Processing Dangerous Paths – On Security and Privacy of the Portable Document Format

Jens Müller
Ruhr University Bochum
jens.a.mueller@rub.de

Dominik Noss
Ruhr University Bochum
dominik.noss@rub.de

Christian Mainka
Ruhr University Bochum
christian.mainka@rub.de

Vladislav Mladenov
Ruhr University Bochum
vladislav.mladenov@rub.de

Jörg Schwenk
Ruhr University Bochum
joerg.schwenk@rub.de

***Abstract*—PDF is the de-facto standard for document exchange. It is common to open PDF files from potentially untrusted sources such as email attachments or downloaded from the Internet. In this work, we perform an in-depth analysis of the capabilities of malicious PDF documents. Instead of focusing on implementation bugs, we abuse legitimate features of the PDF standard itself by systematically identifying *dangerous paths* in the PDF file structure. These dangerous paths lead to attacks that we categorize into four generic classes: (1) Denial-of-Service attacks affecting the host that processes the document. (2) Information disclosure attacks leaking personal data out of the victim's computer. (3) Data manipulation on the victim's system. (4) Code execution on the victim's machine. An evaluation of 28 popular PDF processing applications shows that 26 of them are vulnerable at least one attack. Finally, we propose a methodology to protect against attacks based on PDF features systematically.**

I. Introduction

The Portable Document Format (PDF) is arguably the most widely used data format for office document exchange. While the total number of PDF files is impossible to guess, Adobe announced that 250 billion documents have been opened by Adobe products in 2018 [59]. Being true or not, PDF documents are heavily used in business to business as well as consumer use cases. Exchanging, creating, and archiving invoices and contracts, submitting scientific papers, or collaborating and reviewing texts, are only some scenarios which are hardly imaginable without PDF.

The advantage of using PDF over other document formats, such as Microsoft Word, is its availability on all platforms, including mobile and web. PDF processors are even used on the server-side. For example, uploaded PDF files are converted into images to preview them in forums, wikis, or cloud storage. Modern printers also directly support native PDF processing without the requirement for printer drivers to convert input files to a special data format understood by the printer.

A. Powerful Document Features

Introduced in 1993 by Adobe, PDF was designed to provide a consistent representation of documents, independent of the platform. It supports numerous advanced features, ranging from cryptography to calculation logic [44], 3D animations [5], JavaScript [2], up to form fields [6]. A PDF document can be updated or annotated without losing previous revisions [7] and define specific actions [4], for example, to display a specific page once the viewer opens the document. On top of this, PDF is enriched with different data formats which can be embedded into documents, such as XML [8], or Flash [3]. Each of the formats has its strengths, but allowing their inclusion also enables their weaknesses and concerns. In this work, we analyze the security of native PDF functions.

B. Security and Privacy Threats

We present a systematic and structured analysis of standard PDF features relevant for the security and privacy of users. Even though PDF is a relatively old and well-established data format, our study reveals novel insights regarding the abuse of dangerous features, which are induced by opening a malicious PDF document. We categorize our attacks into four classes:

1) Denial-of-Service (DoS) attacks affecting the processing application and the host on which the PDF file is opened.
2) Information disclosure attacks leak personal data from the victim's computer to the attacker, such as PDF document form data, local files on disk, or NTLM user credentials.
3) Data manipulation attacks modify PDF form values, write local files on the host system, or mask the displayed content of a document based on the opening application.
4) Execution of code on the victim's machine, by silently launching an executable, embedded within the document.

C. Responsible Disclosure

We reported our attacks and findings to the affected vendors and proposed appropriate countermeasures, resulting in CVE-2020-28352, CVE-2020-28353, CVE-2020-28354, CVE-2020-28355, CVE-2020-28356, CVE-2020-28357, CVE-2020-28358, CVE-2020-28359, CVE-2020-28410, CVE-2020-28411, and CVE-2020-28412. While it is possible to mitigate most of the attacks on the implementation-level, all of them are based on legitimate features defined in the PDF standard. To sustainably eliminate the root cause of these vulnerabilities in future implementation, the authors recommend to remove dangerous functionality from the PDF specification or add proper implementation advice to its security considerations.

D. Contributions

Our contributions can be summarized as follows:

- We present a systematic analysis on the security of native PDF features. Therefore, we distillate *dangerous paths* from the 1300-page PDF specification. (section V)

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.23109
www.ndss-symposium.org

- Based on this methodology, we craft our attack vectors, resulting in an overall of 209 different attack variants that can be generalized into four attack classes. (section VI)
- We evaluate 28 popular PDF viewers and show that 26 of them are vulnerable to at least one attack. (section VII)
- We present techniques for JavaScript-based fingerprinting of PDF viewers and bypassing Digital Rights Management, and discuss the consequences of hidden data added by legitimate PDF editors to every document. (section VIII)
- We discuss countermeasures for PDF implementations as well as the specification, and propose a methodology to systematically protect against attack variants. (section IX)
- We release our comprehensive suite of malicious PDF files which can be used by developers to test their software.[1]

II. PDF Basics

This section briefly introduces the PDF document structure. For reasons of clarity, we only describe the building blocks relevant for understanding the attacks of this paper.

A. Basic Blocks

A PDF document consists of four basic sections:

1) A header defining the PDF document version (1.1 to 2.0).
2) A body containing the content, a bundle of PDF objects.
3) An index table with references to each object in the body.
4) A trailer defining the root element of the document and a reference to the index table.

The most important section is the body which contains the PDF objects – the actual content of the document. An object can, for example, define a headline, a text block, or an image.

```
10 0 obj
<< /Length 10 >>                                   % stream length
stream                                         % start of the stream
  Content          % content (e.g., text, image, font, file)
endstream                                        % end of the stream
endobj
```

Listing 1. PDF object `10`, including a 10-byte content stream.

Every object is enclosed by the delimiters `obj` and `endobj` and has has an identifier. In Listing 1, the object's identifier is `10` with generation number `0`. Content can be provided as a string, or – as shown in Listing 1 – as a stream enclosed by `stream` and `endstream`. It can be prefaced with additional information, such as encoding or length. Streams can optionally be compressed. Many documents use `FlatDecode` for this purpose, meaning that the zlib *Deflate* algorithm is used.

B. PDF Forms

With PDF version 1.2, Adobe introduced *AcroForm*s in 1996. Similarly to HTML forms, AcroForms allow to define input fields, checkboxes, and buttons. The user-input can either be stored directly into the document (using incremental updates) or be submitted to a dedicated server. In the latter case, AcroForms use the *Forms Data Format* (FDF), which is based on raw PDF objects, for transmitting the data.

C. Actions & JavaScript

The PDF specification defines multiple *Actions* for various purposes. These actions can be used, for example, to navigate to a certain page in the document (*GoTo* action). Actions are often combined with form elements or *Annotations* (e.g., clickable hyperlinks referencing a website are technically realized by combining a *Link* annotation with a *URI* action). However, actions can also be set to trigger automatically based on various events such as opening, printing, or closing the document.

A special action in PDF is the execution of JavaScript code. Adobe defined a basic set of functions [2], but PDF applications often choose to implement a subset of Adobe's standard as well as to extend their feature set with proprietary functions (see section VIII). JavaScript provides a huge flexibility for documents, for example, complex input validation of forms or changing their values depending on specific conditions.

```
1 0 obj                                                    1
<< /Type /Catalog          % the first processed PDF object  2
   /OpenAction <<      % action executed after opening file  3
   /Type /Action                    % definition of the action  4
   /S /JavaScript               % this is a Javascript action  5
   /JS (JavaScript code) >>                 % JavaScript code  6
>>                                                         7
endobj                                                     8
```

Listing 2. PDF document executing JavaScript after opening (excerpt).

In Listing 2, an example of a PDF action containing JavaScript is shown. The document *Catalog*, which is the first processed object in a PDF document, contains the entry *OpenAction*. The *OpenAction* event defines an action which is executed directly after the document is being opened. In the given example, the JavaScript code defined in Line 6 will be executed.

D. PDF File Handles

A file handle (or PDF *File Specification*) is a multi-purpose object that can be either an embedded file (i.e., a data stream within the document), a local file on disk, a remote URL, or a network share, depending on given parameters and context. File handles define the targets of many PDF actions such as where to submit form data to (via *SubmitForm* action) or which hyperlinks to follow in a document (via *URI* or *GoToR* action).

III. Related Work

PDF documents have been considered as relatively secure against malware and other security threats until 2001 [56], when the *Peachy* virus misused PDF features to run malicious VBScript [57]. In the following years, PDF malware grew to an importance, mostly based on implementation bugs in viewer applications [58, 52]. During this period, PDF malware focused mainly on abusing JavaScript. To estimate the importance of JavaScript-based vulnerabilities in PDF documents we filtered the CVE database for entries relating to 28 PDF processing applications. Since 2003, there are 1325 relevant CVE IDs, of which 73 lead to code execution – the rest being DoS, data leakage, or other vulnerabilities. Of all PDF-related CVE IDs, 138 entries are due to JavaScript.[2] Laskov et al. [34] outline two classes of JavaScript PDF exploits: either the JavaScript

[1] Our test suite of PDF documents can be found at https://pdf-insecurity.org/download/pdf-dangerous-paths/exploits-and-helper-scripts.zip.

[2] The total number of JavaScript related issues may be higher because JavaScript engine bugs usually do not get separate CVE IDs for integrators.

API is targeted directly or the API is abused to target other software components.

In 2008, Filiol et al. analyzed for the first time malicious PDF features beyond JavaScript. Their work was extended in the following years by multiple researches which found new methods to carry out DoS, URI invocation, code execution, and information leakage using PDF files [48, 16, 49, 63, 51, 31, 32]. Even though, the security impact of specific attack variants based on insecure PDF features was understood and fixed in many implementations, new variants were reported in 2018 [24, 30, 50]. In contrast to our work, previous research on insecure features of PDF documents focused on single features, and mainly on single applications such as Acrobat Reader and Foxit Reader, and was not driven by a systematic approach.

To prevent harm, different security tools were proposed, in order to identify maliciously crafted documents [34, 37, 53, 18, 38, 55, 15]. Such tools rely on the detection of known attack patterns and on a structural analysis of PDF files. In 2017, Tong et al. introduced a concept for PDF malware detection based on machine learning and its implementation [62, 61]. Maiorca et al. provided an overview of current PDF malware techniques and compared existing security tools [36]. In our research, we focus on the security of the PDF viewers and not on additional protection tools. Thus, we do not evaluate whether third party tools are able to detect our attacks.

While studying the related work on PDF security, we determined two gaps which we address in this paper. First, there is no systematic approach on how to find attacks based on insecure PDF features since all relevant work, which is widespread in multiple scientific papers, technical reports, and blogposts, focuses on single features or attack variants. Second, there is no comprehensive evaluation of a large set of popular PDF viewers, beyond Acrobat Reader and Foxit Reader.

