RUHR-UNIVERSITÄT BOCHUM

RUB

Vulnerability Report

Attacks bypassing the signature validation in PDF

Vladislav Mladenov, Christian Mainka, Karsten Meyer zu Selhausen, Martin Grothe, Jörg Schwenk

November 15, 2018.
Chair for Network and Data Security

hgi Lehrstuhl für Netz- und Datensicherheit

1 The scope of the vulnerability report

Research Results

As part of our current research, we analyzed signature validation processing on PDF files. In the following report, we present three novel attack classes: Universal Signature Forgery (USF), Incremental Saving Attack (ISA), and Signature Wrapping Attack (SWA) which we describe in chapter 3. Each attack allows an attacker to stealthily manipulate the content of a signed PDF without invalidating the signature, thereby breaking the document integrity protection.

About us

The Chair of Network and Data Security (NDS) has been working since 2003 under the direction of Prof. Dr.-Ing. Jörg Schwenk on the security analysis of cryptographic protocols (especially in connection with browser-based protocols based on TLS) and the XML format for signature generation and encryption. One focus of the department is to comprehensively explore the multitude of cryptographic techniques and standards used in these fields.

2 PDF Basics

This section deals with the foundations of the Portable Document Format (PDF). We give an overview of the file structure and explain how the PDF standard for signatures is implemented.

2.1 Portable Document Format (PDF)

A PDF consists of four parts: *header*, *body*, *xref table*, and a *trailer*, as depicted in Figure 2.1.

PDF header The *header* is the first line within a PDF and defines the interpreter version to be used. The provided example uses version `PDF 1.7`

PDF body The *body* defines the content of the PDF and contains text blocks, fonts, images, and metadata regarding the file itself. The main building block within the body are *object*s, which have the following structure:

```
1 0 obj
↑ ↑   ↑
│ │   └──────────────────── Constant String “obj”
│ └──────────────────────── Generation Number
└────────────────────────── Object Number
...
endobj
```

Listing 2.1: Example of an object declaration within the body.

Each object starts with an Object Number (№_{obj}) followed by a Generation Number (№_{gen}). The №_{gen} should be incremented if additional changes are made to the object. An object can be referenced by using the following scheme: $|\text{№}_{obj} ␣ \text{№}_{gen} ␣ \text{R}|$, e.g., `1 0 R`.

In the example depicted in Figure 2.1, the *body* contains four objects: *Catalog*, *Pages, Page*, and *stream*. The *Catalog* object is the root object of the PDF file. It defines the document structure and can additionally declare access permissions. The *Catalog* references to one *Pages* object which defines the number of the pages and a reference to each *Page* object (e.g., text columns). The *Page* object contains information how to build a single page. In the given example, it only contains a single string object “Hello World!”.

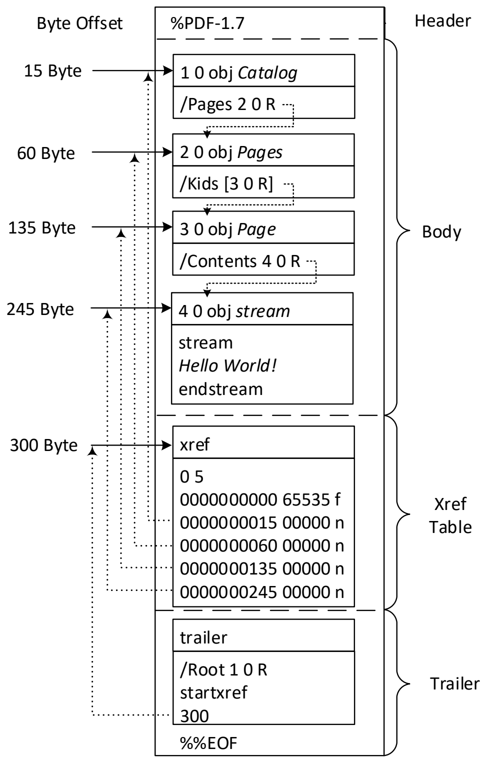

Figure 2.1: A simplified example of a PDF file's internal structure. We depict the object names after the *obj* string for clarification.

