Combining Static and Dynamic Analysis for the Detection of Malicious Documents

Zacharias Tzermias,[*] Giorgos Sykiotakis,[†] Michalis Polychronakis,[‡] Evangelos P. Markatos[*]
[*]Institute of Computer Science, Foundation for Research and Technology—Hellas, Greece
[†]University of Crete, Greece
[‡]Columbia University, USA
{tzermias, markatos}@ics.forth.gr, sykiotak@csd.uoc.gr, mikepo@cs.columbia.edu

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. As vulnerabilities in the major PDF viewers keep surfacing, effective detection of malicious PDF documents remains an important issue. In this paper 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, even when they have been extensively obfuscated.

1. INTRODUCTION

The Portable Document Format (PDF) is one of the most popular file formats for document exchange. As the focus of attackers has recently shifted from server-side to client-side attacks, the universal adoption of the PDF format has rendered PDF documents a prime vector for malware distribution [22]. 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 [13], 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. 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.

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 [17]. 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.

In this paper 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. Our experimental evaluation with real and generated malicious documents, as well as benign PDF files, shows that MDScan can accurately detect a broad range of malicious documents, even when they have been highly obfuscated, while it has a reasonable runtime processing overhead.

2. BACKGROUND

PDF, created by Adobe Systems, has become the de facto file format for the distribution of printable documents. A file adhering to the PDF specification has four main sections: a one-line header with the version number of the PDF specification; the main body of the document, which consists of objects such as text, images, fonts, annotations, or even other embedded files; a cross-reference table with the offsets of the objects within the file; and finally, a trailer for quick access to the cross-reference table and other special objects.

Besides static data, PDF objects can also contain code written in JavaScript. This allows document authors to incorporate sophisticated features such as form validation, multimedia content, or even communication with external systems and applications. Unfortunately, attackers can also

EUROSEC '11, Salzburg, Austria

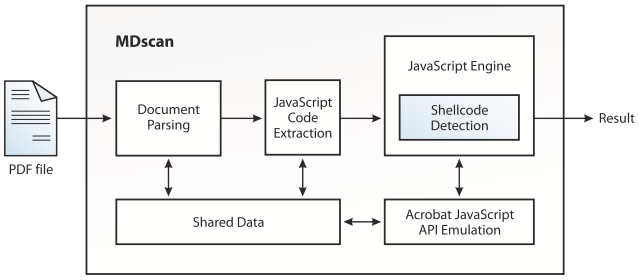

Figure 1: Overall architecture of MDScan.

take advantage of the versatility offered by JavaScript 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 [13] 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 [14], which can open external resources form the same host or the Internet. Although in both cases the application first asks for user authorization, such features are quite hazardous, and after the public exposure of their security implications they were promptly mitigated.

3. DESIGN AND IMPLEMENTATION

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. 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 Fig. 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 [22, 24, 26] of the PDF specification make this process a non-trivial task. In addition, most PDF viewers, including Adobe Reader, attempt 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 Fig. 2, along with other required (according to the PDF specification) elements, such as the `endobj`, `endstream`, and `%%EOF` keywords. 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 object stream data and conceal the embedded JavaScript code. The PDF format supports many different filters for the decompression of arbitrary data (`/FlateDecode`, `/LZWDecode`, `/RunLengthDecode`), the decompression of images (`/JBIG2-Decode`, `/CCITTFaxDecode`, `/DCTDecode`, `/JPXDecode`), or the decoding of arbitrary 8-bit data that have been encoded as ASCII text (`/ASCIIHexDecode`, `/ASCII85Decode`).

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. Special care is taken for the correct handling of filter abbreviations 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 Fig. 3. Similarly, strings in objects can be represented using various other encodings, such as octal or hexadecimal representations with flexible character whitespace requirements [24]. Finally, version 1.5 of the PDF specification introduced the concept of *object streams*, which contain a sequence of PDF objects. Sophisticated attackers have been using this feature for deeper concealment of PDF objects that contain malicious code by wrapping them inside object streams. MDScan 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-

```
%PDF-1.1
1 0 obj <<
  /Type /Catalog
  /Pages 1 0 R
  /OpenAction <<
    /S /JavaScript
    /JS (app.alert({cMsg: 'Hello!'});)
  >>
>>
endobj

2 0 obj <<
  /Title (Malicious Document)
>>

trailer <<
  /Root 1 0 R
  /Info 2 0 R
>>
```

