Shadow Attacks: Hiding and Replacing Content in Signed PDFs

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

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

Simon Rohlmann
Ruhr University Bochum
simon.rohlmann@rub.de

***Abstract*—Digitally signed PDFs are used in contracts and invoices to guarantee the authenticity and integrity of their content. A user opening a signed PDF expects to see a warning in case of *any* modification. In 2019, Mladenov et al. revealed various parsing vulnerabilities in PDF viewer implementations. They showed attacks that could modify PDF documents without invalidating the signature. As a consequence, affected vendors of PDF viewers implemented countermeasures preventing *all* attacks.**

This paper introduces a novel class of attacks, which we call *shadow* attacks. The *shadow* attacks circumvent all existing countermeasures and break the integrity protection of digitally signed PDFs. Compared to previous attacks, the *shadow* attacks do not abuse implementation issues in a PDF viewer. In contrast, *shadow* attacks use the enormous flexibility provided by the PDF specification so that *shadow* documents remain standard-compliant. Since *shadow* attacks abuse only legitimate features, they are hard to mitigate.

Our results reveal that 16 (including Adobe Acrobat and Foxit Reader) of the 29 PDF viewers tested were vulnerable to *shadow* attacks. We introduce our tool *PDF-Attacker* which can automatically generate *shadow* attacks. In addition, we implemented *PDF-Detector* to prevent *shadow* documents from being signed or forensically detect exploits after being applied to signed PDFs.

I. Introduction

Digital signatures can protect Portable Document Formats (PDFs) against manipulations. This feature enables use cases such as signing contracts, agreements, payments, and invoices. Regulations like the eSign Act in the USA [1] or the eIDAS regulation in Europe [2] facilitate the acceptance of digitally signed documents by companies and governments. Asian and South American countries also accept digitally signed documents equivalent to manually signed paper documents [3]. Adobe Cloud, a leading online service for signing PDF documents, provided 8 billion electronic and digital signature transactions in 2019 [4]. In the same year, DocuSign processed 15 million documents each day [5].

a) Signed PDFs prepared by single entities: One typical use case of PDF signatures is that one in which a *single* entity creates both the PDF document and the signature. Invoices created by Amazon are a popular example of this scenario.

Figure 1. A *shadow* PDF document presents a trustworthy content to the signers (top document). After signing this document, the attackers modify the document and enforce another view of the document on victims' side without invalidating the signature (bottom document).

b) Signed PDFs created by multiple entities: Another typical use case is the signing of a contract. For example, this is the case for EU grant agreements, where the European Research Agency and the grant recipients have to sign a PDF document. We can describe the generic process of digitally signing a contract as follows: The collaborators first prepare the PDF contract. Collaborators can be lawyers, designers, or members of different companies. Once they have they finalized the PDF document, the involved parties then digitally sign the contract. The parties sign the PDF sequentially, and the PDF may be exchanged multiple times between the parties.

c) Security of PDF Signatures: In 2019, a comprehensive analysis of the security of digitally signed PDFs revealed severe flaws in multiple applications and found almost all of them to be vulnerable [6]. They used an attacker model in which the attacker possesses a PDF that has been digitally signed by a third party and manipulates it *after* the signature had been added to the document. The vendors have fixed these issues in their recent PDF viewer versions.

In this paper, we investigate the security of these patched versions of PDF viewers. We extend the attacker model

Network and Distributed Systems Security (NDSS) Symposium 2021
21-24 February 2021
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24117
www.ndss-symposium.org

from Mladenov et al. [6] and assume the attacker can place content of his own choice into the PDF file *before* it is signed. This assumption is based on real-world usage of signed PDFs by multiple entities. For instance, the attackers[1] may prepare a PDF document containing seemingly harmless content. They proceed by replacing this content after the document has been signed, see Figure 1. We answer the following research question:

> *Can the visible content of a digitally signed PDF be altered without invalidating a signature if attackers manipulate the PDF before it is signed?*

d) Shadow Attacks: In the analog world, a signer typically adds a handwritten signature at the end of the document. This addition at the end has two major downsides: 1) it is possible to exchange all pages before the signed page with arbitrary content. 2) Attackers could use empty spaces on signed pages to print new content, or they could overpaint existing content. These manipulations are impossible when using digital signatures because this type of signature protects the entire content. So it is assumed that transferring such an attack from the analog world to digital signatures is impossible.

This paper shows that this assumption is false by introducing a new attack class: *shadow* attacks. The idea of *shadow* attacks is that the attackers create a PDF document with two different contents: 1) content expected by the authority reviewing and signing the PDF and 2) hidden content that attackers can reveal after the PDF is signed. In Figure 1, an overview of the attack is shown. The attackers prepare a *shadow* document. In the analog world, this is the step in which the attackers could explicitly leave empty spaces. The *Signers* of the PDF receive the document, review it, and sign it. The attackers use the signed document, modify it, and send it to the victims. In the analog world, the attackers can print their content on the prepared empty spaces. After opening the signed PDF, the victims' PDF viewer successfully verifies the digital signature. However, the victims see different content than the *Signers*. We introduce three variants of the *shadow* attacks, which allow attackers to *hide*, *replace*, and *hide-and-replace* content in digitally signed PDFs. The *shadow* attacks do not rely on a dynamic content replacement. For example, we do not use JavaScript or content loaded from external resources that can be modified after signing the PDF. We consider such attacks trivial, and according to our observations, all viewers already prevent such attacks by warning the user.

e) Automatic Generation and Prevention: To contribute to future research, we present two tools: *PDF-Attacker* and *PDF-Detector*. Both tools are written in Python, and published as open source on https://pdf-insecurity.org. *PDF-Attacker* automatically generates a shadow document by using arbitrary files as an input. After the document is signed, *PDF-Attacker* executes the manipulation steps automatically and stores the manipulated file. *PDF-Detector* detects *shadow* attacks at both stages of their execution: before the file is signed and after the final manipulations. Thus, PDF readers can use *PDF-Detector* to refuse signing *shadow* documents and thus prevent harm. To facilitate the forensic analysis of signed PDF files, *PDF-Detector* can also analyze signed files and detect manipulations made afterwards.

f) Shadow Attack vs. Previous Attacks: On an abstract level, the *shadow* attacks resemble the idea of Incremental Saving Attacks (ISAs) [6]. Both attacks allow the manipulation of digitally signed PDFs without raising any warnings or errors. Both of them abuse a PDF feature called Incremental Update. Incremental Update allows changing the content of a PDF by appending a document modification to the file. However, there are essential differences between the *shadow* attacks and ISAs. ISAs manipulate a PDF by appending a *malformed* Incremental Update, wherein objects are missing or not closed properly. This approach was motivated by viewers providing either a denylist or allowlist of potentially dangerous objects. Based on malformed Incremental Updates or missing dangerous objects in the deny/allowlists, Mladenov et al. [6] were able to bypass the verification of multiple viewers. As a result, the PDF viewers extended the lists with potentially dangerous objects, improved the verification to detect malformed Incremental Updates, and warned users in the event of inconsistencies.

In contrast to previous attacks, our *shadow* attacks do not use a malformed Incremental Update, but *instead are standard-compliant and use well-formed* Incremental Updates. Thus, no inconsistencies in the file structure exist. The *Hide* and *Hide-and-Replace* variants also bypass even perfectly implemented denylists or allowlists. Thus, none of the currently implemented countermeasures which detect malicious Incremental Updates prevents *shadow* attacks.

g) Results: We show the applicability of the *shadow* attacks by evaluating 29 PDF applications and revealing vulnerabilities in 16 of them, including Adobe Reader and Foxit Reader. Moreover, we achieve a privilege escalation on Adobe products allowing the attackers to execute high privileged actions on victims' computers.

h) Contributions: This paper makes the following key contributions:

- We introduce an attacker model that is based on real-world scenarios and allows an attacker to place shadow content into a PDF before it is signed (section III).
- We are the first to present the *shadow* attack class on PDF signatures. We found three different variants that allow the ability to *hide*, to *replace*, and to *hide-and-replace* content without invalidating the signature validation status of a digitally signed PDF (section IV).
- We implemented *PDF-Attacker*, an open-source Python toolset for automatic exploit generation (section VI).
- We show the impact of *shadow* attacks by breaking 16 of 29 PDF applications (see section VII).
- We implemented and evaluated *PDF-Detector*, an open-source shadow attack prevention and detection tool (section VIII).
- We apply *shadow* attacks beyond signed PDFs and reveal a critical code execution vulnerability in Adobe Reader (section IX).

i) Responsible Disclosure: We responsibly disclosed all issues to the respecting vendors. Therefore, we cooperated with the CERT-Bund (BSI) and provided a dedicated vulnerability report, including all exploits, to them. They kindly

[1] In this paper, we use the gender-neutral pronoun *they* for the following entities: victim, attacker, signer, and user.

created the initial contact with all vendors and managed the distribution of the report. In the case of technical queries, we directly supported the vendors to understand and fix the issues. Some of the vendors contacted us regarding a re-test of their countermeasures, which we also provided.

II. Basics

a) PDF File Structure: The Portable Document Format (PDF) is a platform-independent document format. It consists of three main parts, as depicted in Figure 2.

The **first part** defines the *PDF body*. It contains different objects, which are identified by their object number. The most important object is the root object, which is called the `Catalog`. In Figure 2, the `Catalog` has the object identifier `1 0`. The `Catalog` defines the whole PDF structure by linking to other objects in the body. In the example given, the `Catalog` links to a form object `AcroForm`, to a PDF `MetaData` object, and to actual PDF `Pages` object. The `Pages` object can reference multiple `Page` objects, which in turn reference, for example, the actual `Content`, `Font`, and `Images` objects. These object references are technically implemented by using a dedicated reference string based on object numbers. For example, the `Pages` object references the `Page` object by using the reference `5 0 R`. The **second part** of the PDF is the *Xref table*. It contains references to the byte positions of all objects used in the PDF body. Objects that are not *in use* can be explicitly flagged as *free* in the *Xref table*. For example, the image object `9 0` is free and not displayed in the PDF. Although flagged as free, the entry in the *Xref table* for object `9 0` can contain the byte position of the free object. The **third part** is the *Trailer*. It consists of two further references: one to the byte position at which the *Xref table* starts, and another link to the identifier of the root object (`1 0`).[2]

b) Incremental Update: The content of a PDF may be updated for different reasons, for example, by adding review comments or by filling out PDF forms. From a technical perspective, it is possible to add this new content directly into the existing PDF body and add new references in the *Xref table*. However, this is not the case according to the PDF specification. Changes to a PDF are implemented using Incremental Updates.

