript engine of Mozilla Firefox has been instrumented to detect the presence of shellcode during the execution of malicious scripts embedded in rogue web pages that attempt to exploit vulnerabilities in popular web browsers. Unfortunately, we cannot directly modify the JavaScript engine of Adobe Reader since its source code is not available. An alternative approach would be to intercept the routines of the memory allocator used by Acrobat Reader through library interposition, and scan each newly allocated buffer, similarly to the design of Nozzle [43]. An advantage of this technique is that it eliminates the need for custom document parsing, data and code extraction, and emulation of the JavaScript for Acrobat API.

However, we have designed MDScan with the aim to be used as a standalone PDF scanner, and not as a protection enhancement for existing PDF viewers. This allows MDScan to be easily embedded as an additional detection component in ex-

isting intrusion detection systems, virus scanners, or proxy servers. In contrast, a detector integrated with the actual PDF viewer application, in the same spirit as the above browser-embedded systems [27, 43], cannot be easily used as a standalone component. Indeed, this would at least require a fully-blown virtual machine running Windows to host the instrumented viewer, and the viewer should be restarted for every input file. In fact, this design is being used by malicious code analysis systems like CWSandbox [31, 55], which can provide a detailed analysis of the actions and OS-wide side effects of malicious PDF files.

4 Experimental Evaluation

In this section we present the results of the experimental evaluation of our prototype implementation. First, we evaluate the detection effectiveness of MDScan using real PDF samples. We then evaluate the overall processing throughput, as well as the individual overhead of each analysis phase.

4.1 Data Set

For our experiments, we used a diverse set of 11,302 malicious documents gathered from public malware repositories and malicious websites [2,4,5,10], as well as from individual sources. These samples were reported to be malicious and implemented a wide variety of PDF exploits. The above set also includes nine samples generated using the nine different PDF exploit modules of the Metasploit Framework [17]. To measure false negatives, we used a set of 2,000 randomly chosen benign PDF files that we found through Google.

4.2 Detection Effectiveness

We began our evaluation by testing the detection effectiveness of MDScan using real malicious PDF samples. From the 11,302 malicious files, MDScan successfully detected 9,079 (80.3%). From the files that were not detected, 1350 were flagged to have a suspicious behavior, 405 did not attempt to exploit any arbitrary code execution vulnerability, but relied on other features such as `/Launch` and `/URI`, as discussed in Chapter 2. We plan to extend the PDF parsing module to detect these types of attacks by checking the extracted objects for the relevant key-

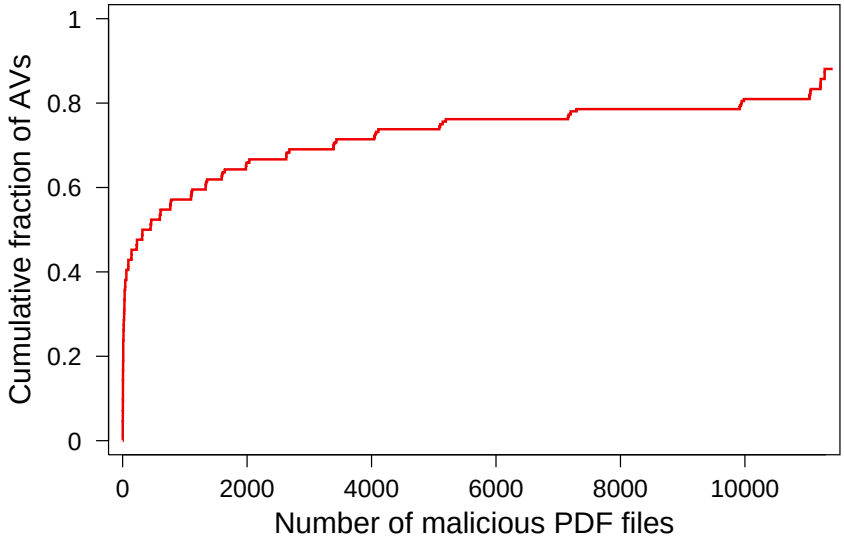

FIGURE 4.1: Cumulative fraction of the virus scanners of VirusTotal that detected a set of 11302 malicious PDF samples.

words. The remaining samples were not detected due to faults during the parsing phase, which we have been investigating.