Figure 2: A malformed (missing cross-reference table, an `endobj` keyword, and `%%EOF`) PDF document that is rendered normally by Adobe Reader.

```
..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: An obfuscated version of the document shown in Fig. 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.

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 Fig. 3. The `info` property provides access to document metadata such as the document title, author, copyright notice, and so on. Other objects that can hold data supplied by the attacker include annotations, XML specifications for embedded forms [10], or even the document pages themselves. For example, the script can read the actual words of a page using the `getPageNthWord` 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 [1], 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 [9]. 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, 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 [13, 20].

Strings in JavaScript are immutable, and thus a modification to an existing string 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 [18], a shellcode detector based on binary code emulation. The runtime heuristics of Nemu can identify the most widely used types of Windows shellcode, including egg-hunt shellcode [18], which has been widely used in malicious PDFs [27].

Our approach is analogous to the one used by Egele et al. [13] 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. 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 [20]. 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 existing 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 [13, 20], 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 [16, 25], 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.

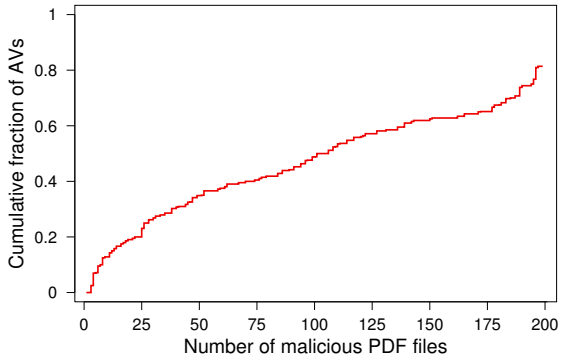

Figure 4: Cumulative fraction of the virus scanners of VirusTotal that detected a set of 197 malicious PDF samples.

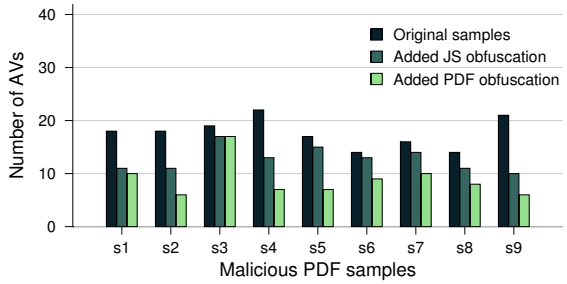

Figure 5: Number of virus scanners (out of 41) of VirusTotal that detected obfuscated versions of malicious PDF files generated using Metasploit.

For our experiments, we used a diverse set of 197 malicious documents gathered from public malware repositories and malicious websites [2–5], as well as from individual sources. The above set also includes nine samples generated using the nine different PDF exploit modules of the Metasploit Framework [6]. We also used a set of 2,000 randomly chosen benign PDF files that we found through Google.

4.1 Detection Effectiveness

We began our evaluation by testing the detection effectiveness of MDScan using real malicious PDF samples. From the 197 malicious files, MDScan successfully detected 176 (89%). From the files that were not detected, 15 did not attempt to exploit any arbitrary code execution vulnerability, but relied other features such as `/Launch` and `/URI`, as discussed in Sec. 2. We plan to extend the PDF parsing module to detect these types of attacks by checking the extracted objects for the relevant keywords. The remaining six samples were not detected due to faults during the parsing phase, which we have been investigating.

For comparison, we submitted all samples to VirusTotal [7] and retrieved the results from 41 antivirus engines (AVs), which we have plotted in Fig. 4. About half of the samples were detected only by half or less of the AVs, while 24 samples were detected by 20% or less. Even for the most detectable samples, there were about 20% or more of the AVs that did not detect them. We also submitted all samples to Wepawet [11, 15], which reported 119 files as malicious, 16 as suspicious, 58 as benign, while four resulted to error.

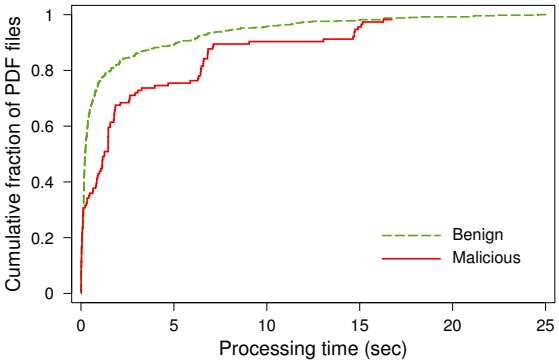

Figure 6: Cumulative distribution of the processing time for malicious and benign PDF samples.

To test further 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 [8]. 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 Fig. 5, in most cases the original Metasploit samples were detected by less than half of the AVs. The additional obfuscation applied in the 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.