Xref table The *Xref table* contains information about all PDF objects. An *Xref table* can contain one or more sections.

- Each *Xref table* section starts with a line consisting of two integer entries a b (e.g., "0 5" as shown in Figure 2.1) which indicates that the following $b = 5$ lines in the *Xref table* describe objects with ID $a \in \{0, \dots, b-1\} = \{0, \dots, 4\}$.
- Lines with the three entries x y z describe an *Xref table* object entry, where x defines the byte offset of the object number a; y defines its $№_{gen}$, and $z \in \{'n',' f'\}$ describes whether the object is in use ("n") or not ("f", say "free"). For example, the line "0000000060 00000 n" is the third line after "0 5" and, thus, describes the in-use object with $№_{obj}$ 2 and $№_{gen}$ 0 at byte offset 60 (see "2 0 obj" in Figure 2.1).

Trailer The *Trailer* is the first processed content of a pdf file. It contains references to the *Catalog* and the *Xref table*.

2.2 PDF Signatures

In this section, the integrity protection of a PDF file provided by a digital signature will be explained further.

Incremental Saving. PDF Signatures rely on a feature of PDF files called *incremental saving* (also known as incremental updates), allowing the modification of a PDF file without changing the previous content.

In Figure 2.2, an original document (shown on a left side) is being modified via *incremental saving* by attaching a new *body*, *Xref table*, and *Trailer* at the end of the file. Within the

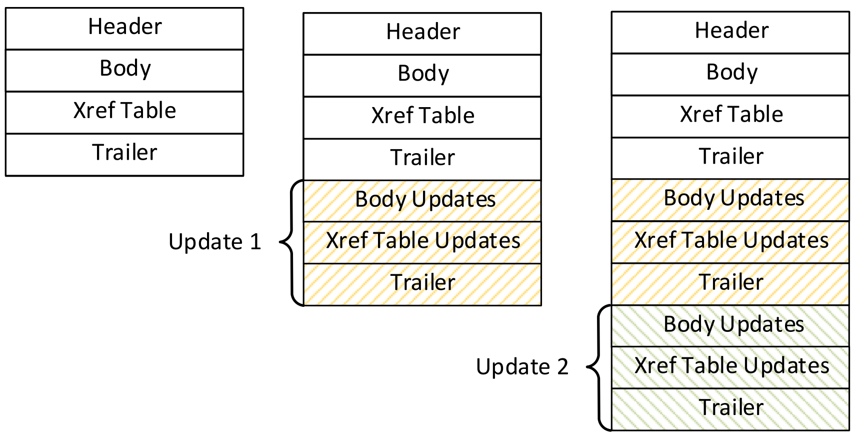

Figure 2.2: Multiple incremental savings applied on a PDF file.

body, new objects can be defined. A new *Pages* object can be defined, containing two pages, for example, `/Kids [3 0 R 3 0 R]`. For reasons of simplicity, the same content was used here twice. The *Xref table* contains only a description of the newly defined objects. The new *Trailer* contains a reference to the *Catalog* (it could be the old *Catalog* or an updated one), the byte offset of the new *Xref table*, and the byte offset of the previously used *Xref table*. This scheme is applied for each *incremental saving*.

Structure of a Signed PDF The creation of a digital signature on a PDF file relies on *incremental saving* by extending the original document with objects containing the signature information.

In Figure 2.3, an example of a signed PDF file is shown. The original document is the same document as depicted in Figure 2.1. By signing the document, an *incremental saving* is applied and the following content is added: a new *Catalog*, a *Signature* object, a new *Xref table* referencing the new object(s), and a *Trailer*. The new *Catalog* extends the old one by adding a new parameter *Perms*, defining the restrictions with respect to changes within the document. The *Perms* parameter references to the *Signature* object.