An Incremental Update adds new objects into a new PDF body, which is directly appended after the previous *Trailer*. To adequately address the new objects, a new *Xref table* and *Trailer* are also appended as well for each Incremental Update. Summarized, a PDF can have multiple bodies, *Xref table*s, and *Trailer*s, if Incremental Update is applied.

c) PDF Signature: For protecting the integrity and the authenticity of a PDF, digital signatures can be applied. For this purpose, a `Signature` object is created and appended to the PDF by using Incremental Update. It is also possible to sign a PDF multiple times (e.g., a contract), resulting in multiple Incremental Updates. The `Signature` object contains all relevant information for validating the signature, such as the algorithms used and the signing certificate. It also defines which bytes of the PDF are protected by the signature, that is, which bytes are used to compute the cryptographic hash that the signature

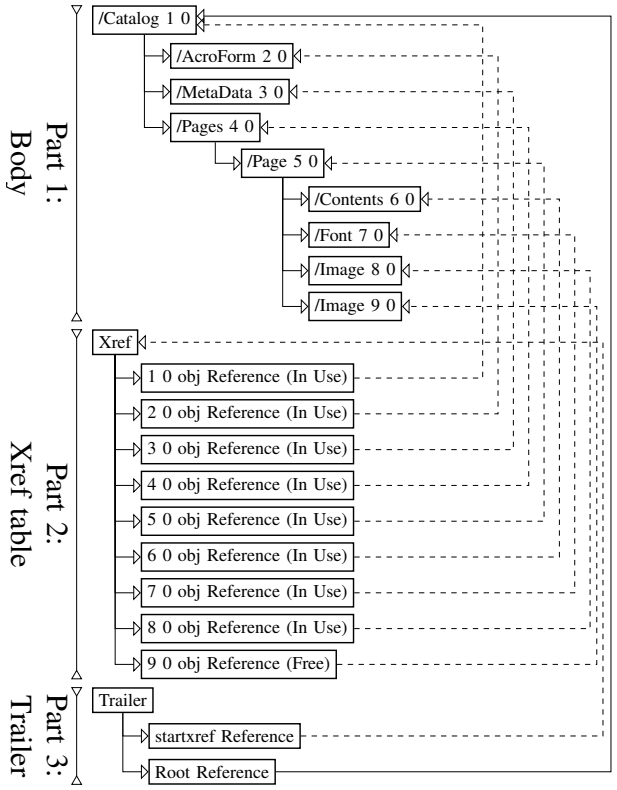


Figure 2. A PDF consists of three parts: body, *Xref table*, and *Trailer*. Solid-lined arrows indicate direct object references. Dashed-lined arrows indicate byte offset references.

algorithm uses. A typical signature starts at the first byte and ends at the last byte of the trailer.[3] Once a user opens a PDF containing a PDF signature, the viewer application automatically validates the signature and it provides a warning if the content has been modified.

d) Incremental Update on Signed Documents: Even on a signed PDF, a further Incremental Update can be applied. Examples are review annotations or additional signatures. Since such Incremental Updates are appended to the signed document, and no changes within the signed area are made, the signature remains valid.

In 2019, Mladenov et al. [6] showed that an Incremental Update can change the presentation of the entire signed document. As a countermeasure, the authors recommended letting the viewer raise a warning if the PDF provides content outside the signature's scope. However, this countermeasure is not standard compliant. There are legitimate use cases where an Incremental Update should not lead to a warning, for example, a second digital signature. Thus, the vendors implemented a different countermeasure by creating a list of potentially dangerous elements forbidden within an Incremental Update. Currently, the viewers search for such elements within an Incremental Update and throw a warning on a match.

In this paper, we focused on the elements which viewers consider *harmless* within an Incremental Update. We show that

[2]The root element does not need to have the identifier `1 0`.

[3]For technical reasons, there is a gap inside this range that is unprotected. It contains a PKCS#7 blob of the signature itself.

attackers can still change the signed document's presentation by neither invalidating the signature nor raising any warnings.

III. Attacker Model

The attacker model is based on real-world use cases in which a PDF document, for example, a contract, is signed. In these use cases, attackers can inject *invisible* parts ("shadow content") into a PDF *before* it is signed. After the signing, the attackers again manipulate the signed PDF. Thereby, they enforce a *visible* change in its content without invalidating the signature.

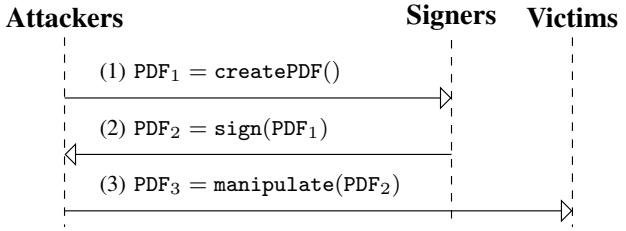


Figure 3. Attacker Model: The attackers prepare the *Shadow Document* ($\texttt{PDF}_1$) which the *Signers* sign ($\texttt{PDF}_2$). Afterward, the *Attackers* modify the content of the signed PDF ($\texttt{PDF}_3$) and send it to the *Victims*.

a) Attacker Capabilities: As shown in Figure 3, the attacker capabilities can be divided into three phases. The output of each phase is a PDF file.

1) The attackers create the PDF document $\texttt{PDF}_1 = \texttt{createPDF}()$ that contains the invisible shadow content (e.g., a text or an image).
2) The signers receive $\texttt{PDF}_1$ (e.g., by email) and create a new document $\texttt{PDF}_2$ by signing $\texttt{PDF}_1$, i.e. $\texttt{PDF}_2 = \texttt{sign}(\texttt{PDF}_1)$.
3) The attackers receive $\texttt{PDF}_2$. They can modify $\texttt{PDF}_2$ again, for instance, the attackers create $\texttt{PDF}_3 = \texttt{manipulate}(\texttt{PDF}_2)$. The attackers send $\texttt{PDF}_3$ to the victims.

The main difference to the previous work [6] is that the attackers are allowed to embed malicious content *before* the PDF is signed instead of solely modifying the PDF after the signature has been applied.

b) Winning Conditions: The attackers are successful (●) if the following conditions are fulfilled:

1) The signers only sign $\texttt{PDF}_1$ if they do not notice of the shadow content. In other words, all changes injected by the attackers must be *invisible* to the signers.
2) The victims see the shadow content once they open $\texttt{PDF}_3$.
3) The signature verification of $\texttt{PDF}_3$ is successful. The victims trust the signers' public key. The victims do not trust any other key. In particular, they do not trust the attackers' key.
4) Opening $\texttt{PDF}_3$ does not show any errors or warnings, for example, due to a malformed file format.

Some PDF viewer show a *warning* even if it validates the unmanipulated $\texttt{PDF}_2$. If the signature validation of the unmanipulated $\texttt{PDF}_2$ and the manipulated $\texttt{PDF}_3$ show exactly the same *warnings*, we call the attackers' success *limited* (◐).

IV. Shadow Attacks: Overview and Preliminaries

The central concept of *shadow* attacks is that the attackers prepare a PDF document by injecting invisible content – "*shadow* content". We call this prepared PDF a "*shadow* document". Afterward, the signing entity, for example, a person or an online signing service, receives the *shadow* document, signs it, and sends it back to the attackers. Despite the integrity protection provided by the digital signature, the attackers can modify the signed *shadow* document and change the *shadow* content's visibility. Nevertheless, the manipulation is not detected, and the digital signature remains valid. Finally, the attackers send the modified signed *shadow* document to the victim. Although the attackers altered it, the signature validation is successful. However, the victims see different content than the signing entity. That is, the victims see the *shadow* content.

A. Shadow Documents in the Real World

Considering the applicability of *shadow* documents, we focus on the following two questions: (1) How can the attackers force the signing of a *shadow* document? (2) Why are the attackers capable of modifying a signed *shadow* document?

a) Signing a Shadow Document: In companies and authorities, relevant documents like contracts or agreements are often prepared by the employees, which take care of most of the details and technicalities. An authorized person then signs the document after a careful review. Another scenario is the signing process of a document within a consortium. Usually, one participant creates the final version of the document, which is then signed by all consortium members. Considering the given examples, a maliciously acting employee or consortium member can inject invisible *shadow* content during the editing. Consequentially, this content will be signed.

Additionally, multiple cloud signing services like Adobe Cloud, DocuSign, or Digital Signature Service exist. Among other functionalities, such services receive a document and sign it. Such services can also be used to sign *shadow* documents.

b) Manipulating a Shadow Document: One can assume that a signed Portable Document Format (PDF) document cannot be changed and that it is final. This assumption is not the case due to the desired features like multiple signatures or annotations. For example, a PDF document can be signed multiple times. This process is essential in many use cases since it allows stakeholders within a consortium to have a single document containing the signatures from all partners. From a technical perspective, each new signature appends new information to the already signed document (see paragraph II-0b). Nevertheless, the document should still be successfully verified for each signature. Additionally, the PDF specification defines interactive features like annotations (e.g., sticky notes and text highlighting). Since annotations do not change the content but only put remarks on it, these changes are considered harmless. Thus, the PDF specification allows the injection of seemingly harmless objects in a signed file without invalidating the signature.

B. Analysis of Document Modifications

Currently, PDF applications analyze the changes made after signing and try to estimate if these changes are legitimate. For

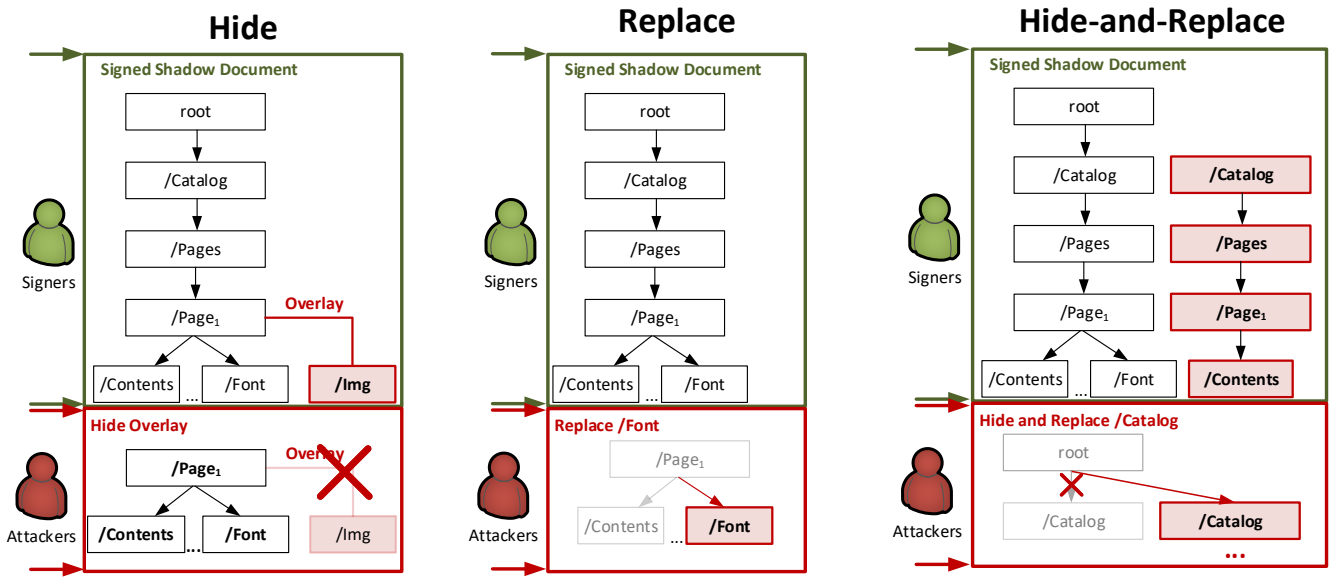


Figure 4. We show three variants of manipulating a *shadow* PDF document without being detected: *Hide*, *Replace*, and *Hide-and-Replace*.

instance, overwriting content on a page of the document is not allowed, leading to invalid signature verification. Such attacks were evaluated in 2019 by Mladenov et al. [6].