For comparison, we submitted all samples to VirusTotal [18] and retrieved the results from 41 antivirus engines (AVs), which we have plotted in Figure 4.1. About half of the samples were detected by 70% of the antiviruses. Despite this high detection rate of the antivirus engines, even for the most detectable samples there were always about 15% of the AVs that did not detect them. To compare the evolution of detection rate through time, we scanned a subset of our malicious data set, consisting of 197 PDF files, at February and September of 2011 with AVs of VirusTotal. As we can observe in Figure 4.2, AVs are constantly improving towards PDF threats. Detection rate has been increased for the half of our samples, from 50% at February, to approximately 65%, 8 months later. We also submitted all samples to Wepawet [25, 29], which reported 8,614 files as malicious, 1,420 as suspicious, 1,228 as benign, while 40 resulted to error.

To further test the effectiveness of existing antivirus systems against PDF threats, we created variants of the nine Metasploit samples by applying additional obfuscation techniques. The first derived set was generated by obfuscating the JavaScript code of the original samples using a publicly available code obfuscator [6]. The second set was generated from the previous one by removing any PDF filter encodings from the objects that contained JavaScript code. We then treated the exposed JavaScript code in each object as a string, and encoded it using its hexadecimal representation.

As shown in Figure 4.3, in most cases the original Metasploit samples were detected by less than half of the AVs. The additional obfuscation applied in the

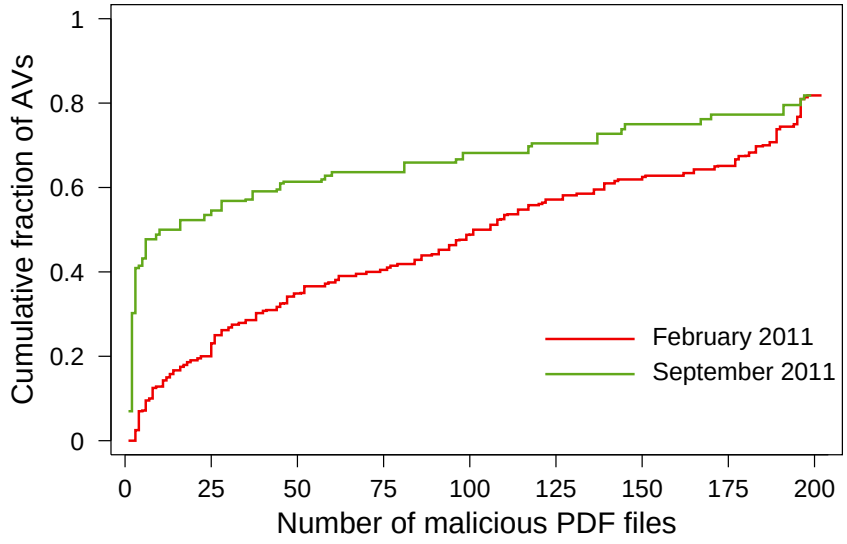

FIGURE 4.2: Cumulative fraction of virus scanners of VirusTotal that detected a set of 197 malicious PDF files at February and September 2011.

samples of the other two sets reduced the detection rate significantly, with only ten or less of the AVs detecting the malicious files in the third set. The only exception is the sample number three, which is detected by almost the same number of AVs irrespectively of the applied obfuscation, due to the inclusion of encrypted mediabox objects that were not altered by our modifications. MDScan successfully detected all 27 samples.

To make a step further, we repeated the previous experiment 8 months later to see which additional obfuscations contribute to lower the detection rate, and how samples are detected through time. Towards this, we used the 9 samples of the Metasploit Framework and created 6 sets of PDF documents. Each set was created by using the samples of the previous set. The first two sets were generating by using solely JavaScript obfuscations and the latter 4 sets incorporate PDF obfuscation techniques discussed in Chapter 3. All 63 PDF samples were scanned with MDScan and successfully identified as malicious.

As shown in Figure 4.4, additional obfuscation further decreases the detection rate. A key observation is that the detection rate of the original Metasploit samples varies from Figure 4.3 to Figure 4.4. In particular, in all nine samples detection rate has been increased confirming the increase reported at Figure 4.2. In only a minority of samples, additional obfuscation resulted in a slight increase of the detection rate, mainly due to trigger of different heuristics at some antivirus engines.