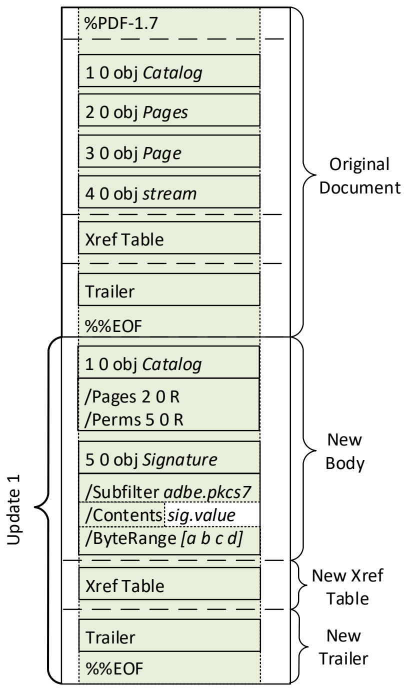

Figure 2.3: A simplified overview of a signed PDF file.

The *Signature* object (`5 0 obj`) contains information regarding the applied cryptographic algorithms for hashing and signing the document. It additionally includes a `Contents` parameter containing a hex-encoded PKCS7 blob, holding the certificates used to sign the document as well as the signature value. The `ByteRange` parameter defines which bytes of the PDF file are used as the hash input for the signature calculation and defines two integer tuples:

(a, b) : Beginning at byte offset `a`, the following `b` bytes are used as input for the hash calculation. Typically, $a = 0$ is used to indicate that the beginning of the file is used while $a + b$ is the byte offset where the PKCS#7 blob begins.

(c, d) : Typically, byte offset c is the end of the PKCS#7 blob, while $c + d$ points to the last byte off the PDF file.

According to the specification, it is *recommended* to sign the whole file except for the PKCS#7 blob.

3 How To Break PDF Signatures

In this section, we present three novel attack classes on PDF signatures: Universal Signature Forgery (USF), Incremental Saving Attack (ISA), and Signature Wrapping Attack (SWA). All attack classes bypass the PDF's signature integrity protection allowing the modification of the content arbitrarily without the victim noticing.

The attacker's goal is to place a *malicious object* into the protected PDF file, such that the target viewer shows different content in comparison to the originally signed PDF file. Nevertheless, the viewer indicates that the signature is valid and that no changes have been made to the document after signing.

3.1 Universal Signature Forgery (USF)

The main idea of USF is to disable the verification by providing invalid content within the signature object or removing the references to the signature object. Thus, despite the fact that the signature object is provided, the validation logic is not able to apply the correct cryptographic operations. Nevertheless, it could be possible that a viewer shows some signature information although the verification is being skipped.

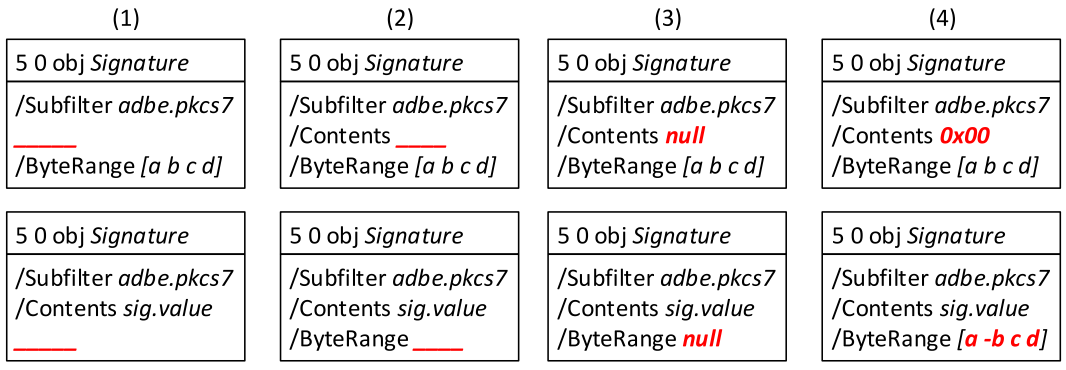

Figure 3.1: Different USF attack vectors manipulating the signature object entries within a signed PDF to bypass the signature validation.

