[EN] A-Z: GWTUpload - DoS

GWT is a Java web framework and GWTUpload is a library extending it with easier file upload. We found a vulnerability allowing to abuse the upload process and cause a denial-of-service of a web application.

GWTUpload is used by adding the library to our GWT project and using its classes directly or by extending them. The class which is responsible for handling file upload request is UploadServlet (https://github.com/manolo/gwtupload/blob/master/core/src/main/java/gwtupload/server/UploadServlet.java). There's also UploadAction (https://github.com/manolo/gwtupload/blob/master/core/src/main/java/gwtupload/server/UploadAction.java) which is built on top of `UploadServlet`, so both of them are interesting for us.

When the servlet receives a `POST` request it invokes a `parsePostRequest` method.

```
709:  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws I
OException, ServletException {
710:    perThreadRequest.set(request);
711:    String error;
712:    try {
713:      error = parsePostRequest(request, response);
714:      Map<String, String> stat = new HashMap<String, String>();
715:      if (error != null && error.length() > 0 ) {
716:        stat.put(TAG_ERROR, error);
717:      } else {
718:        getFileItemsSummary(request, stat);
719:      }
720:      String postResponse = statusToString(stat);
721:      finish(request, postResponse);
722:      renderXmlResponse(request, response, postResponse, true);
```

According to the JavaDoc (https://github.com/manolo/gwtupload/blob/master/core/src/main/java/gwtupload/server/UploadServlet.java#L904): `This method parses the submit action, puts in session a listener where the progress status is updated, and eventually stores the received data in the user session`. Seems unharmful.

GWTUpload uses ServletFileUpload (https://commons.apache.org/proper/commons-fileupload/apidocs/org/apache/commons/fileupload/servlet/ServletFileUpload.html) from Apache Commons which is directly responsible for reading files from an HTTP request. It allows us to set a progress listener, an implementation of the ProgressListener (ProgressListener) interface, which is invoked during the upload to track its process, for example, to implement a dynamic progress bar.

```
904:  protected String parsePostRequest(HttpServletRequest request, HttpServletResponse respon
se) {
[...]
930:    listener = createNewListener(request);
[...]
940:      ServletFileUpload uploader = new ServletFileUpload(factory);
941:      uploader.setSizeMax(maxSize);
942:      uploader.setFileSizeMax(maxFileSize);
943:      uploader.setProgressListener(listener);
944:
945:      // Receive the files
946:      logger.error("UPLOAD-SERVLET (" + session.getId() + ") parsing HTTP POST request ");
947:      uploadedItems = uploader.parseRequest(request);
```

We can notice that GWTUpload creates own listener. To be exact, `createNewListener` can create an instance of one of two classes, but both of them extend AbstractUploadListener (https://github.com/manolo/gwtupload/blob/master/core/src/main/java/gwtupload/server/AbstractUploadListener.java).

So let's see what this listener does when Apache's `ServletFileUpload` updates progress status.

```
 37:  public abstract class AbstractUploadListener implements ProgressListener, Serializable {
[...]
175:    public void update(long done, long total, int item) {
[...]
196:      // Just a way to slow down the upload process and see the progress bar in fast netwo
rks.
197:      if (slowUploads > 0 && done < total) {
198:        try {
199:          Thread.sleep(slowUploads);
200:        } catch (Exception e) {
201:          exception = new RuntimeException(e);
202:        }
203:      }
204:    }
```

That's weird. Usually, we want to process requests as fast as we can, but here we can intentionally slow it down by sleeping the thread whenever the update is invoked. The question that should arise now - who controls the `slowUploads`?

Its value is passed through the constructor.

```
37: public abstract class AbstractUploadListener implements ProgressListener, Serializable {
[...]
68:  public AbstractUploadListener(int sleepMilliseconds, long requestSize) {
69:    this();
70:    slowUploads = sleepMilliseconds;
```

But, if we look back at the `parsePostRequest` method and how listeners are created, we will notice…

```
626:  protected AbstractUploadListener createNewListener(HttpServletRequest request) {
627:    int delay = request.getParameter("nodelay") != null ? 0 : uploadDelay;
628:    if (isAppEngine()) {
629:      return new MemoryUploadListener(delay, getContentLength(request));
630:    } else {
631:      return new UploadListener(delay, getContentLength(request));
632:    }
633:  }
[...]
904:  protected String parsePostRequest(HttpServletRequest request, HttpServletResponse respon
se) {
905:
906:    try {
907:      String delay = request.getParameter(PARAM_DELAY);
908:      String maxFilesize = request.getParameter(PARAM_MAX_FILE_SIZE);
909:      maxSize = maxFilesize != null && maxFilesize.matches("[0-9]*") ? Long.parseLong(maxF
ilesize) : maxSize;
910:      uploadDelay = Integer.parseInt(delay);
```

…that value for `uploadDelay` comes from the request.