To measure the increase of detection rate through time, we resend each sample of the above experiment to VirusTotal every day. As we can observe in Figure 4.5, detection rate is slowly increasing for all variants of a specific sample. Moreover,

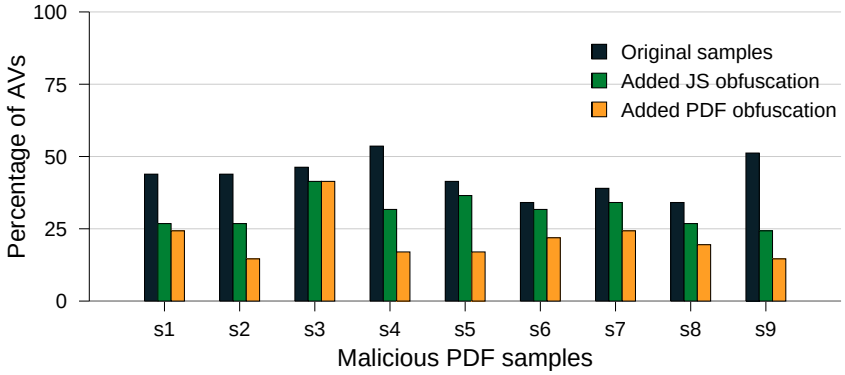

FIGURE 4.3: Percentage of virus scanners of VirusTotal that detected obfuscated versions of malicious PDF files generated using Metasploit at February 2011.

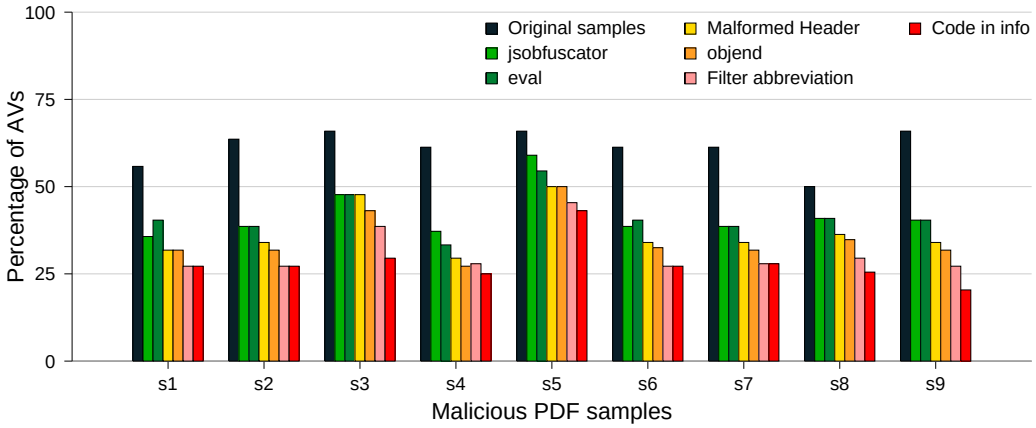

FIGURE 4.4: Percentage of virus scanners of VirusTotal, that detected obfuscated versions of malicious PDF files generated using Metasploit at September 2011.

detection rates of obfuscated versions of the sample, could not reach the detection rate of the original sample, several days after the first scan.

Finally, we tested MDScan for false positives using the set of benign files. After verifying that all 2,000 files were reported as benign by all AVs of VirusTotal, we scanned them using MDScan, which did not misclassify any of them.

4.3 Runtime Performance

We measured the processing time of MDScan for both malicious and benign samples. We repeated each experiment ten times and report the average values. Figure 4.6 shows the distribution of the processing time for all samples in our two datasets, and Figure 4.7 shows the breakdown of the average scanning time for the two sets. As expected, most of the processing time for malicious PDF files is spent