In this paper, we first analyzed which changes are considered harmless by the PDF applications and abused these to exchange the entire content within a PDF document. None of the previous work provides such an in-depth analysis. Thus, a gap concerning the possible manipulations existed. The allowed changes can be summarized as follows.

a) Appending new Xref table *and* Trailer*:* Appending a new *Xref table* and *Trailer* occurs on each change on PDF documents. For instance, for each signing process using the signature information, a new *Xref table* and *Trailer* are generated. Thus, appending these at the end of the file is considered harmless.

b) Overwriting Harmless Objects: In their paper, Mladenov et al. [6] were able to append new objects beyond the signed document by overwriting existing objects and thus replacing the content. The attack was called an Incremental Saving Attack (ISA). Nevertheless, the authors considered only object types: `Catalog`, `Pages`, `Page`, and `Contents`. This is reasonable since these objects directly influence the content shown by opening the document. The applications' vendors fixed the vulnerabilities by detecting the definition of such objects after the signature was applied. Inspired by Markwood et al. [7], we considered the definition of further objects like fonts or metadata, which also influence the presented content.

c) Overlapping Objects: During our analysis, we raised the question regarding the visible presentation of overlapping content. More precisely: "If two objects share the same position on a page, which object shows the application in the foreground and which one in the background?". We determined that the declaration of the object within the document is decisive. In the case of overlapping, the first object is displayed on top of the second one. Thus, we can append the same objects to a PDF file but in a different order. Since the content of the objects is not changed, this Incremental Update is also considered harmless. Nevertheless, the visible content changes when opening the PDF file.

d) Changing Interactive Forms: We observed an unexpected feature applied to interactive forms, which overlays the content of a text field. By clicking on the text field, its content is shown, and the overlay disappears. While we avoid a discussion regarding the usefulness of this feature, we observed that changes on the overlay are considered harmless and do not invalidate the signature.

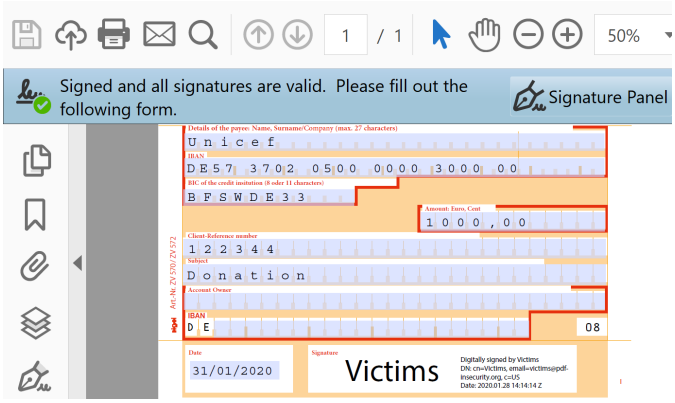
C. Summary

The PDF specification defines a compromise between usability and security by softening the rules regarding the integrity protection of digitally signed documents. This means that signed PDF documents can be extended by applying Incremental Updates. Attackers can inject content within the Incremental Update that is appended to the end of the signed document. Since PDF signatures are computed on a fixed range of bytes of the PDF file, the Incremental Update is outside of that range, and it does not violate that cryptographic protection. By defining exceptions of allowed and forbidden changes, the developer teams are responsible for the detection and classification of dangerous elements within each Incremental Update. Wrong decisions lead to vulnerabilities. In the next section, we show how changes that are classified as harmless can enable the exchanging of content without invalidating the signature.

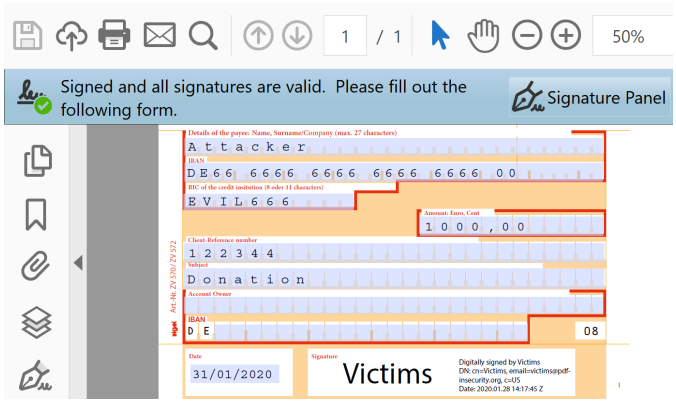
V. Shadow Attacks: Hide, Replace, and Hide-and-Replace

In this section, we present three different classes of *shadow* attacks: *Hide, Replace*, and *Hide-and-Replace*. Each attack class introduces a different technique to stealthily manipulate a signed PDF without causing any warnings or exceptions during its signature validation.

Each attack is based on two manipulation steps made by the attackers as depicted in Figure 3. In Step 1, the attackers



(a) A *shadow* PDF document digitally signed by the victims containing a donation amount.



(b) Manipulated PDF document after signing which contains attackers' account data (top row).

Figure 5. Form-based Attack. On the left side, the victims sign a donation to a non-profit organization. On the right side, the attackers manipulate the signed document to display different bank account information. The validity status of the digital signature remains untouched. Apart from the account information, both documents are indistinguishable.

prepare the document by injecting the *shadow* content. This *shadow* document is sent to the signers. In Step 2, the attackers receive the signed document and make the *shadow* content visible. This document is sent to the victims.

All in all, we created eight different exploits covering all attack variances. In the following sections, we explain the idea of each attack and its execution.

A. Shadow Attack: Hide

This class of *shadow* attacks aims to hide the content relevant to the victims behind a visible layer. For example, the attackers can hide the text "You are fired!" behind a full-page picture showing "Sign me to get the reward!". Once the attackers receive the signed document, they manipulate the document so that the viewer application no longer renders the picture.

Hide attacks have two advantages from the attackers' perspective:

1) Many viewers show warnings if new visible content is added using Incremental Update. However, they do not warn in most cases if content is removed.
2) The objects are still accessible within the PDF. In the example above, the text "You are fired!" can still be detected by a search function. This detection might be necessary if an online signing service is used, and it reviews the document by searching for specific keywords.

We identified two variants of this attack class, which are explained further.

1) Variant 1: Hide Content via Referenced Object: In this attack variant, the attackers create overlay objects such as images or form fields and hide them after the document is signed to reveal the content below these objects. We created three different exploits that hide content via malicious image, hide form fields via malicious form fields, and hide content via malicious form fields.

a) Step 1 – Injecting the shadow content: As shown in Figure 4, the attackers inject one or multiple images and place them over the original content. The images could overlay an entire page or only parts of the content, for example, a digit or passage of text. The attackers entirely control the position and visibility of the placed image.

b) Step 2 – Making shadow content visible: The simplest method for this is to create an Incremental Update, which only updates the *Xref table* by setting the overlay object to *free*. However, many viewers (e.g., Adobe) classify this change as dangerous and throw an error or a warning. For this reason, we use another approach: we use the same object ID within the Incremental Update, but we define it as a different object type. For example, we change the overlay type `Image` to `XML/Metadata`. Additionally, we added an *Xref table* update pointing to the metadata object but keeping the object ID of the overlay.

When opening this manipulated document the overlay is hidden because Metadata cannot be shown. Since adding Metadata to a signed PDF using Incremental Update is considered harmless, the signature remains valid.

Moreover, we observed that attackers could hide form fields if they changed their references to empty objects. To execute the attack, the attackers place the malicious form fields above the original ones in which the attackers place predefined values. The manipulated document is sent to signers. They only see the malicious form fields. After receiving the signed document, the attackers let the malicious fields disappear by referencing them to empty objects. In this way, the original form fields, including the attackers' predefined values, are shown to the victims.

2) Variant 2: Hide Content via Object's Order: During our analysis, we observed that for two different form fields with the same size and at the same x-y position within the document, only the last one is shown. Furthermore, the same form fields can be re-declared within an Incremental Update as long as none of the content changes. Based on both observations, the attackers can build the following exploit.

a) Step 1 – Injecting the shadow content: The attackers inject into the original unsigned document their *shadow* form

fields at the same x-y position as that of the content they want to hide, but they declare their form objects *before* the original ones. The signers see only the original form fields since they are defined after the *shadow* ones.

b) Step 2 – Making shadow content visible: After receiving the document, the attackers append an Incremental Update which copies and pastes the original and the *shadow* form fields. In this case, however, they first place the original and then the *shadow* form fields. As a result, the *shadow* form fields and their values are shown instead of the original ones. Since the objects themselves have not modified, but only their declaration order, the Incremental Update is considered harmless.

B. Shadow Attack: Replace

The main idea of this *shadow* attack class is to use an Incremental Update that directly changes previously declared objects. Since the modification is not allowed for all types of objects, the attacker only changes objects that are considered harmless but can nevertheless change the document's visible content. For instance, the (re)definition of fonts does not change the content directly. However, it influences the view of the displayed content and makes number or character swapping possible. We identified two variants of this attack class.

1) Variant 1: Replace via Overlay: PDF Forms support different input masks, such as text fields, text areas, and radio/selection buttons. Forms can have default values, for example, a predefined text. Users can dynamically change these values and store them in the PDF document.

The attack abuses a dedicated property of PDF text fields. A text field can show two different *values*: the real field value and an overlay value, which disappears as soon as the text field is selected. A form field's real value is contained in an object key named `/V`. The content of the overlay element is defined within a `/BBox` object. The `/BBox` object is comparable to the hint labels known from HTML forms. For example, the hint *username* indicates that the username should be entered into a specific login field. *In contrast* to HTML, in PDF there is no visual difference between the *hint* and the *actual* value. We depict an example attack in Figure 5.

a) Step 1 – Injecting the shadow content: First, the attackers create a transfer slip (PDF_1) containing an interactive form which the signers complete before signing the document. The attackers initialize some of the form elements with default values. In the example provided in Figure 5, the attackers set the values `/V` of the first three form fields to `Attacker` and the attackers' `IBAN` and `BIC`. Second, the attackers set the overlay values (`/BBox`) to `unicef` and the corresponding `IBAN` and `BIC`. As long as the signers do not focus on the prepared values, they believe that the correct values are already pre-filled.

b) Step 2 – Making shadow content visible: The signers sign the PDF without changing the pre-filled forms. Once the attacker receives PDF_2, they update the text fields by replacing the overlay stored in `/BBox` with different values. The values stored in `/V` remain unchanged. Viewers consider this replacement harmless since the original text field value is not changed but rather only the overlay.