So any user who is able to upload files can sleep its own upload thread for any time they want in the range between 0 and 2147483647 (`Integer.MAX`). So, any user can sleep one thread for almost 25 days (2147483647 milliseconds is equal to 596 hours).

Why is that a problem? Java servers do not have the unlimited capability of creating new threads. For example, Tomcat in default configuration (https://tomcat.apache.org/tomcat-8.5-doc/config/http.html) allows to run up to 200 threads. When this limit is reached new threads cannot be created. If a thread is sleeping it also counts to this limit. So, as we can sleep the upload process, we take out one thread from the pool. What if we sent 200 requests? We would exhaust the server's thread pool and no new (any, not only upload) requests will be accepted until sleep ends (reminder: 25 days). In effect, the server would not serve anymore.

But there's one more interesting thing. `uploadDelay` is a field in the servlet. Values of fields in servlets are preserved between requests. So, a malicious user can send just one request with the delay value and it will be also applied to further requests, even those sent by other users (unless someone overwrites it).

The last thing to consider is the size of uploaded files, because if a file is too small `update` method won't be executed at all. The buffer size is hardcoded in `MultipartStream` class in Apache Commons Fileupload

```
    protected static final int DEFAULT_BUFSIZE = 4096;
```

The whole upload request should have at least 4097 bytes.

Demo

So let's try to exploit it in some real application. One of the publicly available (however, not developed anymore) web applications based on GWT and utilizing GWTUpload is Hupa (http://james.apache.org/hupa/), which is part of Apache James project. According to the website:

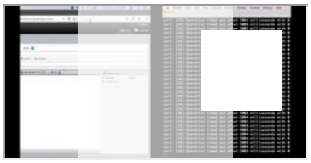
```
Hupa is an Rich IMAP-based Webmail application written in GWT (Google Web Toolkit).

Hupa has been entirely written in java to be coherent with the language used in the James proj
ect. And It has been a reference of a devloping using GWT good practices (MVP pattern and Unit
testing)

Hupa is a functional and well designed email client, ready for reading, sending and managing m
essages, but it still lacks of many features email clients nowadays have.
```

I downloaded and ran on my server the last available build (https://builds.apache.org/job/hupa-trunk/org.apache.james.hupa$hupa/lastSuccessfulBuild/artifact/org.apache.james.hupa/hupa/0.0.5-SNAPSHOT/hupa-0.0.5-SNAPSHOT.war). I went through the application to find the file upload functionality, then I tracked the specific request in

Burp. I reused the URL and session cookie to build a malicious request that I had been repeating until the server stopped responding.

```
curl --data="@file.txt" "http://10.1.1.102:8080/hupa-0.0.5-SNAPSHOT/hupa/uploadAttachmentServlet?delay=2147483647"
```

Also, to confirm what happened, I generated a thread dump (with the command `kill -3 PID`) to be sure that all threads were sleeping.



Who's affected?

Every application utilizing GWTUpload to handle file uploads may be susceptible to the denial of service.

There are a lot of projects on GitHub including GWTUpload library. However, most of them seem to be amateur or abandoned projects.

What now?

GWTUpload itself seems to be not developed anymore (the last commit is dated April 13, 2016). We tried to contact the developer using email address he left on his GitHub profile, however, we haven't received any response. So we submitted the issue (https://github.com/manolo/gwtupload/issues/33) publicly on the GitHub repository.

As we don't expect the vulnerability to be fixed, we would like to show not so elegant solution that everyone can implement.