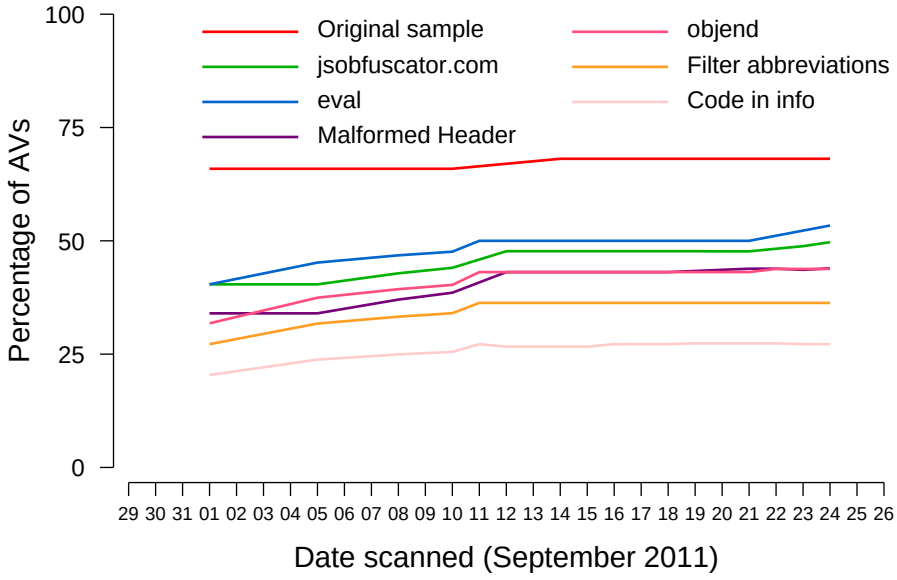

FIGURE 4.5: Evolution of detection rate for sample 9, at VirusTotal

on the emulation of the JavaScript code, which is a much more CPU-intensive operation compared to file parsing and code extraction. The average processing time for malicious inputs is just less than three seconds, with about half of the files being scanned in less than one second.

The average processing time for benign PDF files is 0.75s, with about 80% of the files being scanned in less than one second. In contrast to the malicious files, the amount of time spent on code execution is negligible, since only a small fraction of files contain JavaScript code. Instead, due to the very large size of some of the files, the time spent on parsing and analysis of the PDF objects in each file is significant.

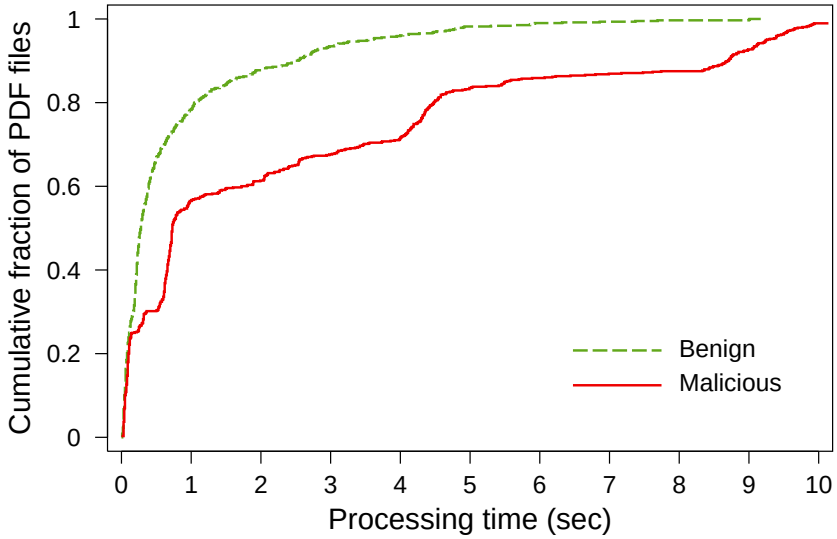

FIGURE 4.6: Cumulative distribution of the processing time for malicious and benign PDF samples.

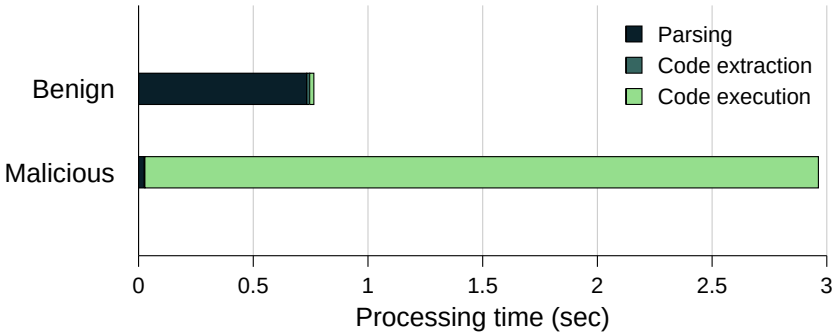

FIGURE 4.7: Average processing time for malicious and benign samples.

5
Case Studies

In this section we perform case study to investigate real attacks that can use PDF documents as a means for malware distribution. Our study starts with Twitter, a microblogging social network that has gained popularity recently. We try to investigate whether PDF attacks exist on this social network. We also, seek for the presence of malicious PDF documents to spam corpuses detected by spamtraps.