Once the victims open PDF_3, the viewer first verifies if the values stored in `/V` within each text field have been changed and differ from the signed values. If the values have been changed and differ from the signed values, the signature validation fails. Since the attackers do not change any values stored in `/V`, the signature remains *valid*. The viewer then processes each text field object and shows the `/BBox` value if it maps to the signed one. Otherwise, the value stored in `/V` is presented. Since the attackers change the `/BBox` value, the value `/V` (being `Attacker`) is shown, and the corresponding malicious transaction slips through.

As a result, the signers and the victims have different views on the same document, which should be prevented by the digital signature. For each attack variant, we create one exploit.

2) Variant 2: Replace via Overwrite: The idea of the attack is based on the ISA described by Mladenov et al. [6]. Consequently, the vendors implemented a list of objects considered dangerous and disallowed their occurrence in Incremental Updates. However, in many applications, fonts are considered harmless, and thus, they can be defined within an Incremental Update. This attack variant proves the opposite.

a) Step 1 – Injecting the shadow content: The attackers analyze the fonts used in the original document and distillate which are relevant for the content. Second, default fonts like *Verdana* or Times New Roman are usually not included in the PDF. In this case, the attackers need to inject the font description as shown in Figure 4.

b) Step 2 – Making shadow content visible: After the document is signed, the attackers append a new font description and overwrite the previous one. The new font description completely changes the presentation of the original text. For example, we created an exploit changing the presentation of the original text *US90 5628 3174 5628 3174* to *US01 2345 6789 2345 6789*. Since the definition of new fonts is considered harmless, the applications verifying the signature do not warn of the changes made. A popular software to create a malicious font description is FontForge.[4]

C. Shadow Attack: Hide-and-Replace

In this *shadow* attack class, the attackers create a *shadow* PDF document which is sent to the signers. The PDF document contains a hidden description of another document with different content. Since the signers cannot detect the hidden (malicious) content, they sign the document. After signing, the attackers receive the document and solely append a new *Xref table* and *Trailer* that enables the hidden objects.

We identified two variants of this attack class. Both variants differ in the way the attackers enable the hidden content after the document had been signed. For each attack variant, we created one exploit.

1) Variant 1: Change Object References: The idea of this attack variant is to use the *Xref table* for changing the reference to the document's `Catalog` (or any other hidden object) to point to the *shadow* document. In Figure 6, an example of the attack is depicted and will be explained further.

a) Step 1 – Injecting the shadow content: The attackers create a PDF file containing two objects with the same object ID (e.g., `4 0 obj`) but different content: "Sign the document

[4]https://fontforge.org/en-US/

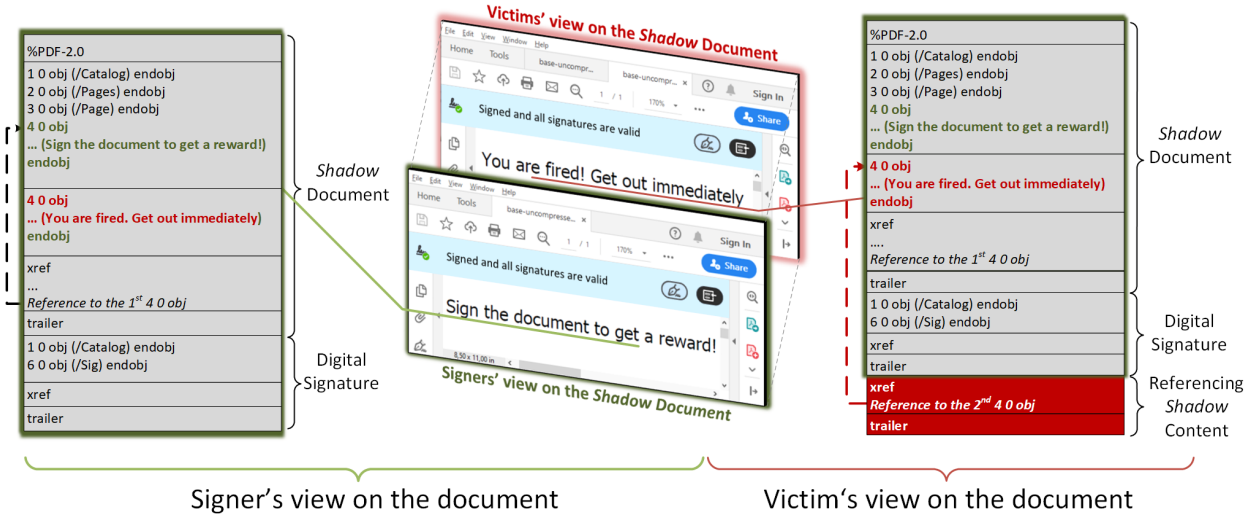


Figure 6. The attackers successfully manipulate a signed document and force different views on the *signers* and the *victims* by using the *Hide-and-Replace* attack variant.

to get a reward!" and "You are fired. Get out immediately". As shown on the left side in Figure 6, within the *Xref table* section, the seemingly harmless content is referenced. The signers only see this content and sign the PDF file.

b) Step 2 – Making shadow content visible: After receiving the signed PDF, the attackers append a new *Xref table* and exchange the reference to the object (e.g., `4 0 obj`) with the malicious content "You are fired. Get out immediately". A new *Trailer* is also appended. Since the inclusion of an *Xref table* pointing to an already defined object within the signed area is considered harmless, there is no warning of the changes made. The signature verification is successful. Nevertheless, the victims see different content than what the signers see.

2) Variant 2: Change Objects Usage: The idea of this attack variant is again to use the *Xref table*. However, instead of changing the reference to the objects, the attackers specify which objects are "in use" and which are not used (i.e., "free"). Using this *Xref table* feature enables attackers to modify the visibility of previously included objects. By this means, attackers can hide "in use" objects and show "free" objects. This is possible without changing the objects themselves. The attackers only touch the *Xref table*, but the signed document's presentation can be changed entirely.

a) Step 1 – Injecting the shadow content: Similar to the first attack variant, the attackers insert the malicious content which is correctly referenced but marked in the *Xref table* as *not in use*. Thus, only the content of the original document is shown to the signers.

```
%%% Xref table in the document sent to the signers. %%%
% Original Xref table
xref                                  % start of the Xref table
1 8                  % 8 objects starting with the object Id 1
0000000010 00000 n          % Object 1 at offset 10 is in use
0000000099 00000 n          % Object 2 at offset 99 is in use
...                                 % Further object references

% Injection point: new malicious but hidden objects
9 1                   % 1 object starting with the object Id 9
0000006666 00000 f          % Object 9 at offset 6666 is free
```

Listing 1. The attackers manipulate the original document by injecting new objects. In the given example, this is the `9 0 obj`. The attackers hide this object by disabling its usage via the *Xref table*.

b) Step 2 – Making shadow content visible: Once, the attackers receive the signed manipulated document, they append a new *Xref table*. The new *Xref table* enables the hidden content and disables the original one.

```
%%% Xref table in the document sent to the signers. %%%
% Original Xref table
xref                                  % start of the Xref table
1 8                  % 8 objects starting with the object Id 1
0000000010 00000 n          % Object 1 at offset 10 is in use
0000000099 00000 f          % Object 2 at offset 99 is free
...                                 % Further object references

9 1                   % 1 object starting with the object Id 9
0000006666 00000 n          % Object 9 at offset 6666 is in use
```

Listing 2. The attackers manipulate the *signed* document by appending the following *Xref table*. In the given example, the object with Id *2* defining the content of a page is disabled and the object with Id *9* is enabled and thus visible.

D. Stealthiness of Shadow Attacks

Shadow attacks require interactions with the signers and the victims. Thus, the attackers must create the document so that neither of the two entities becomes suspicious by merely opening and reviewing the document. This is truly the case for *all* attack variants. However, further actions like text selection, copy-pasting text, or searching within the document might expose the attack's stealthiness.

For all attack classes, we require that the victim cannot detect the *shadow* content in *any way*. Thus, we concentrate on cases in which the signers might detect the attacks.

The *Hide-and-replace* class is entirely concealed from the signers. From the signers' perspective, there is no possibility of detecting the *shadow* content. Variant 2 of the attack is restricted only to form fields since its disappearance is considered harmless by many viewers.

Hide Variant 1 (Hide via Referenced Object) might be detected by searching for a specific text behind the overlay or selecting the overlayed content. With respect to this restriction, the overlayed content can be only a number or a text area that makes the attack hard to detect. Considering *Variant 2 (Hide via Object's Order)* the attack is entirely concealed for form fields since we can hide previously shown fields.

Concerning *Replace Variant 1 (Replace via Overlay)*, the attack can be detected only if a form field is editable and the user clicks into the field. Noteworthy is that the attackers define the capabilities of the form fields and can always deactivate the editability. The *Replace Variant 2 (Replace via Object's Order)* can be detected by searching for the original content or copy-pasting the manipulated content. The copied text contains the original value.

E. Shadow Attack vs. Incremental Saving Attack

In this section, we highlight the differences between our Shadow attack and the attacks known as *ISA* [6] to avoid confusion between both attack concepts. First, we provide details regarding the ISA by analyzing all known and publicly available ISA attack vectors plus their corresponding countermeasures. Second, we explain why these countermeasures are insufficient to mitigate the Shadow attack.

a) Attack Vectors: ISA overwrites content objects directly or by using a malformed Incremental Update to bypass the protection mechanisms. We analyzed all available ISA attack vectors [8] and classified them into four categories: forbidden objects, invalid objects, missing *Xref table*, and missing *Trailer*. We estimated that none of the four categories is generic. Every category depends on the corresponding viewer and even on its version. Additionally, each of them interprets malformed objects and Incremental Updates differently. Finally, one can say that ISA is less generic and software-dependent.

In contrast, the *shadow* attack relies on a well-formed Incremental Update and thus does not depend on each viewer's specific interpretation, but on standard-compliant features.

Considering the creation of malicious PDF documents, ISA scales better than the *shadow* attacks. To carry out ISA, the attackers possess one signed file by a trusted authority, and they can create malicious PDF documents with any content. The attackers can display only content, which was hidden during the signing process and which is already part of the signed document. As a result, the amount of malicious PDF documents, which the attackers can create, is limited.

b) Attacker Model: Both attacks rely on different attacker models: ISA relies on an attacker possessing a digitally signed PDF document. The *shadow* attack additionally assumes that the attackers inject malicious content before the PDF is signed.

c) Countermeasures: We summarized the implemented ISA countermeasures in Table I.