We defined 24 different attack vectors. Eight of them are depicted in Figure 3.1. In the given example, the attack vectors target two values: (1) `Contents` containing the key material and the signature value and (2) `ByteRange` defining the signed content. The manipulation of these parameters is reasoned by the fact that we either remove the signature value or the information which content is signed. In (1), either `Contents` or `ByteRange` are removed from the signature object. Another possibility is defined in (2) by removing only the content of the entries. In (3) and (4), invalid values were specified and tested. Such values are for instance `null`, a zero byte (`0x00`), and invalid `ByteRange` values like negative or overlapping byte ranges.

3.2 Incremental Saving Attack (ISA)

This class of attack relies on the *incremental saving* feature. The idea of the attack is to make an incremental saving on the document by redefining the document's structure and content using the *Body Updates* part. The digital signature within the PDF file protects exactly the part of the file defined in the `ByteRange`. Since the *incremental saving* appends the *Body Updates* to the end of the file, it is not part of the defined `ByteRange` and thus not part of the signature's integrity protection. The signature remains valid, while the *Body Updates* changed the displayed content.

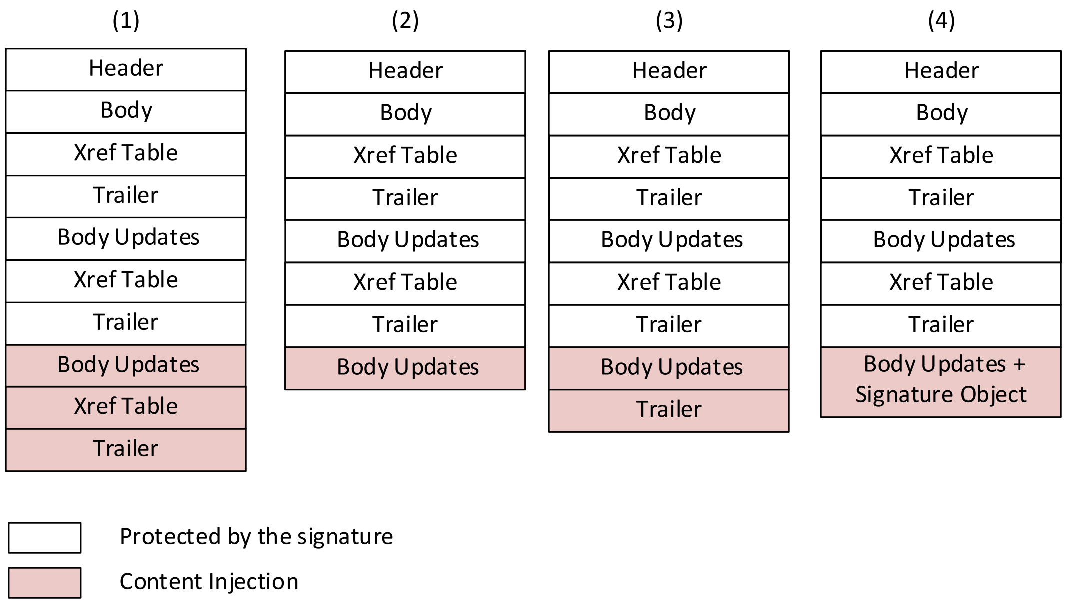

Figure 3.2: Bypassing the signature protection by using *incremental saving*. In (1), the main idea of the attack is depicted, while (2)-(4) are variants to obfuscate the manipulations and prevent a viewer to display warnings.