5.1 Case Study: Twitter

Twitter, introduced in 2006 is a microblogging social network that rapidly became popular throughout the world. A user in Twitter can posts messages—called tweets—limited to 140 characters on his public profile. Moreover, he can interact with other users by following them, thus watch their tweets or ”re-tweet” a user's tweet to his public profile. To categorize tweets, Twitter users include keywords (called hashtags) on their tweets. Tweets with the same hashtag, make this hashtag popular. As a result, popular hashtags appear to Twitter's first page as trending. Despite raw text, Twitter users share URLs as well. To bypass the 140 character limitation, URLs are shortened, using publicly available URL shortening services.

By September 2010, 90 million tweets are posted on Twitter, 25% of them containing URLs [42] Due to its widespread adoption, Twitter is a target for nefarious activities. 8% of 25 million URLs posted to Twitter, lead to potentially malicious sites. [30] The use of shortening services makes detection of malicious URLs a difficult task. Attack campaigns on Twitter also involved the use of malicious documents [46]. Using the latter as a motivation, we want to investigate to what extent Twitter contains links that point to malicious PDF documents.

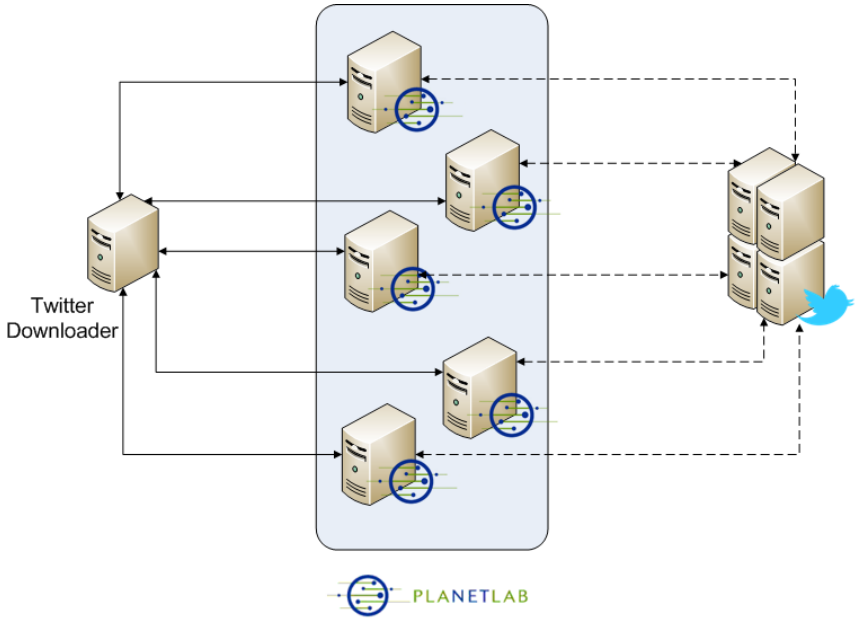

FIGURE 5.1: Architecture of the infrastructure used to download URLs from Twitter

5.1.1 Design and Implementation

In order to investigate whether URLs that lead to malicious documents circulate Twitter, we need to monitor each tweet contained a valid URL. Twitter offers an API to search on its tweet database [14]. We used this API to search throughout tweets for the presence of a valid URL. Due to the very large amount of URLs overwhelm Twitter every day, and due to restrictions faced on Twitter Search API, both in terms of time and of amounts of results returned, we have a smaller set of URLs available. Twitter Search API rate-limits search requests, but the exact number of is not available to the public. Moreover, results returned for each search request issued, are limited to only last 1500.

To overcome these obstacles, we came up with a custom-built Twitter searcher/downloader. It is split in two parts:

- The searcher, that utilizes the Twitter Search API to fetch all results of a given query issued by the downloader, parse each tweet for potential URLs, and send extracted URLs to downloader.
- The downloader, which does the whole job of downloading URLs and discriminate which of these URLs actually point to PDF documents as long as issues search requests and waits results from the searcher. PDF documents pointed by URLs, are kept on a separate space, while others are discarded.

By deploying the searcher to various PlanetLab [13] nodes and the downloader to a local machine, we issued a variety of synthesized queries using the down-

loader, as shown in Figure 5.1, managing to have a constant stream of URLs that were downloading. Queries issued are synthesized and consisted from two parts. The first part is consists of the Top Level Domain string (TLDs) where most popular URL shorteners are being hosted. and the rest is filled with one of the most 500 common words in the English language. We issuing constantly the same synthesized queries to retrieve possible new URLs.