The first countermeasure is the definition of *forbidden objects within an Incremental Update*, i.e., blacklisting the objects `/Pages`, `/Page`, `/Contents`. This is reasonable because each of these objects directly influences the presented content by opening a PDF. All other countermeasures target malformed Incremental Update.

Shadow Attacks	ISA Countermeasures			
	Forbidden Objects	Invalid Objects	Missing *Xref table*	Missing *Trailer*
Hide	●	●	●	●
Replace	◐	●	●	●
Hide-and-Replace	●	●	●	●

● Countermeasure insufficient ◐ Countermeasure partially successful ○ Countermeasure sufficient

Table I. EVALUATING THE ISA COUNTERMEASURES REVEALS THEIR INEFFECTIVENESS AGAINST THE *shadow* ATTACKS. NO CURRENTLY IMPLEMENTED COUNTERMEASURE IS SUFFICIENT.

Shadow attacks are not affected by any of these countermeasures since they do not rely on malformed Incremental Update. Only the *Replace via Overwrite* variant is restricted since the definition of a new font could be detected by extending the current lists with forbidden objects. Nevertheless, the *Replace via Overlay* is not affected. The *Hide* and *Hide-and-Replace* variants are always applicable as long as the viewers allow Incremental Update.

To summarize, PDF viewers have to choose between being standard-compliant (by allowing Incremental Update) and vulnerable, or being secure and not standard-compliant.

VI. PDF-ATTACKER

In this section, we present PDF-Attacker, a toolset that automatically creates shadow attack exploits.

a) Design of PDF-Attacker: PDF-Attacker is written in Python using Jupyter Notebooks. This design enables the high flexibility that is necessary to resemble the shadow attacks. For each attack variant of each *shadow* attack class, we created a separate Jupyter Notebook, so that all exploits can be investigated and extended independently.

Initially, our goal was to use a single Python PDF library for all attacks. It turned out that this is not ideal since every attack addresses different PDF features. For example, for attacks using forms, the *reportlab* library provides many useful features. In contrast, the *hide-and-replace* attacks require low level access to PDF objects, which is possible with *pypdf4*. In the end, we used different libraries for different attacks in order to maximize the functionality of the tool.

b) Configuration of PDF-Attacker: Before starting to work with PDF-Attacker some configuration steps need to be executed. The configuration steps can be summarized as follows:

- Content to manipulate: Independent of the attack variant, PDF-Attacker needs to know which content is in the attack scope. This could be an entire page, a field value, or a font description.
- *Shadow* content: Depending on the attack variant, the shadow content also needs to be prepared. This content could be an image overlapping some content, a malicious font, a malicious value, or an entire document with a specific content.
- Key material: Many PDF applications offer the ability to *digitally sign* a PDF only in the commercial version. Since we do not want to rely on an external software, we

decided to implement a signing module. The corresponding key material can also be specified by using different keys than those provided.

c) Exploit Generation with PDF-Attacker: The exploit generation with PDF-Attacker is separated in three phases, as depicted in Figure 7. In the first phase "Generate Shadow PDF", PDF-Attacker takes an arbitrary PDF as input and inserts the shadow elements according to the chosen attack variant. This phase is the most complex part of the attack. The reason is the complexity and flexibility of the PDF standard. The tool should be able to process different PDF versions, new features, and interactive elements. The tool should be find the relevant content within complex structures and place the attack vector in a usable way. We were able to reduce this complexity by using multiple libraries that parse the PDF files for us, and find the relevant content. The relevant libraries can be summarized as follows:

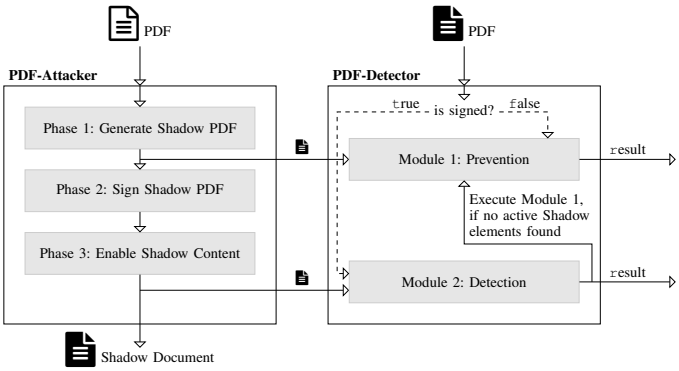


Figure 7. The **PDF-Attacker** takes an arbitrary PDF as input, builds-in the shadow objects (Phase 1), signs the document (Phase 2), and finalizes the attack by enabling the shadow content (Phase 3). The **PDF-Detector** is a tool to detect malicious documents generated in Phase 1 and Phase 3. It can also take an arbitrary PDF as input and is described in section VIII.

- *Hide*: The *wand* library allows the conversion of an arbitrary PDF to an image, which can be used as an overlay.
- *Replace*: The python libraries *reportlab* and *fitz* provide interfaces to work with forms and change their values. Some attack variants require low-level access to PDF source code to manipulate the appearance or exchange the existing fonts. For such cases, we use the *pypdf4* library in addition.
- *Hide-and-Replace*: The preparation of this attack requires creating a complete shadow document and the corresponding *Xref table*. Only the *pypdf4* library provides such a low-level interface allowing us to automate these steps. Some of the attack steps, however, are not supported by any library. Thus, we directly manipulated the PDF.

The "Sign Shadow PDF" phase. This step prepares the PDF that will be signed. We decided to simulate the signing process in Python, using the *endesive* library. This decision allows simulating both the preparation and the modification phase easily. Generally, this step could also be executed externally, for example, by using Adobe Acrobat to sign the prepared PDF. The "making shadow content visible" phase. In this step, the signed PDF is manipulated, so that the *shadow* content is shown.

d) Running PDF-Attacker: The deployment of PDF-Attacker is challenging because many Python libraries rely on external tools. For example, to convert an arbitrary PDF into a PNG, a dedicated *imagemagic* package must be installed, and a proper `policy.xml` must be configured. To minimize the effort of using or extending PDF-Attacker, we used VSCode with *remote docker containers* as deployment. By this means, using PDF-Attacker only requires VSCode and Docker. Everything else, including downloading all relevant packages and setting up the execution environment, is automatically configured.

e) Limitations: We are aware of the limitations concerning the PDF documents used as an input. Manipulations on encrypted documents are not supported. Also, documents having one or multiple Incremental Updates have not yet been tested. This limitation also includes documents that have already been signed and is a natural limitation due to the 1300-pages PDF specification's complexity.

Considering the *Replace via Overwrite* attack, we created one malicious font. Thus, only files having this font, can be attacked. This limitation can be circumvented by automatically extracting the fonts contained in a PDF file and using tools like FontForge [5] to generate malicious fonts on the fly. Due to the high complexity and variants of fonts, we considered this functionality out-of-scope.

Concerning the shadow content, we prepared proofs-of-concept for each attack variant. We allow variations regarding this content. However, more complex changes, including manipulations on multiple forms or pages, are not or only partially supported.

VII. Evaluation

In this section we present the results of our evaluation. The manipulated PDF documents created during the research were tested as black box procedures under all viewing applications listed in Table II.

A. Test Environment

Three computer systems were used for the simulation of the three entities *attackers*, *signers*, and *victims*. While the attackers' and signers' systems are based on Windows 10, we divided the victims' systems into Windows 10, macOS Catalina, and Ubuntu 18.04.3 LTS as a Linux distribution. Thus, we could test the effects of the manipulations on all standard operating systems. As part of a digital ID created in Adobe Acrobat, the signing system is the only system that contains the private key for digital signing. To sign the PDF documents, we used the Apache PDFBox library, Adobe Acrobat Pro 2017, and PDF-Attacker. We created 8 different exploits for all attack variants and evaluated the manipulations under all viewing programs on the victims' systems.

B. Applications

We included PDF viewing applications that could correctly process signed PDF documents. In total, we found 29 PDF

[5] https://fontforge.org/docs/scripting/python.html

Application	Version		Shadow Attack Category: Hide	Replace	Hide-and-Replace	Summary	Fixed (Dec. 7th, 2020)
Adobe Acrobat Reader DC	2019.021.20061	Windows	●	●	●	●	✔
Adobe Acrobat Pro 2017	2017.011.30156		●	●	●	●	✔
Expert PDF 14	14.0.25.3456 64-bit		◐	◐	◐	◐	⚠
Foxit Reader	9.7.0.29455		○	●	●	●	✔
Foxit PhantomPDF	9.7.0.29478		○	●	●	●	✔
LibreOffice Draw	6.4.2.2		○	◐	◐	◐	✔
Master PDF Editor	5.4.38, 64 bit		○	●	●	●	⚠
Nitro Pro	12.16.3.574		◐	◐	◐	◐	⚠
Nitro Reader	5.5.9.2		◐	◐	◐	◐	⚠
PDF Architect 7	7.0.26.3193 64-bit		◐	◐	◐	◐	✔
PDF Editor 6 Pro	6.5.0.3929		●	●	●	●	⚠
PDFelement	7.4.0.4670		●	●	●	●	✔
PDF-XChange Editor	8.0 (Build 331.0)		◐	◐	◐	◐	⚠
Perfect PDF Reader	V14.0.9 (29.0)		◐	◐	◐	◐	⚠
Perfect PDF 8 Reader	8.0.3.5		●	●	●	●	⚠
Perfect PDF 10 Premium	10.0.0.1		●	●	●	●	⚠
Power PDF Standard	3.0 (Patch-19154.100)		●	●	●	●	✔
Soda PDF Desktop	11.1.09.4184 64-bit		○	◐	◐	◐	✔
Adobe Acrobat Reader DC	2019.021.20061	macOS	●	●	●	●	✔
Adobe Acrobat Pro 2017	2017.011.30156		●	●	●	●	✔
Foxit Reader	3.4.0.1012		●	●	●	●	✔
Foxit PhantomPDF	3.4.0.1012		●	●	●	●	✔
LibreOffice Draw	6.4.2.2		○	◐	◐	◐	✔
Master PDF Editor	5.4.38, 64 bit		○	○	○	○	–
PDF Editor 6 Pro	6.8.1.3450		○	○	○	○	–
PDFelement	7.5.7.2895		○	○	○	○	–
Master PDF Editor	5.4.38, 64 bit	Linux	○	●	●	●	⚠
LibreOffice Draw	6.4.2.2		○	◐	◐	◐	✔
Okular	1.9.3		●	●	●	●	⚠
		∑ 29	12● 6◐	16● 10◐	16● 10◐	16● 10◐	15✔ 11⚠

● Application vulnerable. ◐ Vulnerability limited. ○ Not vulnerable.
✔ All reported vulnerabilities are fixed. ⚠ Unfixed application.