Variant 1 Considering variant (1) in Figure 3.2, only two of the evaluated signature validators was susceptible to the attack. This is not very surprising since this type of modification is exactly what a legitimate PDF application would do when editing or updating a

PDF file. A PDF digital signature is designed to protect against this behavior; the signature validator recognizes that the document was updated after signing and shows a warning respectively. To bypass this detection, we included an *Xref table* which: (1) is empty. An empty *Xref table* can be interpreted as a sign that no objects are changed by the last incremental update. Nevertheless, the included updates are processed and displayed by the viewer; (2) contains entries for all manipulated objects and an entry with an incorrect reference to the transform parameters dictionary which is part of the signature object. The result of these manipulations is that the last incremental saving is not detected and no warning is shown, but the new objects are displayed by the PDF viewer.

Variant 2: ISA without *Xref table* and *Trailer* Some of the viewers detected the manipulation by checking if a new *Xref table* and *Trailer* were defined within the new incremental update. By removing the *Xref table* and the *Trailer*, a vulnerable validator does not recognize that *incremental saving* has been applied and successfully verifies the signature without showing a warning. The PDF file is still processed normally by displaying the modified document structure. The cause for this behavior is that many of the viewers are error tolerant. In the given case, the viewer completes the missing *Xref table* and *Trailer*, and processes the manipulated *body*.

Variant 3: ISA with a *Trailer* Some of the PDF viewers do not open the PDF file if a *Trailer* is missing. This led to the creation of this attack vector containing a manipulated *Trailer* at the end of the file. Interestingly, the *Trailer* must not point to a *Xref table*, but any other byte offset within the file. Otherwise, the verification logic detects the document manipulation.

Variant 4: ISA with a copied signature and without a *Xref table* and *Trailer* The previous manipulation technique was improved by copying the *Signature* object within the last incremental update. This improvement was forced by some validators which require any incremental update to contain a signature object, otherwise, they throw a warning that the document was modified after the signing.

By copying the original *Signature* object into the latest incremental update, this requirement is fulfilled. The copied *Signature* object, however, covers the old document and not the updated part. To summarize, a vulnerable validator does not verify whether each incremental update is signed, but only if it contains a signature object. Such verification logic is susceptible to ISA.

3.3 Signature Wrapping Attack (SWA)

The SWA introduces a novel technique to bypass the signature protection without using incremental saving.

The main idea is to move the second part of the signed `ByteRange` to the end of the document while reusing the `xref` pointer within the signed *Trailer* to an attacker manipulated *Xref table*. To avoid any processing of the relocated second part, it can be optionally wrapped by using a stream object or a dictionary. In Figure 3.3, two documents are depicted. On the left side, a validly signed PDF file is depicted. On the right side, a manipulated PDF file is generated by using SWA. During the SWA, the attacker proceeds as

