August 30, 2021

GHSL-2021-087: Pre-auth unsafe deserialization in ZStack - CVE-2021-32836


Alvaro Munoz

Coordinated Disclosure Timeline

- 2021-06-04: Details reported to maintainers using private GHSA
- 2021-07-30: Issue is fixed and advisory is published

Summary

ZStack REST API is vulnerable to pre-auth unsafe deserialization

Product

ZStack (https://en.zstack.io/)

Tested Version

3.10.7-c76 (ZStack-x86_64-DVD-3.10.7-c76.iso)

Details

POST requests to the REST API (`/api`) are handled by the `RESTApiController`:

```
    @RequestMapping(value = RESTConstant.REST_API_CALL, method = {RequestMethod.POST, RequestMethod.PUT})
    public void post(HttpServletRequest request, HttpServletResponse response) throws IOException {
        HttpEntity<String> entity = restf.httpServletRequestToHttpEntity(request);
        try {
            String ret = handleByMessageType(entity.getBody());
            response.setStatus(HttpStatus.SC_OK);
            response.setCharacterEncoding("UTF-8");
            PrintWriter writer = response.getWriter();
            writer.write(ret);
        } catch (Throwable t) {
            StringBuilder sb = new StringBuilder(String.format("Error when calling %s", request.getRequestURI()));
            sb.append(String.format("\nheaders: %s", entity.getHeaders().toString()));
            sb.append(String.format("\nbody: %s", entity.getBody()));
            sb.append(String.format("\nexception message: %s", t.getMessage()));
            logger.debug(sb.toString(), t);
            response.sendError(HttpStatus.SC_INTERNAL_SERVER_ERROR, sb.toString());
        }
    }
```

This controller delegates the request body processing to `RESTApiController.handleByMessageType()`:

```
   private String handleByMessageType(String body) {
        APIMessage amsg = null;
        try {
            amsg = (APIMessage) RESTApiDecoder.loads(body);
        } catch (Throwable t) {
            return t.getMessage();
        }

        RestAPIResponse rsp = null;
        if (amsg instanceof APISyncCallMessage) {
            rsp = restApi.call(amsg);
        } else {
            rsp = restApi.send(amsg);
        }
        return JSONObjectUtil.toJsonString(rsp);
    }
```

The request body is then parsed by the `RESTApiDecoder.loads` method:

```
    public static Message loads(String jsonStr) {
        Message msg = self.gsonDecoder.fromJson(jsonStr, Message.class);
        return msg;
    }
```

Which in turn, uses the custom `Message` deserializer to deserialize the request body:

```
        @Override
        public Message deserialize(JsonElement json, Type typeOfT, JsonDeserializationContext context) throws JsonParseException {
            JsonObject jObj = json.getAsJsonObject();
            Map.Entry<String, JsonElement> entry = jObj.entrySet().iterator().next();
            String className = entry.getKey();
            Class<?> clazz;
            try {
                clazz = Class.forName(className);
            } catch (ClassNotFoundException e) {
                throw new JsonParseException("Unable to deserialize class " + className, e);
            }
            Message msg = (Message) this.gson.fromJson(entry.getValue(), clazz);
            return msg;
        }
```

An attacker in control of the request body will be able to provide both the class name and the data to be deserialized and therefore will be able to instantiate an arbitrary type and assign arbitrary values to its fields. Even though GSON does not call any setters on the attacker-controlled object since it uses reflection to set the values of the fields, an attack is still possible if the attacker can find a class with a `finalize()` method that can cause harm. Examples of such classes are memory corruption gadgets or any other classes with undesired side-effects. As an example, an attacker could send the following request:

```
POST http://192.168.78.132:8080/zstack/api
{'java.net.PlainDatagramSocketImpl': {'fd': {'fd': 0,'closed':false}}}
```

ZStack will use GSON to create an instance of `PlainDatagramSocketImpl` where the socket file descriptor is controlled by the attacker (in this case the STDIN (0) file descriptor). Even though the application will throw a `ClassCastException` when casting the deserialized object to `Message` class, the garbage collector will still claim the memory of the allocated `PlainDatagramSocketImpl` object and will call its `finalize()` method. As described here, the `AbstractPlainDatagramSocketImpl.finalize()` method will use a native function to close the attacker-controlled file descriptor. This can be used by the attacker to perform a Denial of Service attack by being able to close all the file descriptors used by the process (using the vulnerability to close all file descriptors in the range 0..20 will most likely cause ZStack to crash).

Impact

This issue may lead to a Denial Of Service. If a suitable gadget is available, then an attacker may also be able to exploit this vulnerability to gain pre-auth remote code execution.

CVE

- CVE-2021-32836

Resources

GitHub Security Advisory

Credit

This issue was discovered and reported by GHSL team member @pwntester (Alvaro Muñoz).

Contact

You can contact the GHSL team at `securitylab@github.com`, please include a reference to `GHSL-2021-087` in any communication regarding this issue.

GitHub

Product

- Features
- Security
- Enterprise
- Customer stories
- Pricing
- Resources

Platform

- Developer API
- Partners
- Atom
- Electron
- GitHub Desktop

Support

- Docs
- Community Forum
- Professional Services
- Status
- Contact GitHub

Company

- About
- Blog
- Careers
- Press
- Shop

- © 2021 GitHub, Inc.
- Terms
- Privacy
- Cookie settings