Table II. **EVALUATION RESULTS.** OF THE TESTED APPLICATIONS, 16 OUT OF 29 APPLICATIONS ARE VULNERABLE TO AT LEAST ONE ATTACK (●). IN 10 CASES, THE APPLICATIONS SHOW THE SAME WARNING FOR AN *allowed change* (E.G. SIGNING THE DOCUMENT AGAIN) AND A *prohibited change* (E.G. CHANGING CONTENT). WE CALL THIS BEHAVIOR *limited vulnerability* (◐).

applications for Windows, macOS, and Linux. Even if the version numbers do not directly indicate this, the applications PDF Editor 6 Pro, and PDFelement were released in the latest available version in January 2020 for macOS and Windows.

a) Excluded Applications: We only considered applications supporting signature validation. By this means, we excluded popular Linux PDF applications, such as Evince and Okular[6]. For the same reason, we excluded Sumatra (Windows) as well as Preview and Skim (MacOS). We excluded outdated applications that are no longer maintained by the manufacturer, for example, Adobe Reader 9 for Linux. We further excluded online signing services, such as DocuSign and AdobeSign, because they do not provide a visibility layer. These services output a report that denotes whether the PDF signature is valid or invalid. However, it does not provide any information if the *shadow* content is shown or not. Since libraries do not provide the functionality to view PDF documents, we cannot evaluate the attacks' success. Thus, we considered libraries out of scope.

C. Results

Overall, 16 out of 29 PDF viewing applications were vulnerable to at least one presented attack (●) as shown in Table II. For 12 PDF viewers, surprisingly, all three attack classes were successful. Some applications have limited vulnerabilities (◐). A limited vulnerability means that the application always throws a warning, even if a legitimate modification, such as signing the document a second time (e.g., used for contracts). As a result, users do not differentiate between legitimate changes and malicious ones, such as revealing the *shadow* content.

a) Differences in Operating Systems: While we could not find differences for the Adobe products between Windows and macOS versions, we identified significant differences in signature validation of Master PDF Editor, PDF Editor 6 Pro, and PDFelement in these operating systems. No tested attack on the three viewing applications was successful. The reason for these differences lies in the different validation messages shown after opening the signed PDF. On macOS, the three applications throw a warning stating that the signature is *invalid* every time an Incremental Update is detected. In comparison, on Windows, the viewers show that the signature is valid and, in some cases, warn that changes have been made.

The different versions of the applications justify another reason for the divergent results in the operating system's dependence. For instance, on Windows, the Foxit Reader has version 9.7, but on macOS, it has version 3.4. This observation leads to the assumption that both applications can vary in the way PDFs are processed. This assumption is confirmed in Table II. Both applications vary widely regarding the signature validation, which leads to different results concerning the

[6]Due to the chosen Linux distribution, the installed Okular version did not support signature validation for the time we provided our evaluation. In the meantime, this support was added. We found multiple vulnerabilities, which we immediately reported as part of the responsible disclosure process. This process is not finished yet.

shadow attacks. Interestingly, under macOS, Foxit's applications have the unique feature that the signature status only changes from unknown to valid if macOS's keychain contains the private key.

b) Hide: The Hide *shadow* attack class was successful for 12 PDF viewing applications. For the exploit, the overlay image file was re-declared as `/Subtype/XML/Type/Metadata` with `/Subtype /Image /Type /XObject` when using Incremental Updates. The Adobe Acrobat applications then faded out the image file, but at the same time, the applications confirmed a valid signature for the PDF document in UI-Layers 1 and 2. A manually initiated signature check returns the error code 109, but the signature status remains unaffected.

c) Replace: In total, 16 PDF viewing applications were vulnerable to the Replace *shadow* attack class. They split into two different attack variants: *replace via overlay* and *replace via overwrite*. While Adobe viewers correctly classify the font exchange as an unauthorized Incremental Update, the signature remains valid when exchanging field text within the form. We observed the exact opposite case with PDF Editor 6 Pro and PDFelement. While we could successfully manipulate the fonts, the signature status changed to invalid when the field content was exchanged. The Foxit Reader showed another behavior worth mentioning. After the update from version 9.5.0.20723 to 9.7.0.29455, Incremental Updates allow the exchange of fonts without invalidating the signature. Since LibreOffice Draw ignores the fonts contained in the PDF document, the application is immune to this type of attack. However, it is possible to exchange the field text without invalidating the signature.

d) Hide-and-Replace: In 16 PDF viewing applications, we could identify Hide-and-Replace vulnerabilities. The two Adobe viewing applications successfully displayed the content hidden in the document. In contrast, they displayed the signature as valid in UI-Layers 1 and 2. A manually started signature check provides a message about an invalid node within the page structure data, but the signature status remains unaffected.

e) Responsible Disclosure: According to the evaluation depicted in Table II, we started the responsible disclosure process for 26 vulnerable applications – 16 fully vulnerable and 10 with limited vulnerabilities. We cooperated with CERT-Bund (BSI) for the responsible disclosure and created a dedicated vulnerability report. The CERT-Bund thankfully contacted all affected vendors and also related organisations working with digitally signed PDFs. Some vendors responded quickly and informed us that fixes were already implemented (Adobe, Foxit, LibreOffice, Power PDF, Soda PDF). In some cases, the vendors contacted us for technical queries. In other cases, we got a message confirmation but no feedback regarding patches (Master PDF, Nitro, PDF Architect, PDF-XChange, Power PDF). In four cases, we could not get any feedback, despite multiple contact attempts within more than seven months (Expert PDF, PDF Editor Pro, Perfect PDF , PDFelement).

Seven months after our report, we re-evaluated all PDF applications listed in Table II, using the latest available software version[7]. The current status of the fixes can be summarized as follows.

[7]Status as of Dec. 7th, 2020

- ✔ *Fixed applications*: We verified the vendors' fixes for 15 PDF applications. For PDF Architect, PDFelement and Soda PDF, it is necessary to upgrade to the next program version: PDF Architect 8, PDFelement 8, Soda PDF 12.
- ⚠ *Unfixed applications*: The security gaps in six of the vulnerable PDF applications have not yet been closed (Master PDF in Windows and Linux, Nitro Pro, PDF-XChange Editor, Perfect PDF Reader). Five applications (Expert PDF 14, Nitro Reader, PDF Editor 6 Pro, Perfect PDF 8 Reader, Perfect PDF 10 Premium) have not been updated in the meantime. Hence, the reported vulnerabilities are still present in these eleven cases.

To find out how the vendors fixed the vulnerabilities, we contacted all 15 vendors and asked for details of their fixes. We used multiple-choice answers (see section A) that we derived from our *PDF-Detector* implementation (cf. subsection VIII-C) to be able to compare the fixes. We received responses from five vendors. Adobe and PDF Architect responded: "If there exists an Incremental Update after signing, we compare the parsed document with and without this Incremental Update". PDF Architect also reported: "If an Incremental Update contains a font that overwrites an existing font, we mark a signed document as invalid". Foxit described an attack detection procedure comparable to Adobe's solution: "For Replace and Hide-and-Replace, our solution is to analyze the incremental part after the signature, and check whether there is any part that can be modified within the scope of permission. If it is not allowed, the direct return is invalid, if it is allowed, it will show that there is modification". LibreOffice always informs the user about a partially signed document if there were further Incremental Updates after the signature. Before our security report, LibreOffice marked signed documents as invalid only if there were changes in the signed area ("byte range"). Using the *shadow* attacks, we could show that the whole document can be changed with Incremental Updates and without manipulating the signed area. In a bilateral exchange, we were able to convince the LibreOffice development team to mark signed PDF documents as invalid even if the signed area remains cryptographically untouched and an Incremental Update modifies the content. Master PDF and Okular wrote that the vulnerabilities should be closed soon.

Besides the applications, we also found out that two online signing and validation services recognized the impact of the attacks and implemented countermeasures [9, 10].

VIII. PDF-Detector

On an abstract level *shadow* attacks are executed in two steps. First, the attackers prepare a *shadow* document, which hides malicious content. This document is then signed. Second, the attackers manipulate the signed document to show the hidden content while keeping the signature status *valid*.

We developed *PDF-Detector*, a tool to prevent and detect *shadow* attacks. *PDF-Detector* proposes two different approaches to mitigate and detect *shadow* documents concerning the countermeasures. In subsection VIII-A, we discuss a countermeasure detecting *shadow* documents before they are signed. Thus, we prevent *shadow* attacks in the first step of their execution. This countermeasure is suitable for every PDF viewer or application capable of signing PDF documents.

In subsection VIII-B, we propose an algorithm encountering *shadow* documents that have already been signed. This countermeasure addresses the forensic analysis of signed PDF documents assuming that previous software did not prevent the signing of *shadow* documents. By combining both countermeasures, we can prevent *shadow* attacks in both phases of their execution.

We implemented both, *prevention* and *detection*, and tested our code against all exploit files. See subsection VIII-C for our results.

A. Prevention

In this section, we introduce an algorithm capable of detecting hidden content. Thus, users can be warned before signing.
To find hidden (inactive) *shadow* content, it is necessary to reliably extract text, images, and forms from the PDF document for analysis. The attacks of the *Hide* category use overlays of different objects, for example, hiding a text under an image. To detect these overlays, *PDF-Detector* must extract the objects' rectangular coordinates within the PDF document. It can then use these coordinates (left, bottom, right, top) to determine an object's exact position within a page. An image that overlays a text box can be identified by calculating a collision of both objects using the coordinates [11]. When creating a PDF document, a slight overlapping of images and text boxes is not unusual. For this reason, the collision calculation should calculate the value of the overlay in percent. The lower the value, the less content is covered by the object. For the *Replace* category, a *prevention* phase cannot be sensibly implemented. The first step "injecting shadow content" is indistinguishable from a benign injection. For example, inserting multiple fonts when creating a PDF document is not an unusual malicious behavior. For the *Hide-and-Replace* category, this does not apply at all. Here, the first step can be detected, because the *shadow* document path is contained in the document. A promising way to recognize this is to exchange the references to the `Kids` objects for all `Pages` objects. Subsequently, all correctly referenced objects of the newly created PDF document are compared with the source document. In an unmanipulated PDF document, only the positions of the contained pages are swapped. Suppose the PDF document contains *shadow* content of the category *Hide-and-Replace*. In that case, the attackers make it visible in the second step by referencing it. Thus, it can be identified.

B. Detection

While active (hidden) *shadow* content should be discovered during the *prevention* phase, active (visible) *shadow* content is in the focus of the *detection* phase. In practice, this means attackers created a *shadow* document, which the signers signed. Afterwards, the attackers made the *shadow* content visible. For detecting *Hide*, *Hide-and-Replace* and the overlay variant of *Replace* attack classes, *PDF-Detector* can compare the current document with the document before it was signed. This comparison can be technically implemented by removing all data, that is, all Incremental Updates, after the first signature. The discrepancy between the two documents becomes apparent in the second step of the *Hide* attacks, by the absence of an object in the signed document, for example, a missing image object. In contrast, in the second step in *Hide-and-Replace*, attackers deliver a different overall content compared to the unsigned document. The detection of a *Replace* attack in the Font variant is less complex. For this purpose, it is sufficient to scan updates after signing for added `FontFile` objects and to comparing their object number with already contained `FontFiles`.