IV. Attacker Model

In this section, we describe the attacker model, including the attacker's capabilities and the winning condition.

A. Actions of the Victim

The *victim* is an individual who retrieves and opens a malicious PDF document from an attacker controlled source. This is a realistic attack scenario, because even sophisticated users download and open PDF files from untrusted sources such as email attachments or the Internet. For example, invoices or academic papers are usually shared as PDF documents. PDF is often considered as relatively "safe" by end-users [14], compared to other file formats such as Word documents, which are well-known to contain potentially dangerous macros [25].

To open the PDF document, the victim uses a pre-installed application which processes the file in order to display its content. Different applications may process the file, or interpret features of the PDF standard, differently, thereby enabling or disabling the various attack vectors described in this paper.

B. Attacker's Capabilities

The *attacker* can create a new PDF file or modify an existing document which we denote as the *malicious document*. We do not require the malicious document to be compliant to the PDF specification, although the attacker targets basic functionality and features of the PDF standard. The attacker has full control over the document structure and its content. While the attacker can easily craft a malicious document which *looks* benign once opened and interpreted by the PDF application (i.e., similar to a document that the victim would expect), this is not assumed to be necessary, because all attacks are automatically triggered once the file is opened. The only interaction of the victim is to open the malicious document on their computer.

C. Winning Condition

An attack is classified as *successful* if its winning condition is fulfilled. The winning condition – the goal of the attacker – is dependent on the attack class and documented in the corresponding section. For example, in the DoS attack class, the winning condition is reached if the PDF processing application can be forced to consume all available resources (i.e., memory or CPU time). In the information disclosure class of attacks, the winning condition is fulfilled if the attacker manages to obtain sensitive data, such as local files from the victim's disk.

V. Methodology

To identify attack vectors, we systematically surveyed which potentially dangerous features exist in the PDF specification. We started by creating a comprehensive survey with all PDF *Actions* that can be called. As a base, we used the list provided in the PDF specification, see [60, section 8.5.3]. This list contains 18 different actions which we carefully studied. We selected eight actions (see *Call Action* in Figure 1) – the ones that directly or indirectly allow access to a file handle (see *File* in Figure 1) and may therefore be abused for dangerous features such URL invocation or writing to files.

Having a list of security sensitive actions, we proceeded by investigating all objects and related events which can trigger these actions. This process was the most time-consuming part of our investigation since the entire specification was analyzed.

We identified four PDF objects which allow to call arbitrary actions (*Page*, *Annotation*, *Field*, and *Catalog*), as shown in the upper part of Figure 1. For calling them, most objects offer multiple alternatives. The *Catalog* object, for example, defines the *OpenAction* or additional actions (*AA*) as events. Each event can launch any sequence of PDF actions, which are depicted in the middle part in Figure 1 (*Launch*, *Thread*, etc.). In addition, JavaScript actions can be embedded within documents, opening a new area for attacks. By using JavaScript, for example, new annotations can be created, which can have actions that once again lead to accessing file handles.

If a path from an event over an action to file handle[3] exists and is not explicitly blocked by the application opening the document, we denote it as a "dangerous path", resulting, for example, in file system access or URL invocation. Our approach is *comprehensive* in the sense that *all* attacks based on such dangerous paths are covered, because *all* existing paths in the PDF specification down to a file handle are mapped. Another kind of dangerous path arises, when the specification enables objects to create reference circles, resulting in *infinite*

[3] File handles can be embedded files, local files, URLs, or network shares.

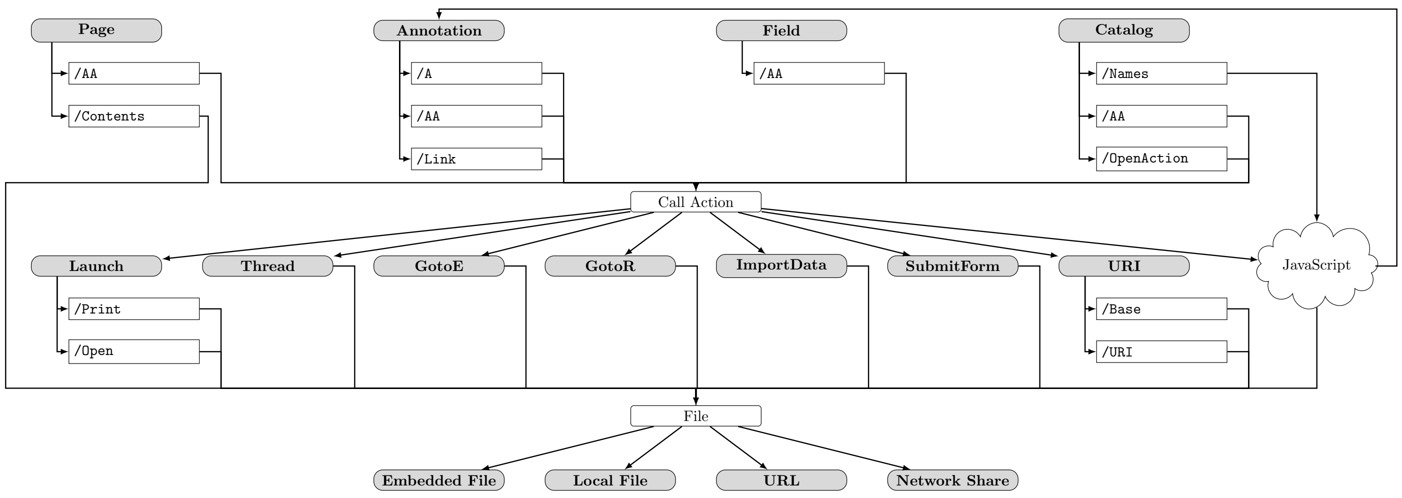

Fig. 1. **Dangerous paths identified by studying the PDF specification (simplified).** There are different special PDF objects (*Catalog*, *Page*, ...) defined that allow to call various actions (*Launch*, *Thread*, ...) which can read from or write to a PDF *File Specification*.

loops. Further discovered attacks – *deflate bombs* and *content masking* – are based on flaws on the document structure level, which we observed during our study of the specification.

Finally, we systematized our results, created a list of all possible attacks, and classified them accordingly. To generate our test suite of malicious PDF documents, we chose a semi-automated approach: we hand-crafted the payloads to test for a particular weakness and wrote a set of helper tools in Python, to generate a broad set of attack variants as well as a valid PDF structure for each test case. To improve the impact of the attacks, we also build exploits by chaining multiple actions. For example, an attacker can craft a document that first reads data from a local file using the *Import* action and then sends the content to the attacker's server using the *SubmitForm* action.

Our efforts resulted in 209 unique PDF files[4], which we manually opened in 28 PDF applications to observe the result. This process can be automated by launching each test for each PDF viewer in a batch script and logging the program's behavior, depending on the attack class (e.g., CPU or memory exhaustion for DoS, file exists checks for *file write access* attacks, etc.).

VI. Attacks

Out this section, we introduce the attacks that we elaborated during our security analysis. The "dangerous path" is given at the end of each attach description.

Clarification of Novelty: Of course, this work is not the first research on PDF security (see section III). However, we are the first covering the entire specification for attacks based on the *dangerous path*. While variants of some attacks have been presented before, our work goes far beyond systematizing existing results. It provides many new insights as well as novel attacks. The novelty level for each attack is given below.

- Well known attacks: code execution via *Launch action*[5]
- Novel attack variants:[6] *infinite loop*, *deflate bomb*, *URL invocation*, *credential theft*, *content masking*.
- Previously unknown attacks: *form data leakage*, *local file leakage*, *form modification*, *file write access*

Previous work relevant to a specific attack is provided in each corresponding attack section.

A. Denial-of-Service

The goal of this class of attacks is to build a specially crafted PDF document which enforces processing applications to consume all available resources (i.e., computing time or memory) or causes them to crash[7]. Note that while the impact of DoS is limited for end-users, it can lead to severe business impairment if the document is processed on a server, for example, by a library that generates preview thumbnails of PDF files uploaded to cloud storage.

1) Infinite Loop: Inducing an endless loop causes the program execution to get stuck. The PDF standard allows various elements of the document structure to reference to themselves, or to other elements of the same type. This can lead to cycles, if not explicitly handled by the implementation. For example, a *Pages* object may reference to other pages, which is a known problem of the specification, discovered in CVE-2007-0104. We systematically studied the PDF standard for further constructs that allow for reference cycles, recursion, or other kinds of loops, and found the following novel variants:

- *Action loop.* PDF actions allow to specify a *Next* action to be performed, thereby resulting in "action cycles".
- *ObjStm loop.* Object streams may extend other object streams allows the crafting of a document with cycles.
- *Outline loop.* PDF documents may contain an outline. Its entries, however, can refer to themselves or each other.

[4] Note that we combined multiple triggering events into single PDF files, thereby testing various paths in parallel and reducing the overall number of required test documents.

[5] Note that even though the danger of the PDF *Launch* action is well known in the sense that it has been publicly documented, security gaps still exist in multiple implementations, as confirmed by our evaluation (see section VII).

[6] Only a small number of variants was known because previous research did not systematically investigate the PDF specification or test all possible paths.

[7] Crashes are classified as a winning condition, because they affect the user experience, especially if further, legitimate documents are already opened by the same PDF application (in multiple tabs/windows) and if there are unsaved changes, resulting in data loss. Furthermore, crashes have lead to code overflow vulnerabilities in the past, which have been classified as critical by Adobe [19].

- *Calculations.* PDF defines “Type 4” calculator functions, for example, to transform colors. Processing hard-to-solve mathematical formulas may lead to high demands of CPU.
- *JavaScript.* Finally, in case the PDF application processes scripts within documents, infinite loops can be induced.

Dangerous paths (examples)

Action ⇒ /Next ⇒ Action
ObjStm ⇒ /Extends ⇒ ObjStm

2) Deflate Bomb: Data amplification attacks based on malicious zip archives are well-known (see [12, 22, 45]). The first publicly documented DoS attack using a “zip bomb” was conducted in 1996 against a Fidonet BBS administrator [1]. However, not only zip files but also stream objects within PDF documents can be compressed using various algorithms such as *Deflate* [20] to reduce the overall file size. The question arises if compression bombs based on malicious PDF documents can be built, in order to cause processing applications to allocate all available memory. We intend to achieve this goal by a chaining a compressed stream to one or multiple `FlateDecode` filters.

Dangerous path

Filter ⇒ /FlateDecode ⇒ [...] ⇒ /FlateDecode

B. *Information Disclosure*

The goal of this class of attacks is to track the usage of a document by silently invoking a connection to the attacker’s server once the file is opened, or to leak PDF document form data, local files, or NTLM credentials to the attacker.