5.1.2 Results

We ran the experiment for several months collecting approximately 100 millions of unique URLs. By running the downloader for a 4-month period, download rate was approximately 1 million URLs per day. From URLs downloaded, we are able to identify, 127728 PDFs. (0.0018 %) All downloaded PDFs were scanned with MDScan, but no evidence for malicious behavior was reported.

5.1.3 Proof of Concept attack

Twitter has been used for several malicious activities. [7] Despite, the efforts being made to improve security on Twitter, attackers still manage to bypass security protections.

In this section we are going to demonstrate a simple PDF-based attack. We used Metasploit Framework to construct a PDF, exploiting a known vulnerability. Fake arguments are given to Metasploit options in order not to harm potential victims. We tested the malicious PDF against Wepawet and VirusTotal, and both services detected them as malicious. Moreover we created fake accounts at Yahoo, Wordpress and Twitter via Tor network. Wordpress was used to upload the malicious PDF, and Twitter account was used to share the URL. Bit.ly shortening service was used, as well to shorten the URL provided by Wordpress and to monitor possible visitors through its statistics page.

Several days after the tweet, both WordPress and Twitter served the URL and the Tweet respectively. Only bit.ly issued a warning for that link, one week after.

5.2 Case Study: Spam Corpus

As discussed in Chapter 2, attackers use fake e-mails at targeted attacks. In this section we seek to answer whether spam campaigns include any form of PDF documents, and whether these document exploit any vulnerabilities on PDF viewer.

Towards this, we collected 24,063,681 spam messages from spamfeed.me [15] and scan them for the presence of any attachments or URLs included. Unfortunately we did not found any PDF attachments or URLs that would lead us to PDF documents.

Possibly attackers have now focused to targeted attacks and are not interested in massive spam campaigns. Because targeted mail is received by only few people, it cannot be detected by antispam companies.

6
Limitations

The JavaScript for Acrobat API exposes an extensive set of features through numerous API calls. Clearly, our approach of emulating the functionality of the various API calls found in malicious PDFs will not scale well if attackers start to use a much broader range of API calls in their code. Furthermore, the complexity of implementing some of the calls might be prohibitive. Still, MDScan would be useful as a first-level detector, and can easily be extended to offload the analysis of PDF files that use unsupported API calls to a fully-blown dynamic analysis system such as Wepawet [25, 29] or CWSandbox [31, 55].

Another advantage of MDScan compared to VM-based systems is that it is agnostic about the particular vulnerability that a malicious PDF may exploit. This is important for exploits that are effective against only specific versions of the PDF viewer application. In such cases, a VM-based analysis system may not observe the actual malicious behavior of the embedded code simply because it does not use the appropriate PDF viewer version.

Currently, MDScan detects only malicious PDFs that exploit vulnerabilities in the JavaScript API of the PDF viewer. As part of our future work, we plan to extend our system to detect other types of malicious activity, such as startup actions though features like `/Launch` and `/URI`, or embedded malicious Flash files [28]. Such attacks can be easily identified at the parsing phase by analyzing the extracted PDF objects.

Finally, an attacker could exploit idiosyncrasies of the PDF viewer's JavaScript engine that may not be exhibited by the interpreter used in MDScan. For example, the JavaScript engine of Adobe Reader does not allow the type of defined global variables to be changed on subsequent assignments, a behavior not common among other JavaScript engines [40, 56]. Any other similar deviations in the behavior of

Adobe Reader's interpreter should be emulated by the JavaScript engine of the detector.

7
Related Work

The proliferation of PDF threats has resulted to many efforts on the manual and automated analysis of malicious PDF files. Researchers have been constantly analyzing and discovering new obfuscation techniques and tricks being used in recently discovered PDF malware samples [19, 24, 28, 44, 54, 56, 57]. Tools and frameworks like Origami [44], malware tracker [9] and PDF Tools [49] help researchers to parse and analyze malicious PDF files. Dynamic malware analysis systems like Wepawet [25, 29], CWSandbox [31, 55], PDF X-Ray [12] can provide a detailed analysis of PDF malware, including the actions of the malicious code after successful exploitation.