C. Implementation Details

PDF-Detector uses combination of the Python libraries *PDFMiner* and *pdfrw*. In practice, PDF documents are often compressed, which further complicates the analysis of the content, since *pdfrw* cannot handle the Deflate compression algorithm used in PDF [12, 13]. For this reason the *PDF-Detector* decompresses the whole PDF document using *pypdftk* and *pdftk* if necessary.

The command-line tool accepts a PDF document as input. First, it checks if the document already contains a signature to select the correct mode. If no signature is found, the *prevention* mode starts and analyzes the document as described in subsection VIII-A. If the document has already been signed, the *detection* mode starts and checks the document for any visible *shadow* content as described in subsection VIII-B. If no visible *shadow* content is found, the analysis is additionally started in *prevention* mode to search for hidden (inactive) *shadow* content.

For assuring that all exploits are correctly detected, we used the PDF files generated by our *PDF-Attacker* implementation. This processing ensures that both phases, *prevention* and *detection*, can be tested. We were able to verify and fine-tune the correct analysis based on these 26 PDF documents, including 4 unsigned documents, 7 inactive *shadow* documents, 7 signed but inactive *shadow* documents, and 8 active *shadow* document. We plan to train the tool with additional documents to strengthen the detection rate and minimize the false positive rate.

IX. Shadow Attack: Beyond Signature Bypasses

The concept of the *shadow* attack is not limited to the attacks on PDF signatures. Analyzing the PDF specification and Adobe products, we observed exciting features and configuration possibilities.

a) High Privileged Actions in PDF: The PDF specification basically defines two kinds of Code Execution (CE): PDF actions and JavaScript. Actions are limited in their functionality. A popular example of actions is URL invocation. In contrast, JavaScript provides a huge function set, including control structures (e.g., if, while). For security reasons, both types of CE are restricted in PDF. For example, URL invocations require user confirmation, and access to other documents and files is blocked.

During our research, we determined that both CE variants can run in privileged mode. This mode allows the execution of security-critical actions without any restriction or user consent. For example, privileged JavaScript can change the UI and functionality of the viewer application's menu items. It can read the content of other opened PDF tabs or even files stored on the machine. It can also invoke URLs without any confirmation.

Typically, a PDF is not allowed to execute privileged JavaScript or actions without user confirmation or configuration changes in the PDF viewer.

One exception exists by using digitally signed PDFs, more concrete *certified PDFs*. Adobe Products users can configure PDFs signed with a specific certificate to be given permission to execute high privileged operations. This setting is disabled by default for most preconfigured CAs and all manually trusted certificates. To our surprise, we found an exception in Adobe Products: *if the private key for a certificate is known, PDF documents signed with this particular certificate are automatically allowed to execute high privileged code*.

b) Attack Idea: Inspired by the shadow attacks concept, we raised the question whether attackers could hide *shadow* actions or *shadow* JavaScript in the PDF so that it is executed after its signing.

For this purpose, we create a new attack based on the following simplified attacker model. We assume that the signers and the victims are the same entities. Additionally, the attackers do not need to manipulate the PDF document after its signing. When taking note of Figure 3, the only step executed by the attackers is to create $\texttt{PDF}_1$ and to embed the high privileged code inside. Once the $\texttt{PDF}_1$ is signed, the *high privileged code is executed* automatically on the signers' machine.

c) Attack Description: The attack works as follows:

1) The attackers generate a PDF containing malicious, high privileged code, for example, read access to other PDF tabs using privileged JavaScript. The code is stored on a specific execution event that will be triggered *after* signing the document, for example, during the *willClose* event, or any other event which is triggered after the signing (e.g., *willSave*, *didSave*).
2) The PDF is sent to the signers. They sign the document.
3) After the signing, the victims save and close the PDF.
4) The closing of the document triggers the *willClose* event and the malicious code inside the PDF is executed.
5) Typically, the privileged JavaScript is not executed because the special permission for this is not granted. However, the application sees that the private key for the certificate used to sign the PDF is known. This mistakenly convinces the application to execute privileged JavaScript – the application assumes that the signer intended to execute the script, because they signed it.

This attack is limited to Adobe products since they define a special policy regarding the CE and handle *signed* PDFs differently than unsigned.

d) Responsible Disclosure: When we initially reported this issue to Adobe, their security team rejected to classify our findings as a vulnerability. They assumed that all collaborators who are working on the document trust each other. After a short discussion, we convinced them that this is not always the case. As a direct result of our finding, Adobe has implemented security controls in their May 2020 release, which "prevent[s] signing until warnings are reviewed".

X. Related Work

a) PDF Signatures: Attacks on electronic signatures which abuse the missing cryptographic protection were described in 2008 and 2012 by Grigg [15, 14]. In 2010, Raynal et al. [16] considered potential security issues regarding the signature verification by criticizing the design of the certificate trust establishment. The first attack that bypassed the cryptography in PDFs was introduced in 2017 by Stevens et al. [17]. The researchers attacked the collision resistance of SHA-1 and created two different PDF files containing the same digest value but different content. In 2019, Mladenov et al. [6] published a comprehensive study regarding the security of PDF signatures and they discovered three novel attacks and revealed all current applications to be vulnerable. More details regarding their relation to our work is discussed in subsection V-E.

b) Content Masking Attacks: A polymorphic attack containing two different files, a PDF and TIFF, have been introduced in 2009 by Buccafurri et al. [18] and re-implemented later by Popescu [19]. Depending on the viewer used, different content is shown. This risk exists if the victims sign the document and are unaware of the hidden content. A similar approach was introduced in 2014 by Albertini [20]. He combined a PDF and a JPEG into a single polyglot file. In 2015, Lax et al. [21] systematized potential security topics related to digitally signed documents including the signature generation process, signed documents containing dynamic content like macros or JavaScript, and polymorphic documents similar to Popescu [19]. All of the attacks rely either on different viewers' usage to open the malicious PDF or loading dynamic content from attackers' controlled source. None of these requirements are needed for the attacks presented in this paper. In 2017, Markwood et al. [7] introduced a novel attack related to content masking by using font encoding. As a result, the researchers could trick automated content analyzing software to process different data than the data displayed.

c) PDF Malware: Since 2010, Raynal et al. [16] abuse legitimate features in PDFs to carry out attacks such as Denial-of-Service (DoS), Server-Side-Request-Forgery (SSRF), and information leakage. In 2013 and 2014, multiple vulnerabilities in Adobe Reader were reported to be abusing legitimate PDF features, JavaScript, and XML [22, 23]. In 2015, Inführ [24] systematized the current risks related to features in PDFs, which lead to security issues. Valentin [25] published a study revealing weaknesses related to malicious URI invocation. In 2018, these attack vectors were extended by Franken et al. [26] who revealed weaknesses in two PDF readers by forcing them to call arbitrary URIs. In the same year, multiple vulnerabilities in Adobe Reader and different Microsoft products were discovered, leading to URI invocation and NTLM credentials leakage [27, 28].

Motivated by the discovered attacks since 2010, different security tools were implemented to detect maliciously crafted documents [29, 30, 31, 32]. Such tools relied on the detection of known attack patterns and structural analysis of PDFs. The list of tools was extended by new malware classifiers based on machine learning [33, 34, 35, 36, 37, 38, 39, 40]. Motivated by previously discovered problems regarding the detection of malicious PDF files [41, 42], Chen et al. [43] published in 2020 a methodology for robust classification of PDF Malware. The

authors achieved 92.27% accuracy and a 0.56% false positive rate.

XI. Future Work

In this section, we discuss several problems that should be addressed by future research.

a) Secure and Insecure Document Updates: One of the main features we abused is Incremental Update. Using Incremental Update, previously hidden content could be displayed without raising security warnings. The main problem is the flexibility of the current specification allowing multiple Incremental Updates without invalidating the signature. However, the developers of the applications are left to themselves to address the problem. They differ between dangerous and harmless Incremental Updates on their own. As a result, inconsistencies regarding the signature validation status and the displayed content exist, depending on which PDF viewer is used. Future research should systematically discover all allowed and forbidden changes to address these inconsistencies, analyze their impact regarding security, and propose a countermeasure if needed. Our attacks only considered one-time signed documents. The attacker model could be adjusted to simulate use cases in which multiple signer entities are involved. In such cases, the attackers can insert content before the next signature is applied. It is currently unclear what kind of changes they can apply and which kind of content could be *shadowed*.

Establishing a systematic evaluation approach is not a trivial task. For instance, the `/Catalog` object can contain up to 28 attributes. One of these attributes is the reference to the `/Pages` object, which can also have up to 30 attributes, and it can refer to further objects. Due to the large amount of test cases, a tool for (semi-)automatically generating PDF test cases should be implemented. One possible approach to achieve this is to implement a tool producing a series of test cases that contains many varieties of manipulations, following a fuzzing alike approach. The main challenge here is to create a meaningful Incremental Update without invalidating the digital signature by applying trivial manipulations.

b) Updates and Parsing Errors: While our research concentrated on PDF standard compliant documents, previous research focused on Incremental Updates that are not standard compliant [6]. A combination of both techniques could reveal new insights. For example, during the responsible disclosure period, we could bypass the implemented countermeasures several times by just removing (or commenting out) code fragments within the PDF document. We encourage the development of fuzzing techniques capable of covering a large number of document variants.

A recently published research article by Kuchta et al. [44] revealed new insights into this problem. The authors analyzed 230 000 real-word PDF documents provided by Garfinkel et al. [45] and discovered that 13.5% of the PDFs were improperly rendered. The authors concentrated only on inconsistencies during rendering without evaluating the security implications. The second study could be provided by extending the scope to security and considering more real-world examples than mentioned by Garfinkel et al. [45].

XII. Conclusion

PDF signatures are designed to protect the integrity and authenticity of PDFs. In contrast to the classical digital signature use cases that apply a signature only once on a target, PDF signatures address more complex use cases. A signed document is allowed to be updated without invalidating its signature, but only in particular cases. Additionally, a PDF can be signed several times in succession. In this paper, we showed how this flexibility could be abused to replace the entire content of the PDF without invalidating the signature. As a result, we found 16 of 29 applications to be vulnerable.

The reasons for this state can be found in the current PDF specification: (1) It describes imprecisely how the signature validation may be implemented. (2) It does not document edge cases and does not propose a solution or a guideline. As a result, developers must solve these problems on their own. (3) The PDF specification should reconsider the feature-richness which weakens the security. Instead, it should apply stricter and more limited handling regarding cryptographic protection. As a reaction to our research, we became a member of the ISO/TC 171/SC 2 technical committee to contribute to future PDF standards.