1) URL Invocation: Tracking pixels in HTML emails are well documented,[8] but the existence of similar technologies for PDF files is largely unknown to the general public. However, PDF documents that silently “phone home” should be considered as privacy-invasive. They can be used, for example, to deanonymize reviewers, journalists, or activists behind a shared mailbox. The goal of this attack is to open a backchannel to an attacker controlled server once the PDF file is opened by the victim. Besides learning *when* the file was opened and by *whom* (i.e., by which IP address), the attacker may learn additional (limited) information such as the victim’s PDF viewer application and operating system, derived from the *User-Agent* HTTP header. The possibility of malicious URI resolving in PDF documents has been introduced by Hamon [27] who gave an evaluation for *URI* and *SubmitForm* actions in Acrobat Reader. We extend their analysis to *all* standard PDF features that allow to open a URL, such as *ImportData*, *Launch*, *GoToR*, JavaScript, and to a to a broad set of viewers.

Dangerous path

[All events] ⇒ [All actions] ⇒ URL

2) Form Data Leakage: Documents can contain forms to be filled out by the user – a feature introduced with PDF version 1.2 in 1996 and used on a daily basis for routine offices tasks, such as travel authorization or vacation requests. Depending on the nature of the form, user input can certainly contain sensitive information (e.g., financial or medical records). Therefore, the question arises if an attacker can access and leak such information. The idea of this attack is as follows: the victim downloads a form – a PDF document which contains form fields – from an attacker controlled source and fills it out on screen, for example, in order to print it. Note that there are legitimate cases where a form is obtained from a third party, while the user input should not be revealed to this party. For example, European SEPA remittance slips can be downloaded from all over the web[9] – even though they have to be manually signed to be accepted by a local bank. The form is manipulated by the attacker in such a way that it silently, without the user noticing, sends input data to the attacker’s server. To the best of our knowledge, we are the first to demonstrate such attacks, which can be carried out using the PDF *SubmitForm* action, or by reading and exfiltrating the form values using standard JavaScript functions.

Dangerous path

Page ⇒ (on close) ⇒ SubmitForm ⇒ URL

3) Local File Leakage: The PDF standard defines various methods to embed external files into a document or otherwise access files on the host’s file system, as documented below.

- *External streams.* Documents can contain stream objects (e.g., images) to be included from external files on disk.
- *Reference XObjects.* This features allows a document to import content from another (external) PDF document.
- *Open Prepress Interface.* Before printing a document, local files can be defined as low-resolution placeholders.
- *Forms Data Format (FDF).* Interactive form data can be stored in, and auto-imported from, external FDF files.
- *JavaScript functions.* The Adobe JavaScript reference enables documents to read data from or import local files.

If a malicious document managed to firstly read files from the victim’s disk and secondly, send them back to the attacker,[10] such behavior would arguably be critical. However, standard PDF functions can be chained together to achieve exactly this. For example, form values can be references to stream objects and every stream, on its part, can reference to an external file. Moreover, forms can be crafted to auto-submit themselves using various events as documented in Figure 1 in section IX. Furthermore, standard JavaScript functions can be used to access local files and leak their content. We give a systematic overview on this new chaining technique in terms of a directed graph containing all chains detected during our evaluation, and are the first to demonstrate these attacks.

Dangerous path

[All events] ⇒ ImportData ⇒ local file
⇒ /Next ⇒ SubmitForm ⇒ URL

4) Credential Theft: In 1997, Aaron Spangler posted a vulnerability in Windows NT on the Bugtraq mailing list [54]: any client program can trigger a connection to a rogue

[8] A recent study of Poddebniak et al. [47] revealed backchannels in 40 out of 48 tested email clients.

[9] E.g., https://www.ibancalculator.com/fileadmin/EU-Ueberweisung.pdf.

[10] Note that exfiltration does not necessarily have to occur via the network: For example, if a cloud storage service generates thumbnail images from uploaded PDF documents, the backchannel can be the rendered image itself. If a reviewer adds comments to a malicious PDF document, local files may unintentionally be included when saving, exporting or printing the document.

SMB server. If the server requests authentication, Windows will automatically try to log in with a hash of the user's credentials. Such captured NTLM hashes allow for efficient offline cracking[11] and can be re-used by applying pass-the-hash or relay attacks [29, 43] to authenticate under the user's identity. This design flaw in the Windows operating system is not solved until today.[12] Back in 1997, Spangler used a remote image to trick web browsers into making a connection to and thereby authenticate to the attacker's host. In April 2018, Check Point Research [50] showed that a similar attacks can be performed with malicious PDF files. They found that the target of *GoToR* and *GoToE* actions can be set to `\\\\attacker.com\\dummyfile`,[13] thereby leaking credentials in the form of NTLM hashes. The issue was fixed quickly by Adobe and Foxit. We describe novel variants of this attack, for example, by using various other techniques to access a network share such as by including it as external content stream or by testing different PDF actions, thereby bypassing existing protection mechanisms.

Dangerous path

[All events] ⇒ [All actions] ⇒ network share

C. Data Manipulation

This attack class deals with the capabilities of malicious documents to silently modify form data, to write to local files on the host's file system, or to show a different content based on the application that is used to open the document.

1) Form Modification: The idea of this attack is as follows: similar to "form data leakage" as described above, the victim obtains a harmlessly looking PDF document from an attacker controlled source, for example, a remittance slip or a tax form. The goal of the attacker is to dynamically, and without knowledge of the victim, manipulate form field data. This can be achieved by crafting the malicious document in such a way that it "modifies itself", and changes certain form fields immediately before it is printed or saved. Interesting form fields to manipulate could be, for example, the recipient of a wire transfer or the declarations regarding taxable income. Technically, form field values can be set using an *ImportData* action which imports form data from an external source or an embedded file, or with JavaScript included in the document. This novel attack technique can be used by an attacker to either get the victim into trouble (e.g., tax fraud suspicion) or to gain financial advantages (e.g., by adding herself as recipient of a tax refund).

Dangerous path

Catalog ⇒ (on print) ⇒ ImportData ⇒ embedded file

2) File Write Access: As previously described, the PDF standard enables documents to submit form data to external webservers. However, technically the webserver's URL is defined using a PDF *File Specification*. This ambiguity in the standard may be interpreted by implementations in such a way that they enable documents to submit PDF form data to a local file, thereby writing to this file. Furthermore, there are various JavaScript functions which allow to write to local files on disk. If successful, this feature can be used to overwrite arbitrary files on the victim's file system and thereby purge their content. Furthermore, write access to local files may even be escalated to code execution if the attacker has write access to certain startup scripts (e.g., *autoexec.bat* on Windows, *.bashrc* on macOS and Linux). JavaScript based attacks to write to local files have previously been shown, for example, in CVE-2018-14280 and CVE-2018-14281 for Foxit Reader. We evaluate write access for a broad range of standard PDF and JavaScript functions. To the best of our knowledge, we are the first to propose the attack variant based on PDF forms that automatically submit data to a local file.

Dangerous path

[All events] ⇒ SubmitForm ⇒ local file

3) Content Masking: The goal of this attack is to craft a document that renders differently, depending on the applied PDF interpreter. This can be used, for example, to show different content to different reviewers, to trick content filters (AI-based machines as well as human content moderators), plagiarism detection software, or search engines, which index a different text than the one shown to users when opening the document. Content masking attacks using polyglot files have been shown in the past by [35, 10]; for example, PDF files that are also a valid JPEG images, if opened by image processing software. Recently, [39] presented "PDF mirage", which applies font encoding to present a different *displayed* content to humans than to text exfiltration software. We propose a new approach which targets edge cases in the PDF specification, leading to different parts of the document actually being *processed* by different implementations. To achieve this, we systematically studied the PDF standard for ambiguities at the syntax and structural level, as documented below.

- *Stream confusion.* It is unclear how content streams are parsed if their *Length* value does not match the offset of the *endstream* marker, or if syntax errors are introduced.
- *Object confusion.* An object can overlay another object. The second object may not be processed if it has a duplicate object number, if it is not listed in the *XRef* table, or if other structural syntax errors are introduced.
- *Document confusion.* A PDF file can contain yet another document (e.g., as embedded file), multiple *XRef* tables, etc., which results in ambiguities on the structural level.
- *PDF confusion.* Objects before the PDF header or after an *EOF* marker may be processed by implementations, introducing ambiguities in the outer document structure.

There are numerous variants of the four test classes mentioned above, resulting in a total of 94 different edge cases.

Dangerous path

None (document structure level flaws)

[11] For NTLMv2, it is estimated that cracking eight character passwords of any complexity takes around 2,5 hrs on a modern GPU [17]. Previous versions (NTLMv1, LM) are trivial to crack and can be considered as broken [40].

[12] Microsoft introduced the possibility to define "NTLM blocking" in the Windows security policy, but is has to be actively enabled by administrators. Furthermore, some ISPs block port 445, however this cannot be relied on.

[13] Note that the \ character must be escaped in PDF strings, leading to \\.

D. Code Execution: Launch Action

The goal of this attack is to execute attacker controlled code. This can be achieved by silently launching an executable file, embedded within the document, to infect the host with malware.[14]

The PDF specification defines the *Launch* action, which allows documents to launch arbitrary applications. The file to be launched can either be specified by a local path, a network share, a URL, or a file embedded within the PDF document itself. The standard does not provide any security considerations regarding this obviously dangerous feature; it even specifies how to pass command line parameters to the launched application. Therefore, it can be said that PDF offers "command execution by design" – if the standard is implemented in a straightforward manner. An example of a malicious document which contains an embedded executable file (*evil.exe*) that is launched once the document is opened (*OpenAction*) is depicted in Listing 3.

```
1 0 obj
<< /Type /Catalog /Names <<
   /EmbeddedFiles << /Names [(evil.exe) 2 0 R] >> >>
   /OpenAction << /S /Launch /F (evil.exe) >>
>>
endobj

2 0 obj
<< /Type /EmbeddedFile /Length 1337 >>
stream
[executable code]
endstream
```

Listing 3. PDF document to launch an embedded executable.

The danger of *Launch* actions is well-known and has first been discussed in 2008 by Blonce et al. [13] for Acrobat Reader. Modern PDF viewers should warn the user before executing potentially malicious files – or stop supporting this insecure feature at all. We extend the analysis of Blonce et al. to a broad set of 28 modern PDF implementations and to all potentially dangerous paths and thereby show that attack variants leading to code execution are possible until today.

Dangerous path

[All events] ⇒ Launch ⇒ embedded/local file or URL

VII. Evaluation

A. Denial-of-Service

In the following section, we discuss the results for DoS attacks. Due to the large number of test cases, a fully detailed evaluation is given in Table VI in the appendix. We classify an application as vulnerable if it either hangs (e.g., consuming unusually large amounts of CPU or memory) or if the program crashes. A controlled program termination (i.e., raising an exception before closing) is not considered as a vulnerability.

