Code Vulnerabilities in NSA Application Revealed

BY DENNIS BRINKROLF | APRIL 06, 2021

Security

Emissary is a P2P based data-driven workflow engine that runs in a heterogeneous possibly widely dispersed, multi-tiered P2P network of compute resources. The application's Java source code is distributed by the official GitHub repository of the U.S. National Security Agency (NSA). An interesting pick for our research team to look at its code security.

In our analysis, we discovered several code vulnerabilities in Emissary version 5.9.0. A combination of these vulnerabilities allows remote attackers to execute arbitrary system commands on any Emissary server. All in all, this may lead to the compromise of the whole P2P network.

In this blog post we analyze the technical root cause of three different security issues and demonstrate how attackers could exploit these. We reported all issues responsibly to the affected vendor who released multiple security patches to protect all users against the most severe vulnerabilities.

Impact

During the analysis of Emissary 5.9.0 we found the following code vulnerabilities that enable different ways to attack the application:

- Code Injection (CVE-2021-32096)
- Arbitrary File Upload (CVE-2021-32094)
- Arbitrary File Disclosure (CVE-2021-32093)
- Arbitrary File Delete (CVE-2021-32095)
- Reflected Cross-site-Scripting (CVE-2021-32092)

Access to the web application (and its vulnerable features) is protected by HTTP Digest Authentication. By default, there is only one administrator account that has access to the web application. However, the web application is vulnerable to Cross-Site Request Forgery (CSRF) attacks. This allows an attacker to abuse the browser of an authenticated victim to manipulate the state of the web application. For example, the CSRF vulnerability can be combined with the Code Injection vulnerability to achieve remote code execution. You can find out more about CSRF and how it can be exploited by attackers in our previous blog post.

For demonstration purposes we've created a short video that shows how quick and easy a server is compromised.

Technical Analysis

In the following, we look at the root cause of three vulnerabilities in the source code of Emissary. First we introduce the Code Injection vulnerability that can be exploited via CSRF. In the next step, we analyse two vulnerabilities (Arbitrary File Disclosure, Cross-site Scripting) that can be combined by an attacker to extract the administrator credentials of the HTTP Digest Authentication used by Emissary.

Remote Ruby Code Execution (CVE-2021-32096)

The administration area of Emissary includes a console feature to evaluate Ruby code. Since the entire web application does not use CSRF tokens, an attacker can execute arbitrary Ruby code on the server through the browser of a logged-in administrator. Let's have a look at the source code.

src/main/java/emissary/server/mvc/ConsoleAction.java

```
47   @POST
48   @Path("/Console.action")
49   @Produces(MediaType.TEXT_PLAIN)
50   public Response rubyConsolePost(@Context HttpServletRequest request) {
52       RubyConsole console = getOrCreateConsole(request);
56       try {
57           final String cmd = request.getParameter(CONSOLE_COMMAND);
67           if ("eval".equals(cmd)) {
69               String commandString = request.getParameter(CONSOLE_COMMAND_STRING);
70               if (commandString != null) {
79               try {
80                   result = console.evalAndWait(commandString, 60000);
81               }
92           }
94       }
118  }
```

In line 57 the user controlled post parameter `CONSOLE_COMMAND` is received and in line 67 it is checked if this parameter is equal to the string *eval*. If it is, the next attacker controlled post variable `CONSOLE_COMMAND_STRING` is received in line 69 and passed to the function `evalAndWait()` from the class `RubyConsole` in line 80. When following the function `evalAndWait()` we will get to the `eval()` function as shown below.

src/main/java/emissary/scripting/RubyConsole.java

```
331  public Object eval(String expression) throws Exception{
332      Object result = null;
333      try{
334          result = rubyEngine.eval(expression,rubyContext);
339      }
349  }
```

The function `eval()` receives a Ruby `expression` as the first parameter which can be controlled by an attacker in order to execute the vulnerable function `eval()` of the Ruby engine in line 334. This allows an attacker to inject arbitrary Ruby code for execution on the server (S5334).

The Ruby Console is obviously intended as a feature and is not an actual vulnerability. The problem is, however, that the web application does not use CSRF tokens and an adversary can thus abuse any feature of the software within an attack.