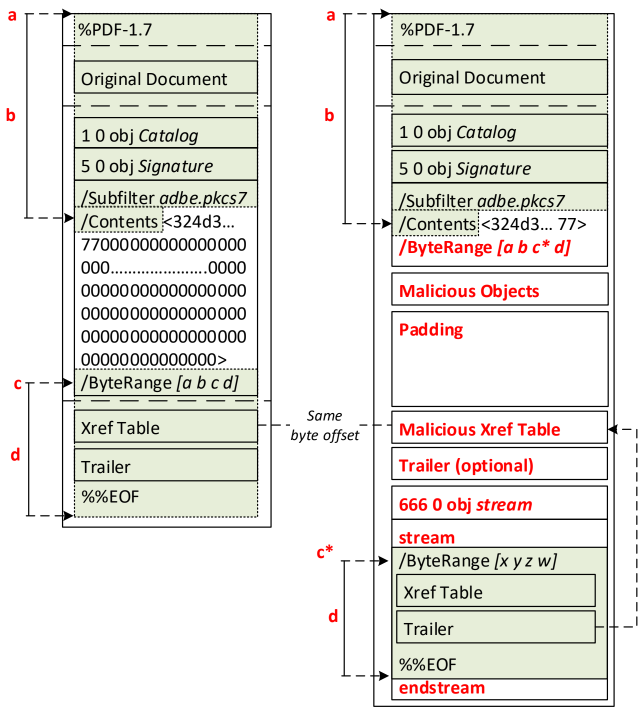

Figure 3.3: A comparison of the original document and the manipulated document by using the Signature Wrapping Attack (SWA). Malicious objects are placed before the malicious *Xref table* table by deleting unused zero Bytes in `Contents`.

follows:

Step 1 (optional): The attacker deletes the padded zero Bytes within the `Contents` parameter to increase the available space for injecting manipulated objects.[1]

Step 2: The attacker defines a new `/ByteRange [a,b,c*,d]` by manipulating the `c` value, which now points to the second signed part placed on a different position within the document.

Step 3: The attacker creates a new *Xref table* pointing to the new objects. It is essential that the byte offset of the newly inserted *Xref table* has the same byte offset as the previous *Xref table*. The position is not changeable since it is referenced by the

[1]During signing the size of the signature value (and the corresponding certificate) is not known and thus it is roughly estimated. The unused bytes are later filled with zero Bytes.

signed *Trailer*. For this purpose, the attacker can add a padding block (e.g., using whitespaces) before the new *Xref table* to fill the unused space.

Step 4: The attacker injects malicious objects which are not protected by the signature. There are different injection points for these objects. They can be placed *before* or after the malicious *Xref table*. If Step 1 is not executed, it is only possible to place them *after* the malicious *Xref table*.

Step 5 (optional): Some PDF viewers need a *Trailer* after the manipulated *Xref table*, otherwise they cannot open the PDF file or detect the manipulation and display a warning message. Copying the last *Trailer* is sufficient to bypass this limitation.

Step 6: The attacker moves the signed content defined by `c` and `d` at byte offset `c*`. Optionally, the moved content can be encapsulated within a stream object.

Noteworthy is the fact that the manipulated PDF file does not end with `%%EOF` after the `endstream`. The reason, why some validators throw a warning that the file was manipulated after signing, is because of an `%%EOF` after the signed one. To bypass this requirement, the PDF is not correctly closed. However, it will be still processed by any viewer.

4 Countermeasures

In this section, we propose concrete countermeasures fixing the previously introduced attacks. We carefully studied the main reasons for the attacks on PDF signatures. We determined two root causes: (1) The specification does not provide any information with concrete procedure on how to validate signatures. There is no description of pitfalls and any security considerations. Thus, developers must implement the validation on their own without a best-common-practice information. (2) The error tolerance of the PDF viewer is abused to create non-valid documents bypassing the validation, yet correctly displayed to the user.

The Verification Algorithm. Considering a proper countermeasure, we defined an algorithm which addresses USF, ISA, and SWA but does not negatively affect the error tolerance of the PDF viewers. It describes a concrete approach on how to compute the values necessary for the verification and how to detect manipulations after signing the PDF file. The specified algorithm must be applied for each signature within the PDF document. As an input, it requires the PDF file as a byte stream and the signature object.

```
INPUT: PDFBytes, SigObj

// ByteRange is mandatory and must be well-formated
byteRange = SigObj.getByteRange

// Preventing USF:
if (byteRange == null OR byteRange.isEmpty) return false

// Parse byteRange
if (byteRange.length≠4) return false
for each x in byteRange { if $x \neq$ instanceof(int) return false}
a, b, c, d = byteRange
// BytRange must cover start of file
if ($a \neq 0$) return false;
// Ensure that more than zero bytes are protected in hashpart1
if ($b \leq 0$) return false
// Ensure that sencond hashpart starts after first hashpart
if ($c \leq b$) return false
// Ensure that more than zero bytes are protected in hashpart2
if ($d \leq 0$) return false
// Preventing ISA. ByteRange must cover the entire file.
if ($(c + d) \neq PDFBytes.length$) return false;