To evaluate the attacks introduced in section VI, we tested them on 28 popular PDF processing applications that were assembled from public software directories for the major platforms (Windows, Linux, macOS, and Web).[15] In addition to native PDF viewers, we evaluated the most popular web browsers because modern browsers have the ability to directly render PDF files (e.g., from a website). If a "viewer" and an "editor" version was available we tested both. All applications were tested in the default settings, neither relaxing nor hardening their security policies. We only classified attacks as successful, if they did *not* require any kind of user interaction besides opening the malicious document. For example, PDF applications which present a confirmation dialog before performing a certain attack were labeled as *not vulnerable*. Evaluation results are depicted in Table I.

Obviously, the criticalness of each attack differs. For example, the impact of code execution based on a malicious document is much higher than DoS. As one can see, PDF applications for macOS and Linux, which implement only a subset of PDF standard features, can be considered as relatively secure. This also holds for web browsers, which apply additional sandboxing mechanisms (e.g., to prevent file system access).

B. Denial-of-Service

In the following section, we discuss the results for DoS attacks. Due to the large number of test cases, a fully detailed evaluation is given in Table VI in the appendix. We classify an application as vulnerable if it either hangs (e.g., consuming unusually large amounts of CPU or memory) or if the program crashes. A controlled program termination (i.e., raising an exception before closing) is not considered as a vulnerability.

1) Infinite Loop: Each of the tested applications running natively on Windows, macOS, or Linux, except PDF Studio Viewer/Pro and Evince, was vulnerable to at least one attack variant and could be tricked into an endless loop. It is noteworthy that CVE-2007-0104 still works in six applications until today. Our novel attack variants, such as *GoTo* loops (9 vulnerable), *Action* loops (9 vulnerable), *Outline* loops (9 vulnerable) and JavaScript (13 vulnerable) cause endless loops in various PDF interpreters. The impact is either a crash of the program, or the application becoming completely unresponsive, often combined with a high consumption of CPU time. Browser based PDF viewers instead perform much better. We observed that for Chrome, Firefox, and Opera only the current tab gets stuck in an endless loop and becomes unresponsive, which is why we classified the vulnerability as "limited" here. We assume this is because modern browsers sandbox each tab and enforce resource limits, thereby restricting the impact of, for example, a malicious or runaway website.

2) Deflate Bomb: To evaluate the impact of compression bombs, we crafted a valid PDF file containing a long string of 10 GB of repeated characters, "AAA...", within a *Deflate* compressed content stream. To display this string to the user, a PDF processing application must first decompress it. The maximum compression ratio that can be achieved with the *Deflate* algorithm is 1023:1. However, the PDF file size can be drastically reduced by applying multiple *Deflate* filters to the same stream, resulting in an amplification factor of 18 470 265 (i.e., 578 bytes on disk are decompressed to 10 GB in memory). The attack resulted in memory exhaustion in 20 applications, of which three applications crashed after a short period of time. In various cases, the operating system slowed down noticeably or became completely unresponsive. In contrast to attacks

[14] Note that there are other methods to gain code execution (e.g., based on memory corruption); however, they are out of scope in this paper. Our focus is on abusing of legitimate features, not bugs in PDF viewer implementations.

[15] Note that some PDF applications are available for multiple platforms. In such cases we limited our tests to the platform with the highest market share.

Attack Category			DoS		Information Disclosure				Data Manipulation			RCE
Application	Version		Infinite loop	Deflate bomb	URL invocation	Form data leakage	Local file leakage	Credential theft	Form modification	File write access	Content masking	Code execution
Acrobat Reader	(2019.012.20035)	Windows	●	●	●	○	○	○	●	○	○	○
Foxit Reader	(9.7.1)		●	●	○	○	○	●	○	○	○	○
PDF-XChange Viewer	(2.5.322.9)		●	●	●	●	●	●	●	○	●	○
Perfect PDF Reader	(8.0.3.5)		●	●	●	●	○	●	●	○	○	○
PDF Studio Viewer	(2018.4.3)		○	●	●	●	◐	○	○	○	○	●
Nitro Reader	(5.5.9.2)		●	●	●	●	○	●	○	○	●	●
Acrobat Pro	(2019.012.20035)		●	●	●	○	○	○	●	○	○	○
Foxit PhantomPDF	(9.7.1)		●	●	○	○	○	●	○	○	○	○
PDF-XChange Editor	7.0.326.1		●	●	●	●	●	●	●	●	○	○
Perfect PDF Premium	(10.0.0.1)		●	●	●	●	○	●	●	○	○	○
PDF Studio Pro	(2018.4.3)		○	●	●	●	◐	○	○	○	○	●
Nitro Pro	(13.24.1.467)		●	○	●	●	○	●	○	○	●	●
Nuance Power PDF	(3.0.0.17)		●	●	●	○	●	●	○	○	●	●
iSkysoft PDF Editor	(6.5.0.3929)		●	○	○	○	○	○	○	○	○	○
Master PDF Editor	(5.1.36)		●	○	●	●	◐	●	○	●	○	○
Soda PDF Desktop	(11.0.16.2797)		●	●	●	○	○	●	○	○	●	●
PDF Architect	(7.0.30.3196)		●	●	●	○	○	●	○	○	●	○
PDFelement	(6.8.0.3523)		●	○	○	○	○	○	○	○	○	◐
Preview	(10.0.944.4)	Mac	●	●	○	○	○	○	○	○	○	○
Skim	(1.4.41)		●	●	○	○	○	○	○	○	○	○
Evince	(3.34.1)	Linux	○	●	○	○	○	○	○	○	○	◐
Okular	(1.3.2)		●	●	○	○	○	○	○	○	○	◐
MuPDF	(1.16.0)		●	○	○	○	○	○	○	○	○	◐
Chrome	(70.0.3538.77)	Web	◐	●	●	●	○	○	○	○	○	○
Firefox	(72.0.2)		◐	●	●	○	○	○	○	○	○	○
Safari	(13.1.2)		○	○	○	○	○	○	○	○	○	○
Opera	(57.0.3098.106)		◐	◐	●	●	○	○	○	○	○	○
Edge	(44.18362.1.0)		○	○	○	○	○	○	○	○	○	○

● Application vulnerable ◐ Vulnerability limited ○ Not vulnerable

TABLE I. Evaluation results: Out of 28 tested PDF applications, 26 are vulnerable to at least one attack.

based on infinite loops, even browsers such as Chrome and Firefox were fully affected, while in Opera only the current tab became unresponsive. The remaining seven PDF applications did refuse to decompress the whole stream, but instead aborted decompression after a reasonable amount of time – probably after a watchdog limit was reached.

It is noteworthy that we did not even have to actually *open* the malicious document on Windows and Linux in order to cause DoS to the operating system. Both Windows File Explorer and Gnome Nautilus file manager try to preview the document if the containing directory is opened, and thereby process its content resulting in resource exhaustion. MacOS (Finder) was not vulnerable, because it stopped thumbnail generation, probably after a resource limit was hit.

Although DoS attacks against web servers were not tested for ethical reasons, applications processing PDF files on the server-side are likely to be affected too. For example, Evince and Okular, which are both vulnerable, are based on Poppler,[16] a popular PDF library used by various cloud storage providers and file-hosting solutions such as Seafile[17] in order to generate preview images of uploaded PDF documents.

C. Information Disclosure

1) URL Invocation: To evaluate if malicious documents can enforce PDF applications to trigger a connection to an attacker controlled server, we combined various PDF features with techniques to automatically call them once the document was opened. The results for auto-triggered PDF actions resulting in URL invocation are as follows: *URI* action (9 vulnerable), *GoToR* (1 vulnerable), *Launch* (6 vulnerable), and *SubmitForm* (11 vulnerable). For seven applications, we could use standard JavaScript functions to invoke a connection. In one viewer, we could set a URL as the external content stream of an image, which was loaded from the attacker's server. In two viewers, we were able to inject a subset of XHTML, leading to XHTML tags being being processed which triggered a remote connection. Altogether, 17 PDF applications could be tricked into (silently) invoking a connection to our server, once a malicious document was opened by the user. It can be

[16] See https://poppler.freedesktop.org/.

[17] See https://www.seafile.com/.

concluded that it is relatively easy to craft a PDF document which reports back to the author (or a third party) when the document is opened, in a majority of the tested applications.

Note that for vulnerable PDF interpreters in web browsers this issue can lead to further, web-security weaknesses. For example, a malicious document uploaded by the attacker to a social media website can trigger same-site requests[18] if viewed by the victim. This would otherwise be forbidden by the browser and may be exploited to perform actions in the context of the user's account, in case same-site cookies [64] are used by the web application to protect against cross-site request forgery (CSRF).

2) Form Data Leakage: To test if form data can be leaked silently, without the user knowing, we modified the standard U.S. individual tax return form 1040[19] to send all user input to our webserver once the document is either printed or closed. This can be done by combining the *DP* ("did print") and *PC* ("page closed") events of the *Catalog* and *Annotation* objects with a *SubmitForm* action or JavaScript. We classify the attack as successful if a PDF application passes filled-in form data without the user being made aware of it (i.e., no warning message or confirmation dialog displayed). Nine applications are vulnerable to this attack, using forms that auto-submit themselves. For two additional applications, we were able to use JavaScript to access form data and silently exfiltrate it to our server. Nine applications did ask the user before sending the data, which we consider as sane behavior. Another eight PDF interpreters (e.g., on macOS and Linux) did not support the feature of submitting PDF form data at all.

3) Local File Leakage: Although part of the standard, only two applications (i.e., PDF-XChange Editor and Nuance Power PDF) support the feature of external streams. For both applications, we were able to craft a document which embeds arbitrary files on disk into the document and silently leaks them to an external server using both auto-submitting forms and JavaScript. Exfiltration happens in the background once the document is opened, without the user noticing and without any visible changes to the document. For another three applications, we were able to include and automatically leak the contents of FDF files and XML-based XFDF files (using the *ImportData* action or the *ImportFDF* JavaScript function). We classify this vulnerability as limited, because it is restricted by file type – yet it should be clear that such behavior is not desired either. Note that this attack is different from "form data leakage" as mentioned before, because although (X)FDF files usually contain PDF form data, this attack results in the contents of *external* (X)FDF files from disk being leaked, which is may be completely unrelated to the form data of the currently opened (malicious) document. For PDF-XChange Viewer, we were additionally able to use standard JavaScript functions to access arbitrary files and the leak them.

4) Credential Theft: We installed Responder[20] as a rogue authentication server to obtain the client's NTLM hashes when opening the malicious document. We were able to leak the hashes of NTLM credentials to our server without the user noticing or being asked for confirmation to open a connection to the rogue network shared drive on 12 out of the 18 Windows based PDF viewers. Using hashcat,[21] we could perform successful brute force attacks on the hashes of simple 5-character passwords within within seconds.[22] Note that, by design, only applications running on Windows are affected. We used a mixture of techniques to accomplish this goal: external streams, standard PDF actions, as well as JavaScript. Various readers were affected by multiple test cases. It is interesting to note that, although Foxit fixed this issue in 2018 for PhantomPDF/Reader, we could identity bypasses using four different techniques. This is because – apparently – accessing a share invocation via *GoToR* actions (as documented in the original exploit) was prohibited, however, using other action types, such as auto-printing a file on a network shared drive, we were again able to enforce NTLM hashes being leaked.