Nozzle [43] detects heap-spraying attacks mounted by malicious web sites against browsers. Using library interposition, Nozzle monitors the frequency of the calls to the memory allocator's routines and also analyzes the contents of the allocated areas for the presence of binary code. Given that heap-spraying is also frequently used in malicious PDFs, Nozzle can also potentially detect a malicious PDF when rendered within the browser However, more stealthy heap-spraying can evade this detection approach [26]. Egele et al. [27] also propose a system for the detection of attacks that use malicious JavaScript code against the browser. As in MDscan, the malicious code is identified by instrumenting the JavaScript interpreter of Mozilla Firefox to detect the presence of shellcode that is revealed during execution in the address space of the interpreter. Rieck et al. [45] propose a system that inspects web pages using both static and dynamic code analysis to extract malicious patterns using machine learning. This system is embedded as a web proxy, and achieves high detection rate along with excellent run-time performance. Moreover Snow et al [48] stated the disadvantages of software-based emulation, and proposed a framework that uses hardware virtualization to detect code injec-

tion attacks. As a case study, a corpus of malicious PDF documents was analyzed using the proposed framework.

8
Conclusion

Malicious PDF files remain an important threat, requiring effective and robust detection mechanisms. As we have demonstrated, the effectiveness of existing antivirus systems against malicious PDF files is quite modest, given that in most cases the samples were well known and quite old, and at the same time is highly affected by the application of simple obfuscation techniques.

In this thesis we presented MDScan which is not affected by JavaScript code obfuscation, and is robust against most of the known obfuscation techniques based on intricacies of the PDF format specification. At the same time, it does not rely on any specific vulnerability or exploit features, which allows the detection of previously unknown threats. Combined with its standalone design, we believe that these features make MDScan an effective detection component for larger network or host-level attack detection systems. However, due to its emulation of the JavaScript for Acrobat API, MDScan will probably need to be combined with VM-based analysis systems in case PDF threats start to employ more advanced or diverse API calls.

Moreover we conducted a study towards identifying malicious documents in the wild, both at a popular social network and at spam mail corpuses.

Bibliography

[1] http://www.jailbreakme.com/_/.

[2] http://www.malwaredomainlist.com/.

[3] Adobe. http://www.adobe.com/.

[4] BLADE - Block All Drive-by Download Exploits. http://www.blade-defender.org/.

[5] Contagio Malware Dump. http://contagiodump.blogspot.com/.

[6] Free Javascript Obfuscator. http://www.javascriptobfuscator.com/.

[7] Hackers Use Twitter to Control Botnet — Threat Level. http://www.wired.com/threatlevel/2009/08/botnet-tweets/.

[8] ISO 32000-1:2008. http://www.iso.org/iso/catalogue_detail.htm?csnumber=51502.

[9] malware tracker. http://www.malwaretracker.com/.

[10] Offensive Computing — Community Malicious code research and analysis. http://www.offensivecomputing.net/.

[11] PDF reference and Adobe extensions to the PDF specification. http://www.adobe.com/devnet/pdf/pdf_reference.html.

[12] PDF X-RAY. https://www.pdfxray.com/.

[13] PlanetLab — An open platform for developing, deploying, and accessing planetary-scale services. http://www.planet-lab.org/.

[14] REST API Resources — Twitter Developers. https://dev.twitter.com/docs/api.

[15] Spamfeed.me. http://spamfeed.me.

[16] SpiderMoneky (Javascript-C) Engine. http://www.mozilla.org/js/spidermonkey/.

[17] The Metasploit Project. http://www.metasploit.com/.

[18] Virustotal. http://www.virustotal.com/.

[19] 4 ways to die opening a PDF, 2009. http://esec-lab.sogeti.com/dotclear/index.php?post/2009/06/26/68-at-least-4-ways-to-die-opening-a-pdf.

[20] PDF Most Common File Type in Targeted Attacks - F-Secure Weblog, May 2009. http://www.f-secure.com/weblog/archives/00001676.html.

[21] How many ways can you remotely exploit an iPhone?, August 2010. http://www.f-secure.com/weblog/archives/00002003.html.

[22] PDF Based Targeted Attacks are Increasing - F-Secure Weblog, March 2010. http://www.f-secure.com/weblog/archives/00001903.html.

[23] JailbreakMe Lulz, July 2011. http://www.f-secure.com/weblog/archives/00002199.html.

[24] M. Cova. Malicious PDF trick: XFA. http://www.cs.bham.ac.uk/~covam/blog/pdf/.

[25] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th International World Wide Web Conference (WWW)*, 2010.

[26] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou. Heap taichi: exploiting memory allocation granularity in heap-spraying attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.