Acknowledgment

The authors would like to thank Sebastian Lauer, Paul Rösler, Marcus Niemietz, and Jörg Schwenk for their valuable discussions, feedback, and support. Simon Rohlmann was supported by the German Federal Ministry of Economics and Technology (BMWi) project “Industrie 4.0 Recht-Testbed” (13I40V002C). Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany’s Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] United States Government Printing Office, “Electronic signatures in global and national commerce act,” 2000. [Online]. Available: https://www.govinfo.gov/content/pkg/PLAW-106publ229/pdf/PLAW-106publ229.pdf

[2] E. Union, “Regulation (eu) no 910/2014 of the european parliament and of the council on electronic identification and trust services for electronic transactions in the internal market and repealing directive 1999/93/ec,” 2014. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32014R0910

[3] Wikipedia. (2019) Electronic signatures and law. [Online]. Available: https://en.wikipedia.org/wiki/Electronic_signatures_and_law

[4] Adobe. (2018, Nov.) Adobe fast facts. [Online]. Available: https://www.adobe.com/about-adobe/fast-facts.html

[5] DocuSign. (2019) Docusign 2019 annual report. [Online]. Available: https://s22.q4cdn.com/408980645/files/doc_financials/2019/Annual/DocuSign-FY2019-Annual-Report.pdf

[6] V. Mladenov, C. Mainka, K. Meyer zu Selhausen, M. Grothe, and J. Schwenk, “1 trillion dollar refund – how to spoof pdf signatures,” in *ACM Conference on Computer and Communications Security*, Nov. 2019.

[7] I. Markwood, D. Shen, Y. Liu, and Z. Lu, "PDF Mirage: Content Masking Attack Against Information-Based Online Services," in *26th USENIX Security Symposium (USENIX Security 17), (Vancouver, BC)*, 2017, pp. 833–847.

[8] pdf-insecurity.org. (2020, apr) Exploits. [Online]. Available: https://www.pdf-insecurity.org/signature/evaluation_2018.html#desktop-viewer-applications

[9] Intarsys. (2020) Releasenotes: Signlive 7.1.6. [Online]. Available: https://www.intarsys.de/sites/default/files/Dokumente/ReleaseNotes_SignLive_7.1.6.txt

[10] I. PDF. (2020, September) Investigating pdf shadow attacks: What are shadow attacks? [Online]. Available: https://itextpdf.com/en/blog/technical-notes/investigating-pdf-shadow-attacks-what-are-shadow-attacks-part-1

[11] M. contributors. (2019, Nov.) 2d collision detection. [Online]. Available: https://developer.mozilla.org/en-US/docs/Games/Techniques/2D_collision_detection

[12] A. S. Incorporated, *PDF Reference, version 1.7*, 6th ed., November 2006.

[13] P. Maupin. (2017, Sep.) pdfrw 0.4: Project description. [Online]. Available: https://pypi.org/project/pdfrw/#all-examples

[14] I. Grigg. (2008) Technologists on signatures: looking in the wrong place. [Online]. Available: http://financialcryptography.com/mt/archives/001056.html

[15] ——. (2012) Signatures on fax & email - if you did not intend to be bound, why did you bother to write it? [Online]. Available: http://financialcryptography.com/mt/archives/001364.html

[16] F. Raynal, G. Delugré, and D. Aumaitre, "Malicious Origami in PDF," *Journal in Computer Virology*, vol. 6, no. 4, pp. 289–315, 2010. [Online]. Available: http://esec-lab.sogeti.com/static/publications/08-pacsec-maliciouspdf.pdf

[17] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full sha-1," in *Annual International Cryptology Conference*. Springer, 2017, pp. 570–596.

[18] F. Buccafurri, G. Caminiti, and G. Lax, "Fortifying the dalì attack on digital signature," in *Proceedings of the 2nd International Conference on Security of Information and Networks*, ser. SIN '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 278–287. [Online]. Available: https://doi.org/10.1145/1626195.1626262

[19] D. Popescu, "Hiding malicious content in PDF documents," *CoRR*, vol. abs/1201.0397, 2012. [Online]. Available: http://arxiv.org/abs/1201.0397

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

[21] G. Lax, F. Buccafurri, and G. Caminiti, "Digital document signing: Vulnerabilities and solutions," *Information Security Journal: A Global Perspective*, vol. 24, no. 1-3, pp. 1–14, 2015.

[22] B. Rios, F. Lanusse, and M. Gentile. (2013) Adobe reader same-origin policy bypass. [Online]. Available: http://www.sneaked.net/adobe-reader-same-origin-policy-bypass

[23] A. Inführ. (2014, Dec.) Multiple pdf vulnerabilities – text and pictures on steroids. [Online]. Available: https://insert-script.blogspot.de/2014/12/multiple-pdf-vulnerabilites-text-and.html

[24] ——. (2015, Sep.) Pdf – mess with the web. [Online]. Available: https://2015.appsec.eu/wp-content/uploads/2015/09/owasp-appseceu2015-infuhr.pdf

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

[26] G. Franken, T. V. Goethem, and W. Joosen, "Who left open the cookie jar? a comprehensive evaluation of third-party cookie policies," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 151–168. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/franken

[27] A. Inführ. (2018, May) Adobe reader pdf - client side request injection. [Online]. Available: https://insert-script.blogspot.de/2018/05/adobe-reader-pdf-client-side-request.html

[28] C. P. Research. (2018, April) Ntlm credentials theft via pdf files. [Online]. Available: https://research.checkpoint.com/ntlm-credentials-theft-via-pdf-files/

[29] P. Laskov and N. Šrndić, "Static detection of malicious javascript-bearing pdf documents," in *Proceedings of the 27th annual computer security applications conference*. ACM, 2011, pp. 373–382.

[30] D. Maiorca, G. Giacinto, and I. Corona, "A pattern recognition system for malicious pdf files detection," in *International Workshop on Machine Learning and Data Mining in Pattern Recognition*. Springer, 2012, pp. 510–524.

[31] C. Smutz and A. Stavrou, "Malicious pdf detection using metadata and structural features," in *Proceedings of the 28th annual computer security applications conference*. ACM, 2012, pp. 239–248.

[32] I. Corona, D. Maiorca, D. Ariu, and G. Giacinto, "Lux0r: Detection of malicious pdf-embedded javascript code through discriminant analysis of api references," in *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*. ACM, 2014, pp. 47–57.

[33] D. Maiorca, D. Ariu, I. Corona, and G. Giacinto, "A structural and content-based approach for a precise and robust detection of malicious pdf files," in *2015 International Conference on Information Systems Security and Privacy (ICISSP)*. IEEE, 2015, pp. 27–36.

[34] N. Šrndić and P. Laskov, "Hidost: a static machine-learning-based detector of malicious files," *EURASIP Journal on Information Security*, vol. 2016, no. 1, p. 22, 2016.

[35] L. Tong, B. Li, C. Hajaj, C. Xiao, and Y. Vorobeychik, "A framework for validating models of evasion attacks on machine learning, with application to pdf malware detection," *arXiv preprint arXiv:1708.08327*, 2017. [Online]. Available: https://arxiv.org/pdf/1708.08327.pdf

[36] D. Maiorca and B. Biggio, "Digital investigation of pdf files: Unveiling traces of embedded malware," *IEEE Security and Privacy: Special Issue on Digital Forensics*, In Press. [Online]. Available: https://pralab.diee.unica.it/sites/default/files/maiorca17-sp.pdf

[37] S. Dey, A. Kumar, M. Sawarkar, P. K. Singh, and S. Nandi, "EvadePDF: Towards evading machine learning based PDF malware classifiers," in *Communications in Computer and Information Science*, vol. 939, 2019, pp. 140–150.
[38] Y. Li, Y. Wang, Y. Wang, L. Ke, and Y. an Tan, "A feature-vector generative adversarial network for evading PDF malware classifiers," *Information Sciences*, vol. 523, pp. 38–48, 2020.
[39] A. Corum, D. Jenkins, and J. Zheng, "Robust PDF Malware Detection with Image Visualization and Processing Techniques," in *Proceedings - 2019 2nd International Conference on Data Intelligence and Security, ICDIS 2019*, 2019, pp. 108–114.
[40] "On training robust PDF malware classifiers," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/chen-yizheng
[41] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers," in *Proceedings of the 2016 network and distributed systems symposium*, vol. 10, 2016.
[42] N. Srndic and P. Laskov, "Practical evasion of a learning-based classifier: A case study," in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 197–211.
[43] Y. Chen, S. Wang, D. She, and S. Jana, "On training robust PDF malware classifiers," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/chen-yizheng
[44] T. Kuchta, T. Lutellier, E. Wong, L. Tan, and C. Cadar, "On the correctness of electronic documents: studying, finding, and localizing inconsistency bugs in PDF readers and files," *EMPIRICAL SOFTWARE ENGINEERING*, vol. 23, no. 6, pp. 3187–3220, DEC 2018.
[45] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *digital investigation*, vol. 6, pp. S2–S11, 2009.

APPENDIX

EMAIL SENT TO VENDORS

We contacted all vendors via email to gather information on how they fixed the shadow attacks. To get comparable results, we requested the information in a multiple-choice fashion.

> *Dear Sir or Madam,*
> *we are security researchers at* Ruhr University Bochum*. Together with the CERT-Bund (reference: [CERT-Bund#2020030228000759] / CVE-2020-9592 / CVE-2020-9596), we informed you in March of this year about security vulnerabilities (called "Shadow Attacks") in your PDF application. For our research, we are very interested in how the vulnerabilities mentioned above were fixed. In order not to take up too much of your time, we have created some keywords and would like to ask you to tick the appropriate lines. Of course we are also grateful for any further comments in this context.*
> ***Attack variant "Hide-and-Replace":***
> ☐ *We have not fixed the vulnerability yet.*
> ☐ *We always mark a signed document as invalid, if there exist any Incremental Updates after signing.*
> ☐ *If there exist an Incremental Update after signing, we compare the parsed document with and without this Incremental Update.*
> ☐ *We check the document for hidden object paths before signing.*
> ☐ *If there are two or more complete object paths (Pages → Page → Content), we always mark a signed document as invalid.*
> ☐ *Other methods:*
> *Further comments:*
> ***Attack variant "Hide":***
> ☐ *We have not fixed the vulnerability yet.*
> ☐ *We always mark a signed document as invalid, if there exist any Incremental Updates after signing.*
> ☐ *If there exist an Incremental Update after signing, we compare the parsed document with and without this Incremental Update.*
> ☐ *If an Incremental Update overlays a form or content stream object, we mark a signed document as invalid.*
> ☐ *Other methods:*
> *Further comments:*
> ***Attack variant "Replace":***
> ☐ *We have not fixed the vulnerability yet.*
> ☐ *We always mark a signed document as invalid, if there exist any Incremental Updates after signing.*
> ☐ *If there exist an Incremental Update after signing, we compare the parsed document with and without this Incremental Update.*
> ☐ *If an Incremental Update contains a font that overwrites an existing font, we mark a signed document as invalid.*
> ☐ *If an Incremental Update delete or overwrite an existing image or form object, we mark a signed document as invalid*
> ☐ *Other methods:*
> *Further comments:*