D. Data Manipulation

1) Form Modification: To test the feasibility of crafting PDF documents that silently manipulate their own form data, we once again modified the U.S. tax return form 1040. We added an *ImportData* action that changes the refund account number to the attacker's account number once the document is printed.[23] We used the *WP* ("will print") event for this purpose. Unfortunately, from an attacker's point of view, none of the tested applications supports importing form data from an *embedded file* within the document itself – or from an external URL. By using standard PDF JavaScript functions (`getAnnots()[i].contents`), we were however able to modify PDF form data in six applications. JavaScript also allowed us to temporarily store the original user data and undo our manipulation immediately after the document had been printed, using the *DP* ("did print") event, and to enforce that these modifications are only performed until a certain date, thereby making it more difficult to reproduce the manipulation.

2) File Write Access: Only three applications allowed to submit form data to a local file. While Foxit PhantomPDF and Foxit Reader explicitly ask the user before writing to disk, Master PDF Editor silently writes to or overwrites arbitrary files with attacker controlled content by auto-submitting the form data to a PDF *File Specification*. We also tested six standard PDF JavaScript functions to write to disk. The `extractPages()` function allowed us to write data to arbitrary locations on disk in PDF-XChange Editor. The other applications did not support writing files with JavaScript at all, asked the user for confirmation, or showed a "Save as" dialog, instead of automatically writing the file to a given location.

3) Content Masking: We define an application as vulnerable if we can create a document that displays certain text in this, and only in this, application, while a completely different text is displayed in all other tested PDF viewers – with the exception of two applications utilizing the same underlying PDF interpreter (e.g., Evince/Okular are both based on Poppler). Furthermore, if a vendor produces a "viewer" and an "editor" version of an application, both may also

[18] HTTP `POST` requests in Chrome and Opera, `GET` requests in Firefox.

[19] Available for download from https://www.irs.gov/pub/irs-pdf/f1040.pdf.

[20] See https://github.com/SpiderLabs/Responder.

[21] See https://hashcat.net/hashcat/.

[22] Of course, it is up to the configuration of the victim's setup (e.g., password strength and security policy) if efficient cracking attacks are actually feasible.

[23] It must be noted that, in practice, this attack does not only have a technical component. It will only work if the attacker's bank accepts the deposit, see https://www.irs.gov/faqs/irs-procedures/refund-inquiries/refund-inquiries-18.

display the same text. Of our 94 hand-crafted edge cases, 63 rendered differently when opened in different applications. Full details are given in Table VII in the appendix. For three PDF interpreter engines (six applications), we found a case where certain text was displayed *only* in this interpreter. For other PDF interpreters, we could not find edge cases that resulted in a *unique* appearance (i.e., no other interpreter displaying the same text), therefore we did not classify them as vulnerable. It must, however, be noted that test cases can potentially be chained together, which may result in getting more applications to render unique content. This challenge is considered as future work. Another interesting use of this technique would be fingerprinting PDF interpreters applied in web applications to process or preview documents based on the rendered result of PDF file uploads.

E. Code Execution: Launch Action

In theory, by chaining PDF standard features, an attacker can easily get code execution "by design". We combined a *LaunchAction* with an *OpenAction* event to achieve this goal and launch an executable file. Surprisingly, this worked out of the box on six applications. The *.exe* file was launched without any confirmation dialog being displayed. The other tested applications asked the user for confirmation (5 viewers) before executing the file, denied to launch executable files (Acrobat Reader/Pro),[24] or did not support the *LaunchAction* at all in the default settings (11 viewers). Three Linux based viewers (Evince, Okular, and MuPDF) use *xdg-open*[25] to handle the file to be launched, thereby delegating the security decision to a third-party application. On our Debian GNU/Linux test system, this resulted in code execution with minimal user interaction; by referencing an *.exe* from a *Link* annotation, the file was executed with `/usr/bin/mono`, an emulator for .NET executables, if the user clicked *somewhere* into the document.[26] This was also a requirement for PDFelement. We classify these vulnerabilities as "limited" because – even though no confirmation dialog is presented to the user – the exploit is not *fully* automated.[27] PDF Architect 6, which we initially tested, was also vulnerable to code execution. However, version 7 had removed support for the *Launch* action. Finally, it must be said that, even if a confirmation dialog is presented, attackers may apply social engineering techniques to trick the victim into launching the file.

Because the *Launch* action can be considered as a dangerous feature, we conducted a large-scale evaluation of 294 586 PDF documents downloaded from the Internet[28], in order to research if there are any legitimate use cases at all. Of those documents, only 532 files (0.18%) contained a *Launch* action. While none of the files was classified as malicious according to the VirusTotal database,[29] we conclude that the *Launch* action is rarely used in the wild and its support should be removed by PDF implementations as well as the standard.

VIII. Additional Findings

In this section, we present additional insights related to JavaScript, Digital Rights Management, and hidden data in PDF documents.

A. JavaScript-based Fingerprinting

While the syntax of JavaScript code embedded in PDF documents is based on the ECMA standards [21], there is no specification of the Document Object Model (DOM) for PDF documents. Furthermore, the API provided by Adobe [2] is rather descriptive than prescriptive, i.e., lacking any form of IDL definitions. Thus, the objects and properties visible to JavaScript differ widely between different viewers. This results in embedded JavaScript engines of PDF viewers being easily fingerprinted via their provided functionality. As a simple proof of concept, we show that one can distinguish every JavaScript supporting PDF viewer already by recursively enumerating and counting the properties of the execution environment.

We show that the surface of the JavaScript API differs significantly between viewers. Using a crawler written in JavaScript we automated the enumeration of the API. The results, containing various details on all encountered properties, are extracted as JSON. Table II shows the number of properties grouped by their `type`. The greatly varying number of available functions highlights the disparity between implementation; this ranges from viewers only being capable of running loops and simple arithmetic without any further API (e.g., Evince), to viewers with only a handful of functions (e.g., PDF XChange Viewer: 114), to an almost complete coverage of the Adobe API (e.g., Acrobat Reader: 6742).
Additionally, many of the identified functions are not documented in the Adobe PDF JavaScript standard and do not yield any result on public search engines. The absence of public knowledge of these properties indicates that they are not intended to be used by authors of PDF documents. It is questionable whether these hidden APIs are well tested. We used the extracted JSON results as input for JavaScript code which simply calls every available function in the API with zero to four empty-string parameters. This already was enough to crash four PDF applications, thereby enabling DoS attacks.

Identifying the application is a useful preparation stage for attacks. It allows an attacker to send a first PDF document to the victim that replies back (e.g., using JavaScript APIs) which PDF viewer is used by the victim, and then exploit the vulnerabilities of this specific viewer by sending a second specially crafted attack PDF file to the victim.

B. Digital Rights Management

PDF documents can be "protected" based on questionable client-side security mechanisms. For example, the specification allows to restrict certain document capabilities, such as *printing*, *copying text*, or *editing content*. Technically, a special *permissions* object is added to the document which, according

[24] Note that Adobe products use a blacklist of potentially "dangerous" file extensions. However, various bypasses have been identified in the past [49].

[25] See https://www.freedesktop.org/wiki/Software/xdg-utils/.

[26] Readers may ask themselves: How often did I click in this document to jump to a certain section? Would I anticipate this can lead to code execution?

[27] Note that this is the only vulnerability described in this paper that requires a bit of user interaction and is not automatically triggered once the document is opened, because such events are not supported by Linux based readers.

[28] We obtained the dataset from the Cisco Umbrella 1 Million list of domains (see https://s3-us-west-1.amazonaws.com/umbrella-static/index.html). Instead of crawling each website directly for PDF documents, we searched the Internet Archive (see https://web.archive.org) for links to PDF files in each each domain and then retrieved all linked PDF documents from the original (live) website.

[29] See https://www.virustotal.com/.

Application		# functions	# objects	# numbers	# strings	# booleans
Acrobat Reader DC	Windows	6742	320	398	492	357
Foxit Reader		1900	130	79	146	30
PDF-XChange Viewer		114	58	68	183	1
Perfect PDF Reader[1]		💣	💣	💣	💣	💣
PDF Studio Viewer		💣	💣	💣	💣	💣
Nitro Reader		1067	159	55	84	10
Acrobat Pro DC		6851	714	388	482	358
Foxit PhantomPDF		1902	130	79	146	30
PDF-XChange Editor		3529	166	219	270	61
Perfect PDF Premium[1]		💣	💣	💣	💣	💣
PDF Studio Pro		💣	💣	💣	💣	💣
Nitro Pro		💣	💣	💣	💣	💣
Nuance Power PDF		206	88	109	730	0
iSkysoft PDF Editor		–	–	–	–	–
Master PDF Editor		1134	75	57	94	10
Soda PDF Desktop		2559	117	156	214	141
PDF Architect		2317	112	146	194	135
PDFelement		–	–	–	–	–
Preview	Mac	–	–	–	–	–
Skim		–	–	–	–	–
Evince	Linux	➲	➲	➲	➲	➲
Okular		➲	➲	➲	➲	➲
MuPDF		–	–	–	–	–
Chrome	Web	1183	73	46	87	21
Firefox		–	–	–	–	–
Safari		➲	➲	➲	➲	➲
Opera		1182	73	46	87	21
Edge		–	–	–	–	–

[1] JavaScript must be enabled in settings ➲ No feedback channel

– JavaScript support is not available 💣 Application crashes

TABLE II. JAVASCRIPT EXECUTION ENVIRONMENT DIFFERENCES.

		Access Permissions		
Application		Print	Copy	Edit
Acrobat Reader DC	Windows	○	○	–
Foxit Reader		○	○	–
PDF-XChange Viewer		○	○	–
Perfect PDF Reader		○	○	–
PDF Studio Viewer		○	○	–
Nitro Reader		○	○	–
Acrobat Pro DC		○	○	○
Foxit PhantomPDF		○	○	○
PDF-XChange Editor		○	○	○
Perfect PDF Premium		○	○	○
PDF Studio Pro		○	○	○
Nitro Pro		○	○	○
Nuance Power PDF		○	○	○
iSkysoft PDF Editor		○	○	○
Master PDF Editor		○	○	○
Soda PDF Desktop		○	○	○
PDF Architect		○	○	○
PDFelement		○	○	○
Preview	Mac	○	○	–
Skim		○	●	–
Evince	Linux	●	●	–
Okular		●	●	–
MuPDF		–	●	–
Chrome	Web	○	○	–
Firefox		●	●	–
Safari		●	○	–
Opera		●	●	–
Edge		○	○	–

● Permissions ignored ○ Permissions honored – Not available

TABLE III. ACCESS PERMISSION ENFORCEMENT IN PDF VIEWERS.

to the standard, *should* be respected by consumer applications. As it is completely up to the client application (i.e., the PDF viewer) to enforce PDF permissions, they cannot be considered as effective security mechanisms. In reality, various PDF applications, especially on Linux, do not interpret PDF permissions at all. To evaluate which viewers "conform to the standard" and enforce PDF access permissions, we saved a document using Adobe Acrobat Reader, with "printing", "copying text", and "editing" disabled. The results are given in Table III.