Leaking the Admin Password

Further, we detected an Arbitrary File Disclosure and Cross-site Scripting vulnerability. Both can be combined to read arbitrary files from the Emissary server. For example, an attacker could read the stored admin credentials for the HTTP Digest Authentication and then login to Emissary as an administrator to take over the installation.

```
# Defines the members of the Emissary HashUserRealm for jetty authentication
# Passwords can be clear text, obfuscated, or checksummed. Use
org.mortbay.jetty.security.Password
# to generate obfuscated or checksummed passwords
#
# <Username>: <password> [, <rolename> ... ]
#
# This default value is the obfuscation of 'emissary123'
#
emissary: OBF:1j8x1kmy1jnb1zsp1yf41ri71yf21zt11jk71kjo1j65, emissary

#
# This is the default console login, obf of ' console123'
#
console: OBF:1iun1ll81kn01vv11wg21wfw1vu91kjm1kxu1irz, support
```

Arbitrary File Disclosure (CVE-2021-32093)

Emissary's feature to show certain configuration files contains a File Disclosure vulnerability (S2083) that can be used to read any file from the server. In line 35, the user-controlled HTTP GET variable `CONFIG_PARAM` is received from the query string. The variable `configName` is not sanitized and can contain any file path. The content of the opened file in line 44 is then printed in line 45.

src/main/java/emissary/server/mvc/internal/ConfigFileAction.java

```
32   @GET
33   @Path("/ConfigFile.action")
34   @Produces(MediaType.TEXT_PLAIN)
35   public Response configFile(@QueryParam(CONFIG_PARAM) String configName) {
36       try {
44           String content = IOUtils.toString(
                 ConfigUtil.getConfigStream(configName), StandardCharsets.UTF_8);
45           return Response.ok().entity(content).build();
46       }
51   }
```

By using a Path Traversal attack and injecting character sequences like `../` a malicious user can traverse through the file system and read any file on the system, including the HTTP Digest Authentication file that contains the secret credentials. However, this feature is only available to authenticated users and in a CSRF attack it is not possible to read the request's response. A remote attacker needs another vulnerability.

Reflected Cross-site-Scripting (CVE-2021-32092)

We found a Cross-Site Scripting vulnerability (S5131) in the error response message of the `DocumentAction` class. When a requested document is not found, user input is reflected without any output encoding.

src/main/java/emissary/server/mvc/DocumentAction.java

```
62   @GET
63   @Path("/Document.action/{uuid}")
64   @Produces(MediaType.APPLICATION_XML)
65   public Response documentShow(
       @Context HttpServletRequest request,
       @PathParam("uuid") String uuid
     ) {
69       final List<IBaseDataObject> payload = wsp.take(uuid);
...
82       return Response.status(400).entity("<error>uuid " + uuid + " not found</error>").build();
92   }
```

The user controlled GET variable `uuid` is passed in line 65 via a path. Then, the variable is used in line 69 in the function `wsp.take()` and if no element is found for the passed `uuid`, an error message is printed in line 82. So the user-controlled input is concatenated and printed with an error message.

An attacker can therefore craft a malicious link that passes a payload via the `uuid` parameter which executes JavaScript in the victim's browser. The HTTP response has an XML content type (see line 64) but this does not prevent an attacker from executing an arbitrary JavaScript payload. Once this payload executes in an authenticated victim's browser it can exploit the File Disclosure vulnerability to read the administrator credentials and send these to an attacker-controlled server. With this, a remote attacker is able to gain access to the credentials and to authenticate. Next to the previously described Code Injection vulnerability, we also reported a File Delete and File Upload vulnerability that could be exploited by the attacker once authenticated.

Timeline

Date	Event
24.09.2020	We ask for security contact on GitHub issues
01.10.2020	We ask for security contact via generic email

15.12.2020	Emissary releases 5.11.0 to fix RCE and sets up new email
17.12.2020	We inform about email problem
07.01.2021	Emissary resolves email problem, we inform about remaining vulnerabilities
01.02.2021	We ask for status update: work in progress
26.02.2021	We inform about upcoming disclosure
02.03.2021	Emissary releases 6.1 and informs that all issues should be fixed
05.03.2021	We inform about unpatched CSRF and Path Traversal

Summary

In this blog post, we analyzed three vulnerabilities found in Emissary. The combination of the vulnerabilities can lead to a complete takeover of an Emissary installation. We evaluated the causes in the Java code and explained how an attacker can exploit them. We also showed that a simple authentication is not enough to secure a web application and how intended features of developers offer a high potential for attackers. If you are hosting an Emissary instance and have not yet updated your installation, we highly recommend that you do so now. Last but not least, we would like to thank the Emissary team who addressed most of the issues in the latest release 6.1.

DENNIS BRINKROLF
Security Researcher
in