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.

4.2 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 6 shows the distribution of the processing time for all samples in our two datasets, and Fig. 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 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 1.5s, 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.

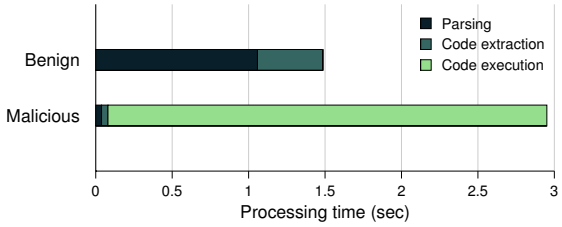

Figure 7: Average processing time for malicious and benign samples.

5. 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 [11, 15] or CWSandbox [16, 25].

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 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 [14]. 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 [19, 26]. Any other similar deviations in the behavior of Adobe Reader's interpreter should be emulated by the JavaScript engine of the detector.

6. 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 [9, 10, 14, 21, 24, 26, 27]. Tools and frameworks like Origami [21] and PDF Tools [23] help researchers to parse and analyze malicious PDF files. Dynamic malware analysis systems like Wepawet [11, 15] and CWSandbox [16, 25] can provide a detailed analysis of PDF malware, including the actions of the malicious code after successful exploitation.

Nozzle [20] detects heap-spraying attacks mounted by ma-

licious 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 [12]. Egele et al. [13] 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.

7. 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.

MDScan 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.

Acknowledgments

This work was supported in part by the FP7-PEOPLE-2009-IOF project MALCODE and the FP7 project SysSec, funded by the European Commission under Grant Agreements No. 254116 and No. 257007, and by the project i-Code, funded by the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme of the European Commission—Directorate-General for Home Affairs under Grant Agreement No. JLS/2009/CIPS/AG/C2-050. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained herein. Zacharias Tzermias and Evangelos Markatos are also with the University of Crete.

8. REFERENCES

[1] http://www.mozilla.org/js/spidermonkey/.
[2] http://www.blade-defender.org/.
[3] http://www.malwaredomainlist.com/.
[4] http://www.offensivecomputing.net/.
[5] http://contagiodump.blogspot.com/.
[6] http://www.metasploit.com/.
[7] http://www.virustotal.com/.
[8] http://www.javascriptobfuscator.com/.
[9] 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.
[10] M. Cova. Malicious PDF trick: XFA. http://www.cs.bham.ac.uk/~covam/blog/pdf/.
[11] 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.
[12] 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.
[13] 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.
[14] E. Filiol. New viral threats of PDF language. Black Hat Europe, March 2008.
[15] S. Ford, M. Cova, C. Kruegel, and G. Vigna. Wepawet. http://wepawet.cs.ucsb.edu/.
[16] T. Holz. Analyzing malicious pdf files, 2009. http://honeyblog.org/archives/12-Analyzing-Malicious-PDF-Files.html.
[17] 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)*, 2007.
[18] 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.
[19] S. Porst. How to really obfuscate your PDF malware. RECON, July 2010.
[20] 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.
[21] F. Raynal, G. Delugré, and D. Aumaitre. Malicious origami in pdf. *J. Comput. Virol.*, 6(4):289–315, November 2010.
[22] K. Selvaraj and N. F. Gutierres. The rise of PDF malware, 2010. http://www.symantec.com/connect/blogs/rise-pdf-malware.
[23] D. Stevens. PDF tools. http://blog.didierstevens.com/programs/pdf-tools/.
[24] D. Stevens. Malicious PDF documents explained. *IEEE Security and Privacy*, 9(1):80–82, 2011.
[25] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.
[26] J. Wolf. OMG WTF PDF. 27th Chaos Communication Congress (27C3), December 2010.
[27] B. Zdrnja. Sophisticated, targeted malicious pdf documents exploiting cve-2009-4324, 2010. http://isc.sans.edu/diary.html?storyid=7867.