Of the tested 28 applications, five viewers completely ignore the user access permissions. For another two viewers, we could observe inconsistent behavior. For example, Safari allows to print the document but prohibits copying its text, in a document where both actions are prohibited.

C. Hidden Data in PDF Documents

In this section we discuss two privacy-related PDF issues – *evitable metadata* and *revision recovery* – which allow anyone obtaining the file to reveal potentially sensitive information.

1) Evitable Metadata in PDF Documents: In 2005, the former US President Bush gave a speech on the war in Iraq and published a strategy document on the White House website. The metadata of the PDF document revealed a Duke University political scientist as the original author of the document [33]. Afterwards, the NSA published best practices addressing risks involved with hidden data and metadata in PDF files [9]. This example shows that there are valid use-cases where the author of a document prefers to remain anonymous. The issue of unwanted metadata in various file formats is well-known and has been discussed in [11, 46]. Even though metadata is a feature of the PDF standard, from a privacy perspective, creator software should avoid to include excessive metadata by default and instead let users opt-in. Although many PDF documents are created with non-PDF software (e.g., LaTeX, office suites, or system printers), all professional PDF editors offer the creation of PDF files as well. They are especially used when designing complex PDF documents that, for example, include forms and JavaScript. During the creation process, these editors generate special PDF metadata objects, which can contain sensitive information (e.g., usernames or dates).

To identify which amount of information is included by modern applications, we created a minimal document with each PDF editor and identified the metadata in the saved file, which can either be found in the *Document Information Dictionary* or within a *Metadata Stream*. The results are given in Table IV. All tested PDF editors store the date of creation and modification, as well as the creator software, including its version number. Eight editors store the author's name, derived from the name of the currently (at creation time) logged in user. We classify the level of data exposure as “full”, if a PDF editor silently stores the author’s name (i.e., the username) and as “limited” if only dates or creator software strings are stored.

We also performed a large-scale evaluation, of 294 586 PDF files downloaded from the Internet of which 173 112 (58%) contained an author name. Of course, we cannot make any statement if this information was included on purpose or by accident. The single largest creator software of documents containing an author was Microsoft Office with 64 167 files.

2) Revision Recovery: The PDF standard allows editing applications to modify existing documents while only appending to the file and leaving the original data intact. Whenever new content is added to the document, it is not simply inserted into the existing body section. Instead, a new body section is appended at the end of the PDF file containing new objects.[30] This feature is called “incremental updates”. It enables authors, for example, to undo changes. However, it also enables third parties to restore previous versions of the document, which may not be desired in the context of privacy and document security. Especially when sensitive content is explicitly redacted/blackened in a document to be published, this can be dangerous. Instead of deleting the underlying text object, PDF editors may simply overlay a black rectangle, allowing for easy “unredaction”. Poorly redacted documents revealing classified information have been published by the Washington Post [23], the Pentagon [41], Facebook [42], and many others. Although this is a well-known problem and has been researched for PDF documents generated by various office suites [26], modern PDF editors have an explicit “redact” function, which has not yet been comprehensively evaluated. Therefore, we systematically analyze how document modification and text redaction is implemented in PDF editors.

To test if sensitive information can be recovered from a document redacted by a PDF editor, we used two PDF files – one containing selectable text, the other containing a scanned document (i.e., an image).[31] We applied the PDF editor’s “redact” function to draw a black rectangle over parts of the document as well as the “delete” function to remove the text or image. In all tested PDF editors, the “redaction” feature was found to be secure, because the actual *content* of the text or image object was modified, thereby overwriting potentially sensitive content in the file. However, we determined potential security issues in Acrobat Pro and and four other PDF editors, whereby we deleted the content (text or image). The removed content is not displayed anymore, but it is still contained in the file and can be extracted. We do classify the level of data exposure as “limited” in our evaluation (see Table IV), because the “delete” function is not explicitly promoted as a secure feature, even though users may misinterpret it as such. To conclude, redaction tools in PDF viewers can be considered as well-developed these days. The only identified risk is caused by removing sensitive information without explicitly using the *redact* feature of the PDF editors. This approach does not provide the same security level and should be avoided.

Application		Evitable Metadata	Revision Recovery
Acrobat Pro DC	Windows	◐	◐
Foxit PhantomPDF		●	○
PDF-XChange Editor		◐	○
Perfect PDF Premium		●	○
PDF Studio Pro		◐	○
Nitro Pro		●	◐
Nuance Power PDF		●	◐
iSkysoft PDF Editor		●	◐
Master PDF Editor		◐	○
Soda PDF Desktop		●	○
PDF Architect		●	○
PDFelement		●	◐

● Full data exposure ◐ Limited data exposure ○ No exposure

TABLE IV. HIDDEN DATA IN PDF DOCUMENTS.

IX. COUNTERMEASURES

In this section, we discuss short-term mitigations as well as more generic in-depth countermeasures to be considered by implementations and future versions of the PDF standard.

A. Towards an Unambiguous Specification

To counter infinite loops, constructs that can lead to cycles or recursion, such as self-referencing objects, must be prohibited in implementations (e.g., by remembering their path) and ambiguous formulations should be removed from the standard. A clearly stated specification would also help to prevent content masking attacks. In practice, this is not trivial as it would require a formal model of the PDF standard, in order to prove that the model is cycle free, and that a certain document can only be processed in one single way. Furthermore, it must be noted that an unambiguous PDF specification would only protect the document structure, not embedded data formats such as calculator functions, XML, JavaScript, Flash, etc.

B. Resource Limitation and Sandboxing

To counter compression bombs, [45] propose to halt decompression once the size of the decompressed data exceeds an upper limit. This strategy should be applied by PDF processing applications. It must, however, be noted that a single document can contain thousands of streams to be processed in a row. In general, the authors think that limiting the resources to be consumed by a single document, by sandboxing it – similar to a tab in a modern web browser – is a good approach, thereby preventing a malicious document to affect the whole application or even the whole operating system.

C. Identification of Dangerous Paths

Considering Figure 1, our attacks took a path from the top to the file handle. If the path was neither blocked nor required user consent, the attack was successful. Many viewer applications blocked particular paths, but failed to block all of them, thereby allowing us to bypass existing protection mechanisms. This reveals the need for a systematic approach to analyze insecure features in PDF documents. Two positive examples for blocking dangerous paths are Safari and Edge.

[30] A new *XRef* index table and a new trailer must also be appended.

[31] We used the scan of a document from WWI, describing cipher techniques, which was recently declassified by the CIA and can be downloaded from: https://www.cia.gov/library/readingroom/docs/Secret-writing-document-one.pdf.

These application blocked all but one path: *Annotation ⇒(link) ⇒ URI Action ⇒ URL*. In addition, this path required user interaction by actively clicking on a *Link Annotation*. This example illustrates how a secure PDF application should work. We would like to see more applications that restrict the dangerous paths systematically (e.g., by removing them completely or by asking the user for consent). This would reliably prevent all possible variants of *URL invocation*, *form data leakage*, *local file leakage*, *credential theft*, *form modification*, *file write access*, and *code execution* attacks discussed in this paper.

Launch	*Thread*	*GoToE*	*GoToR*	*SubmitForm*	*ImportData*	*URI*
532 (0.18%)	4416 (1.49%)	0 (0.00%)	693 (0.23%)	64 (0.02%)	0 (0.00%)	46 612 (15.82%)

TABLE V. PDF ACTIONS IN 294 586 ANALYZED DOCUMENTS.

As part of this work, we conducted a large-scale evaluation of 294 586 publicly available PDF documents. We analyzed these files for the various PDF action types by first uncompressing all contained streams and then searching for the patterns which define a certain action (e.g., `/SubmitForm`). Results on how many documents contain a certain action are depicted in Table V. As one can see, the only action-based PDF feature that is widely in practice is the *URI* action, which can be restricted to a *Link Annotation*. Insecure features instead are rarely used in real-world PDF documents. Therefore, it can be concluded that PDF viewers should drop support for potentially dangerous features such as the *Launch* action or at least disable them in the default settings.

D. Removing or Restricting JavaScript

JavaScript support in PDF applications is extremely varied. The absence of a sound test suite to accompany the standard makes it difficult for developers to create compliant and robust implementations. In addition, the great disparity between PDF viewers regarding their feature support complicates the effective utilization of JavaScript by authors of PDF documents. While we could observe some viewers to borrow a stable JavaScript engine from other projects, such as SpiderMonkey or V8, multiple viewers provide very unstable homebrewed solutions which can be crashed with ease. Unrelated to the used engine, many viewers implement obscure JavaScript API functions without providing public documentation. Neither their purpose nor resistance to exploitation is clear.

Given that PDF is supposed to be a format for portable documents, the need to embed a full programming language is debatable. Many legitimate use cases of JavaScript in PDF, such as input validation of form fields, can be covered without a programming language, as established and proven in HTML5.[32]. Any scenario exceeding the declarative markup features of PDF should be considered to be implemented as a web application instead of a PDF document, given that JavaScript support and the security of modern web browsers is well researched and robustly implemented.

E. Implementing Privacy by Default

PDF editors should not include excessive metadata such as usernames in the default settings. Furthermore, all editing functions (redaction, modification, and deletion of elements) should be performed on the actual object to prevent a third party from recovering previous versions of the document. Such best practices regarding metadata and text redaction should not only be applied by PDF editors, but by all applications that allow to export content to PDF (e.g., office suites).

X. CONCLUSION

PDF is more than a simple document format. Each standard compatible PDF viewer must support a large set of additional features. While PDF exploitation caused by implementation bugs, such as buffer overflow based code execution, has been a long-standing research area with many important results, a security evaluation of standard PDF features has just started.

A. Systematization of PDF Processing Model

The research presented in this paper can be seen as a first step towards a systematization of research on PDF security *within* the PDF data processing model. All of our test cases fall within the PDF specification, and mitigations against the described attacks often consist in omitting certain standard PDF features (e.g., the *Launch* action). However, research in this direction, until now, was limited to picking some functionality, evaluating it and in case of successful attacks, (partially) disabling this single functionality. This will close single security holes, but will not result in a provably secure PDF viewer specification. Instead, we have to fully understand the data processing model behind the PDF standard to be able to define what *secure PDF rendering* means.

B. Future Research Directions

1) Printers and PDF Libraries Used by Web Applications: Modern printers are able to natively process PDF files and print them to paper. Some of our attack classes are highly relevant to these embedded interpreters. Examples are DoS, local file leakage, content masking, or code execution. Sending a PDF document to a company employee which does render on a desktop PDF viewer, but causes a DoS attack on network printers, may have a large attack potential. Web applications which parse uploaded PDF files (e.g., to generate preview images) also may show security weaknesses. While we did not evaluate PDF parser libraries used in printers or in web applications, our attack vectors may still be applicable here.