```

```
24 // The pkcs7 blob starts at byte offset (a+b) and goes to offset c
25 pkcs7Blob = PDFBytes[(a+b):c]
26 // Preventing USF. Pkcs7Blob value is not allowed to be null or empty.
27 if (pkcs7Blob == null OR pkcs7Blob.isEmpty) return false
28 // pkcs7Blob must be a hexadecimal string [0-9,a-f,A-F]
29 if (pkcs7Blob contains other chars than [0-9,a-f,A-F]) return false
30
31 // Parse the PKCS\#7 Blob
32 sig, cert = pkcs7.parse(pkcs7Blob)
33
34 // Select (a+b) bytes from input PDF begining at byte a=0, i.e. 0 ... a+b-1
35 hashpart_1=PDFBytes[a:(a+b)]
36
37 // Select (c+d) bytes from input PDF begining at byte c, i.e. c ... c+d-1
38 hashpart_2=PDFBytes[c:(c+d)]
39
40 // Verify signature
41 return pkcs7.verify(sig, cert, hashpart_1||hashpart_2)
```

Listing 4.1: Pseudo-code preventing USF, ISA and SWA.

In Line 4, we first extracts the ByteRange from the signature object. For preventing USF, we ensure that ByteRange is not null or empty in Line 7.

Lines 9-22 then validate the values a, b, c, d of the ByteRange. First, Line 10 ensures that it contains exactly four values in order minimize an attacker's attack surface. Line 11 additionally ensures that each ByteRange value is an integer. Lines 14 to 20 ensure that ByteRange satisfies the following condition: $0 = a < b < c < (c + d)$, which is equivalent to $a = 0$ *and* $b > 0$ *and* $c > d$ *and* $d > 0$. Enforcing this condition ensures that the signature always covers the beginning of the file ($a = 0$), prevents signed blocks of length zero ($b > 0$, $d > 0$), and ensures that both signed blocks are non-overlapping ($c > b$). Finally, we verify that ByteRange covers the entire file (Line 22) in order to detect ISA.

Lines 24-29 parse the Contents parameter of the signature object, which is a PKCS#7 blob. The important aspect is that we interpret everything that is not covered by the ByteRange as the Contents parameter of the PDF signature. Theoretically, the check in Line 27 should never fail, because we previously verified $(a + b) = b < c$, thus it holds that pkcs7Blob.length > 0. Nevertheless, we leave this line here due to its importance for preventing SWA. Line 29 additionally ensures that only hex characters can be in the unprotected part of the PDF file, preventing further unwanted modifications of the file.

Lines 31-32 parse the PKCS#7 blob and extract the information to be used for the signature verification.

Lines 34-38 determine the bytes of the input PDF that are signed.

Finally, Line 41 calls the PKCS#7 verification function and returns the validity status of the signature.

Drawback. Specifying the algorithm in Listing 4.1 requires a change in the PDF specification which defines `ByteRange` as an optional parameter[**?** , Section 8.7]. In this case, the signature value will be computed only over the signature dictionary leaving the entire document unprotected. Such a feature allows an even more powerful attack since the attacker can create validly signed documents by only injecting the signed signature dictionary without a `/ByteRange`. Currently, none of the evaluated viewers supports this feature.

Additionally, the algorithm leads to one usability issue if multiple signatures are provided. Although these signatures are valid, only the one covering the entire document will be displayed as valid. This problem can be addressed by providing additional information to the user that some of the signatures are valid but cover only a specific revision and not the entire document. Adobe uses a similar approach for the signature validation. All Adobe viewers show information about the document revision protected by a signature and allow only to open this revision. Thus, a user can easily verify which information is signed and which is not.