[27] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the 6th international conference on Detection of Intrusions and Malware, & Vulnerability Assessment (DIMVA)*, 2009.

[28] E. Filiol. New viral threats of PDF language. Black Hat Europe, March 2008.

[29] S. Ford, M. Cova, C. Kruegel, and G. Vigna. Wepawet. http://wepawet.cs.ucsb.edu/.

[30] C. Grier, K. Thomas, V. Paxson, and M. Zhang. @spam: The Underground on 140 Characters or less. In *ACM Conference on Computer and Communications Security'10*, pages 27–37, 2010.

[31] T. Holz. Analyzing malicious pdf files, 2009. http://honeyblog.org/archives/12-Analyzing-Malicious-PDF-Files.html.

[32] M. Hypponen. Intelligence Sector Hit by a Targeted Attack - F-Secure Weblog, Januray 2010. http://www.f-secure.com/weblog/archives/00001862.html.

[33] M. Hypponen. On-going Targeted Attacks Against US Military Contractors - F-Secure Weblog, Januray 2010. http://www.f-secure.com/weblog/archives/00001859.html.

[34] M. Hypponen. Military Targets - F-Secure Weblog, July 2011. http://www.f-secure.com/weblog/archives/00002203.html.

[35] P. Kunert. Phishers switch focus to targeted attacks, warns Cisco, July 2011. http://www.theregister.co.uk/2011/07/01/cybercrims_shift_focus/.

[36] J. Leyden. iOS jailbreak howdunnit partially solved, August 2010. http://www.theregister.co.uk/2010/08/03/ios_jailbreak_howdunnit/.

[37] J. Leyden. Only jailbroken iPhones, iPads can be safe from latest vuln, July 2011. http://www.theregister.co.uk/2011/07/07/jailbreak_security_risk/.

[38] W.-J. Li, S. Stolfo, A. Stavrou, E. Androulaki, and A. D. Keromytis. A study of malcode-bearing documents. In *Proceedings of the 4th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 231–250, 2007.

[39] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, December 2010.

[40] S. Porst. How to really obfuscate your PDF malware. RECON, July 2010.

[41] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFRAMEs point to us. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[42] L. Rao. Twitter Seeing 90 Million Tweets Per Day, 25 Percent Contain Links, September 2010. http://techcrunch.com/2010/09/14/twitter-seeing-90-million-tweets-per-day/.

[43] P. Ratanaworabhan, B. Livshits, and B. Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th USENIX Security Symposium*, Aug. 2009.

[44] F. Raynal, G. Delugré, and D. Aumaitre. Malicious origami in pdf. *J. Comput. Virol.*, 6(4):289–315, November 2010.

[45] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 31–39, New York, NY, USA, 2010. ACM.

[46] Sean-Paul Correll. Dont Get Caught by the Grinch on Twitter, December 2010. http://pandalabs.pandasecurity.com/dont-get-caught-by-the-grinch-on-twitter/.

[47] K. Selvaraj and N. F. Gutierres. The rise of PDF malware, 2010. http://www.symantec.com/connect/blogs/rise-pdf-malware.

[48] K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos. SHELLOS: enabling fast detection and forensic analysis of code injection attacks. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[49] D. Stevens. PDF tools. http://blog.didierstevens.com/programs/pdf-tools/.

[50] D. Stevens. PDF filter abbreviations, 2008. http://blog.didierstevens.com/2009/05/11/pdf-filter-abbreviations/.

[51] D. Stevens. PDF stream objects, 2008. http://blog.didierstevens.com/2008/05/19/pdf-stream-objects/.

[52] D. Stevens. Escape from PDF, 2010. http://blog.didierstevens.com/2010/03/29/escape-from-pdf/.

[53] D. Stevens. Quickpost: PDF header %!PS-adobe-n.n PDF-m.m, 2010. http://blog.didierstevens.com/2010/01/21/quickpost-pdf-header-ps-adobe-n-n-pdf-m-m/.

[54] D. Stevens. Malicious PDF documents explained. *IEEE Security and Privacy*, 9(1):80–82, 2011.

[55] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.

[56] J. Wolf. OMG WTF PDF. 27th Chaos Communication Congress (27C3), December 2010.

[57] B. Zdrnja. Sophisticated, targeted malicious pdf documents exploiting cve-2009-4324, 2010. http://isc.sans.edu/diary.html?storyid=7867.