2) Automatic Test Vector Generation: Automatically generating test vectors from a human-readable specification remains an open problem in software engineering. This especially holds for compliance tests. Even if such generation tools were available, it would be questionable whether the test suite of PDF files used in our evaluation could be generated by them. Although our test cases are valid PDF documents, they are edge cases and are not necessarily reproduced by *specification coverage* [28]. We conclude that an open question for the research community is to generate such security test cases automatically, not only relying on compliance test vectors.

[32] See https://html.spec.whatwg.org/multipage/input.html#input-impl-notes.

References

[1] Access Denied. *DFS Issue 55*. http://textfiles.com/magazines/DFS/dfs055.txt. May 1996.

[2] Adobe Systems. *Acrobat JavaScript Scripting Guide*. 2005.

[3] Adobe Systems. *Adobe Supplement to the ISO 32000, BaseVersion: 1.7, ExtensionLevel: 3*. 2008.

[4] Adobe Systems. *Applying Actions and Scripts to PDFs*. https://helpx.adobe.com/acrobat/using/applying-actions-scripts-pdfs.html. 2019.

[5] Adobe Systems. *Displaying 3D Models in PDFs*. https://helpx.adobe.com/acrobat/using/displaying-3d-models-pdfs.html. 2017.

[6] Adobe Systems. *How to fill in PDF forms*. https://helpx.adobe.com/en/acrobat/using/filling-pdf-forms.html. 2019.

[7] Adobe Systems. *Starting a PDF review*. https://helpx.adobe.com/acrobat/using/starting-pdf-review.html. 2019.

[8] Adobe Systems. *XMP Specification Part 1*. 2012.

[9] National Security Agency. *Hidden Data and Metadata in Adobe PDF Files: Publication Risks and Countermeasures*. 2008.

[10] A. Albertini. "This PDF is a JPEG; or, This Proof of Concept is a Picture of Cats". In: *PoC 11 GTFO 0x03* (2014). URL: https://www.alchemistowl.org/pocorgtfo/pocorgtfo03.pdf.

[11] C. Alonso et al. *Disclosing Private Information from Metadata, Hidden Info and Lost Data*. 2008.

[12] P. Bieringer. *Decompression Bomb Vulnerabilities*. 2001.

[13] A. Blonce, E. Filiol, and L. Frayssignes. "Portable Document Format Security Analysis and Malware Threats". In: *BlackHat Europe* (2008).

[14] Boxcryptor. *Malware in Email Attachments: Which File Extensions are Dangerous?* https://boxcryptor.com/blog/post/malware-in-email-attachments/. 2019.

[15] C. Carmony et al. "Extract Me If You Can: Abusing PDF Parsers in Malware Detectors." In: *NDSS*. The Internet Society, 2016.

[16] A. Castiglione, A. De Santis, and C. Soriente. "Security and Privacy Issues in the Portable Document Format". In: *Journal of Systems and Software* 83.10 (2010), pp. 1813–1822.

[17] T. Claburn. *Use an 8-char Windows NTLM password?* https://www.theregister.co.uk/2019/02/14/password_length/. Feb. 2019.

[18] I. Corona et al. "Lux0r: Detection of Malicious PDF-Embedded JavaScript Code through Discriminant Analysis of API References". In: *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*. ACM. 2014, pp. 47–57.

[19] CVE Details. *Adobe Acrobat Reader: Security Vulnerabilities (DoS)*. https://www.cvedetails.com/vulnerability-list/vendor_id-53/product_id-497/opdos-1/Adobe-Acrobat-Reader.html. 2006.

[20] P. Deutsch. *DEFLATE Compressed Data Format Specification*. 1996.

[21] ECMA. *ECMAScript Language Specification, 3rd Edition*. 1999.

[22] E. Ellingsen. *ZIP File Quine: Droste.zip*. https://web.archive.org/web/20160130230432/http://www.steike.com/code/useless/zip-file-quine/.

[23] K. Foss. *Washington Post's scanned-to-PDF Sniper Letter More Revealing Than Intended*. http://web.archive.org/web/20040204141449/http://planetpdf.com/mainpage.asp?webpageid=2434. 2002.

[24] G. Franken, T. Van Goethem, and W. Joosen. "Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-Party Cookie Policies". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 151–168.

[25] J. Gajek. "Macro Malware: Dissecting a Malicious Word Document". In: *Network Security* 2017.5 (2017), pp. 8–13.

[26] S. Garfinkel. "Leaking Sensitive Information in Complex Document Files – and How to Prevent It". In: *IEEE Security & Privacy* 12.1 (2013), pp. 20–27.

[27] V. Hamon. "Malicious URI resolving in PDF documents". In: *Journal of Computer Virology and Hacking Techniques* 9.2 (2013), pp. 65–76.

[28] Michael Harder, Benjamin Morse, and Michael D Ernst. "Specification Coverage as a Measure of Test Suite Quality". In: (2001).

[29] Chris Hummel. "Why Crack When You Can Pass The Hash". In: *SANS Institute InfoSec Reading Room* 21 (2009).

[30] A. Inführ. *Adobe Reader PDF - Client Side Request Injection*. 2018.

[31] A. Inführ. *Multiple PDF Vulnerabilities – Text and Pictures on Steroids*. 2014.

[32] A. Inführ. *PDF – Mess with the Web*. Sept. 2015.

[33] B. Krebs. *Document Security 101*. http://voices.washingtonpost.com/securityfix/2005/12/document_security_101_1.html. 2005.

[34] P. Laskov and N. Šrndić. "Static Detection of Malicious JavaScript-bearing PDF Documents". In: *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM. 2011, pp. 373–382.

[35] J. Magazinius, B. Rios, and A. Sabelfeld. "Polyglots: Crossing Origins by Crossing Formats". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. ACM. 2013, pp. 753–764.

[36] D. Maiorca and B. Biggio. "Digital Investigation of PDF Files: Unveiling Traces of Embedded Malware". In: *IEEE Security and Privacy: Special Issue on Digital Forensics* (2017).

[37] D. Maiorca, G. Giacinto, and I. Corona. "A Pattern Recognition System for Malicious PDF Files Detection". In: *International Workshop on Machine Learning and Data Mining in Pattern Recognition*. Springer. 2012, pp. 510–524.

[38] D. Maiorca et al. "A Structural and Content-Based Approach for a Precise and Robust Detection of Malicious PDF Files". In: *2015 International Conference on Information Systems Security and Privacy (ICISSP)*. IEEE. 2015, pp. 27–36.

[39] I. Markwood et al. "PDF Mirage: Content Masking Attack Against Information-Based Online Services". In: *26th USENIX Security Symposium (USENIX Security 17), (Vancouver, BC)*. 2017, pp. 833–847.

[40] M. Marlinspike. "Divide and Conquer: Cracking MS-CHAPv2 with a 100% success rate". In: *CloudCracker [online]* 29 (2012).

[41] K. McCarthy. *That classified US military report's secrets in full*. https://theregister.co.uk/2005/05/03/military_report_secrets/. 2005.

[42] A. Nusca. *Facebook settlement revealed via poor PDF redaction*. https://www.zdnet.com/article/facebook-settlement-revealed-via-poor-pdf-redaction/. 2009.

[43] N. Ochoa. *Pass-The-Hash Toolkit-Docs & Info*. 2008.

[44] Parker, T. *How to do (not so simple) form calculations*. https://acrobatusers.com/tutorials/print/how-to-do-not-so-simple-form-calculations. July 2006.

[45] G. Pellegrino et al. "In the Compression Hornet's Nest: A Security Study of Data Compression in Network Services". In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 801–816.

[46] C. Pesce. *Document Metadata, the Silent Killer...*

[47] D. Poddebniak et al. "Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 549–566.

[48] S. Rautiainen. "A Look at Portable Document Format Vulnerabilities". In: *Information Security Technical Report* 14.1 (2009), pp. 30–33.

[49] F. Raynal, G. Delugré, and D. Aumaitre. "Malicious Origami in PDF". In: *Journal in Computer Virology* 6.4 (2010), pp. 289–315. URL: http://esec-lab.sogeti.com/static/publications/08-pacsec-maliciouspdf.pdf.

[50] Check Point Research. *NTLM Credentials Theft via PDF Files*. https://research.checkpoint.com/ntlm-credentials-theft-via-pdf-files/. 2018.

[51] B. Rios, F. Lanusse, and M. Gentile. *Adobe Reader Same-Origin Policy Bypass*. http://www.sneaked.net/adobe-reader-same-origin-policy-bypass. Jan. 18, 2013.

[52] K. Selvaraj and N. Gutierrez. *The Rise of PDF Malware*. Tech. rep. Symantec, 2010. URL: https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/security-response-rise-of-pdf-malware-10-en.pdf.

[53] C. Smutz and A. Stavrou. "Malicious PDF Detection Using Metadata and Structural Features". In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM. 2012, pp. 239–248.

[54] Aaron Spangler. *WinNT/Win95 Automatic Authentication Vulnerability (IE Bug #4)*. https://insecure.org/sploits/winnt.automatic.authentication.html. Mar. 1997.

[55] N. Šrndić and P. Laskov. "Hidost: A Static Machine-Learning-Based Detector of Malicious Files". In: *EURASIP Journal on Information Security* 2016.1 (2016), p. 22.

[56] Sutherland, E. *First Reported PDF Virus Is Not 'Peachy'*. http://web.archive.org/web/20030617154329/http://www.osopinion.com/perl/story/12626.html. 2001.

[57] Symantec. *VBS/PeachyPDF@MM*. Aug. 2001. URL: https://www.symantec.com/security-center/writeup/2001-080705-1926-99.

[58] Symantec. *W32/.Yourde-A*. Apr. 2003. URL: https://www.symantec.com/security-center/writeup/2003-050108-4923-99.

[59] Adobe Systems. *Fast Facts*. 2018. URL: https://www.adobe.com/content/dam/cc/en/fast-facts/pdfs/fast-facts.pdf.

[60] Adobe Systems. *PDF Reference, version 1.7*. sixth edition. Nov. 2006.

[61] L. Tong et al. *A Framework for Validating Models of Evasion Attacks on Machine Learning, with Application to PDF Malware Detection*.

[62] L. Tong et al. "Feature Conservation in Adversarial Classifier Evasion: A Case Study". In: *CoRR* abs/1708.08327 (2017).

[63] H. Valentin. "Malicious URI Resolving in PDF Documents". In: *BlackHat Abu Dhabi* (2012).

[64] M. West and M. Goodwin. "Same-site Cookies". In: *Internet Engineering Task Force Secretariat* (2016), pp. 1–14.

		Pages loop				GoTo loop				Action loop			Calculator functions		Outline loop		ObjStm loop	JavaScript loop			Deflate bomb
	Application	A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	C3	D1	D3	E2	E3	F1	G1	G2	G3	–
Windows	Acrobat Reader DC	○	○	○	○	○	○	○	💣	○	○	🕘	○	○	○	○	🕘	🕘	💣	🕘	🕘
	Foxit Reader	○	○	🕘	○	○	💣	💣	💣	○	💣	💣	○	○	○	○	○	🕘	○	🕘	🕘
	PDF-XChange Viewer	💣	💣	💣	💣	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	🕘
	Perfect PDF Reader	○	○	💣	💣	○	○	○	○	○	○	○	○	○	○	🕘	○	○	○	○	🕘
	PDF Studio Viewer	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	🕘
	Nitro Reader	○	○	○	○	💣	○	○	○	○	○	○	○	○	💣	💣	○	🕘	🕘	🕘	○
	Acrobat Pro DC	○	○	○	○	○	○	○	💣	○	○	○	○	○	○	○	🕘	🕘	💣	🕘	🕘
	Foxit PhantomPDF	○	○	🕘	○	💣	💣	💣	💣	○	💣	💣	○	○	○	○	○	🕘	○	🕘	🕘
	PDF-XChange Editor	○	○	○	○	○	🕘	🕘	○	○	💣	💣	○	○	○	○	○	🕘	○	🕘	🕘
	Perfect PDF Premium	○	○	💣	💣	○	○	○	○	○	○	○	○	🕘	○	🕘	○	○	○	○	🕘
	PDF Studio Pro	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	🕘
	Nitro Pro	○	○	○	○	💣	○	○	○	○	○	○	○	○	💣	💣	○	🕘	🕘	🕘	○
	Nuance Power PDF	○	○	💣	💣	💣	○	○	💣	💣	💣	💣	💣	○	💣	💣	○	🕘	○	🕘	💣
	iSkysoft PDF Editor	○	○	○	○	○	○	○	○	○	○	○	○	○	🕘	🕘	○	○	○	○	○
	Master PDF Editor	○	○	○	○	💣	○	○	○	○	○	○	○	○	○	○	○	🕘	○	🕘	○
	Soda PDF Desktop	○	○	○	○	○	○	○	○	○	💣	💣	○	○	💣	💣	○	🕘	💣	💣	🕘
	PDF Architect	○	○	○	○	○	○	○	○	○	💣	💣	○	○	💣	💣	○	🕘	💣	💣	🕘
	PDFelement	○	○	○	○	○	○	○	○	○	○	○	○	○	💣	🕘	○	○	○	○	○
Mac	Preview	○	○	○	○	○	○	○	○	💣	💣	💣	○	○	○	○	○	○	○	○	🕘
	Skim	○	○	○	○	○	○	○	○	💣	💣	💣	○	○	○	○	○	○	○	○	🕘
Linux	Evince	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	💣
	Okular	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	🕘	○	🕘	💣
	MuPDF	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	🕘	🕘	🕘	○
Web	Chrome	○	○	○	○	(🕘)	○	○	○	○	○	(🕘)	○	○	○	○	○	(🕘)	○	○	🕘
	Firefox	○	○	○	(🕘)	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	🕘
	Safari	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
	Opera	○	○	○	○	(🕘)	○	○	○	○	○	(🕘)	○	○	○	○	○	(🕘)	○	○	(🕘)
	Edge	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○

💣 Application crashes 🕘 Applications hangs (🕘) Only current tab hangs ○ Not vulnerable

TABLE VI. DETAILED RESULTS FOR THE DENIAL-OF-SERVICE CLASS OF ATTACKS.

APPENDIX

A. *Availability of Artifacts*

We released a comprehensive test suite of malicious PDF files which can be used by developers to test their software. All proof of concept exploit files are available for download from https://pdf-insecurity.org/download/pdf-dangerous-paths/exploits-and-helper-scripts.zip, to allow for easy reproduction.

B. *Evaluation Details: Denial of Service*

In Table VI, full evaluation details for the DoS class of attacks are given. Test cases (e.g., $A1$) follow the same naming convention as the proof-of-concept files provided as artifacts, which are available online.

C. *Evaluation Details: Content Masking*

Table VII shows detailed evaluation results for content masking attacks. Each column corresponds to a test case in the artifacts. Columns which did not produce ambiguous results (i.e., render similar in all tested applications) have been stripped for reasons of clarity.

Acknowledgements

Jens Müller was supported by the research training group "Human Centered System Security", sponsored by the state of North Rhine-Westfalia. Dominik Noss was supported by the research project "MITSicherheit.NRW" funded by the European Regional Development Fund North Rhine-Westphalia (EFRE.NRW). In addition, this work was supported by the German Research Foundation (DFG) within the framework of the Excellence Strategy of the Federal Government and the States – EXC 2092 CASA.

Application	A1	A3	A4	A5	B1	C1	C2	C3	C4	C6	C7	C8	CX	D1	D2	D4	E3	E4	F1	F3	G1	G3	H2	H3	H5	H6	I3	J1	K1	K4	K5	K6	K7	K8	M3	M4	N1	N2	N3	N4	N5	P1	P3	P4	P6	P7	P8	P9	PX	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	QX
Acrobat Reader/Pro	2	2	1	1	1	–	1	1	–	1	1	–	1	2	2	2	2	1	2	2	–	–	–	–	–	–	2	–	–	1	–	2	2	2	1	2	2	2	1	1	1	–	–	–	2	2	2	–	–	–	–	–	–	2	2	–	–	–	–
Foxit Reader	2	2	2	1	2	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	2	1	2	2	2	1	2	2	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
Foxit PhantomPDF	2	2	2	1	2	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	2	1	2	2	2	1	2	2	2	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
PDF-XChange Viewer	2	2	2	2	2	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	1	1	2	1	2	1	2	1	1	2	1	2	2	2	2	2	2	1	2	2	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
PDF-XChange Editor	2	2	2	1	2	2	1	1	2	1	1	2	1	2	2	2	2	–	2	2	2	2	2	1	2	1	2	1	2	2	1	2	2	2	1	2	2	1	2	2	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Nitro Reader/Pro	2	2	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	1	2	2	1	1	–	1	–	1	2	1	2	1	1	2	2	2	1	1	2	1	2	2	2	1	2	2	2	2	2	–	–	–	–	2	2	2	2	2	2	2	2
Nuance Power PDF	2	2	1	1	1	1	1	1	1	1	1	1	1	2	2	2	1	1	1	2	1	1	–	2	–	2	2	2	1	1	1	1	1	1	1	1	2	1	2	2	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Soda PDF Desktop	2	2	2	1	2	2	1	1	2	1	1	2	1	2	2	2	2	–	1	1	1	1	1	1	1	1	2	1	1	1	1	1	1	1	2	2	1	1	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
PDF Architect	2	2	2	1	2	2	1	1	2	1	1	2	1	2	2	2	2	–	1	1	1	1	1	1	1	1	2	1	1	1	1	1	1	1	2	2	1	1	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Poppler (Evince/Okular)	2	2	2	1	2	1	–	–	1	–	–	1	–	2	2	2	2	1	2	2	–	–	–	–	–	–	2	–	2	1	2	2	2	2	2	2	2	1	2	2	2	2	–	–	2	2	2	2	2	2	2	–	–	2	2	–	–	–	–
Chrome	2	2	2	1	2	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	2	1	2	2	2	1	2	2	1	1	2	2	1	1	1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
Firefox	2	2	2	1	2	1	1	1	1	1	1	1	1	2	2	2	2	2	1	2	–	–	1	–	1	–	–	–	–	1	–	2	2	2	–	–	1	1	2	2	2	–	–	–	–	2	2	2	2	2	2	–	–	2	2	–	–	–	–
Opera	2	2	2	1	2	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	2	1	2	2	2	1	2	2	1	1	2	2	1	1	1	2	2	2	2	2	2	2	2	2	2	2	–	–	–	–
Perfect PDF Reader	1	1	2	–	1	1	1	2	1	1	2	1	1	1	1	1	1	1	2	2	1	2	1	1	1	1	2	1	1	1	1	2	2	2	–	–	2	1	2	2	2	1	2	–	–	2	–	2	–	2	–	2	–	–	–	2	–	–	–
Perfect PDF Premium	1	1	2	1	2	1	1	2	1	1	2	1	1	1	1	1	2	1	2	2	1	2	1	1	1	1	2	1	1	1	1	2	2	2	–	–	2	1	2	2	2	1	–	–	–	2	–	2	–	2	–	–	–	2	2	–	–	–	–
PDF Studio Viewer/Pro	1	2	2	1	2	2	–	–	2	–	–	2	–	2	2	2	2	–	2	2	1	1	1	1	1	1	–	1	1	1	1	1	1	1	2	2	2	1	2	2	2	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
iSkysoft PDF Editor	1	1	2	1	2	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	1	1	1	1	1	1	1	1	1	2	1	2	2	2	1	2	2	1	1	2	2	1	1	1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
Master PDF Editor	1	1	2	1	2	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	1	1	1	1	1	1	1	1	1	2	1	2	2	2	1	2	2	1	1	2	2	1	1	1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
PDFelement	1	1	2	1	2	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	1	1	1	1	1	1	1	1	1	2	1	2	2	2	1	2	2	1	1	2	2	1	1	1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
Preview	1	2	2	1	2	1	–	–	1	–	–	1	–	2	2	2	2	1	2	2	–	–	–	–	–	–	2	–	–	1	–	2	2	2	2	2	2	1	2	2	2	–	–	–	2	2	2	2	2	2	2	–	–	2	2	–	–	–	–
Skim	1	2	2	1	2	1	–	–	1	–	–	1	–	2	2	2	2	1	2	2	–	–	–	–	–	–	2	–	–	1	–	2	2	2	2	2	2	1	2	2	2	–	–	–	2	2	2	2	2	2	2	–	–	2	2	–	–	–	–
MuPDF	1	2	2	1	2	2	1	1	2	1	1	2	1	2	2	2	2	2	2	2	1	1	2	1	2	1	2	1	2	1	1	2	2	2	1	1	2	1	2	2	2	1	1	1	2	2	2	2	2	1	1	1	1	2	2	1	1	1	1
Safari	1	2	2	1	2	1	–	–	1	–	–	1	–	2	2	2	2	1	2	2	–	–	–	–	–	–	2	–	–	1	–	2	2	2	2	2	2	1	2	2	2	–	–	–	2	2	2	2	2	2	2	–	–	2	2	–	–	–	–
Edge	1	1	2	–	2	–	–	–	–	–	–	–	–	2	2	2	2	2	2	2	2	2	–	–	–	–	–	–	2	2	2	2	2	2	1	1	2	1	2	2	2	2	2	–	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

1 First text is displayed   2 Second text is displayed   – No text is displayed

TABLE VII. DETAILED RESULTS FOR THE CONTENT MASKING CLASS OF ATTACKS.