Fix DIY

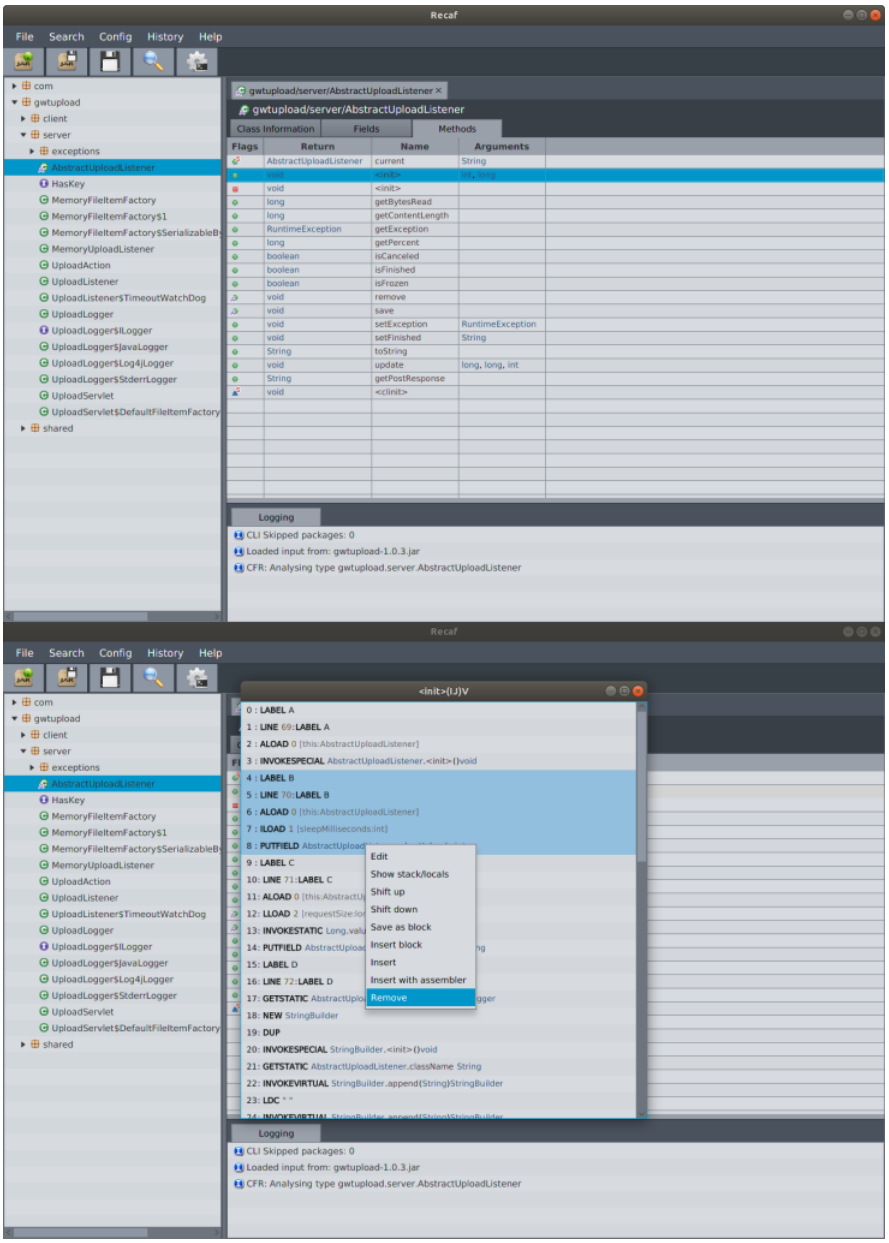
The idea is to modify the code so the `Thread.sleep` method won't be executed. It can be done in many ways but it seems that the easiest one is to remove line 70 in AbstractUploadListener class, so that `slowUpload` will always have default `0` value and because of that threads won't be ever put to sleep.

```
68:  public AbstractUploadListener(int sleepMilliseconds, long requestSize) {
69:    this();
70:    slowUploads = sleepMilliseconds;
```

If we use in our project GWTUpload source code, then the problem is already solved. It's more problematic if we use jar and we don't want to recompile the library. In this case, we need to do some bytecode modification.

There's a tool called Recaf (https://col-e.github.io/Recaf/) which is a Java bytecode editor. What we need to do:

1. Load the jar into the tool
2. Find constructor of AbstractUploadListener
3. Find bytecode corresponding to line 70.
4. Remove bytecode
5. Export jar and replace it in our project.



Bonus

If you looked closely you could see a `PARAM_MAX_FILE_SIZE` parameter that is also truthfully processed by the library. It can be abused to prevent other users from uploading files but without hanging up the whole application. We'll leave to the reader to find out the exact way. ;)

Summary

Some vulnerabilities may exist in third party code which we do not control. However, the implications are the same as with the code we write ourselves. In the case of this library, any user with the possibility to upload files is able to completely neutralize our application. However, vulnerabilities in included third party code are not the only kind of problem. There is also a problem who is responsible for fixing such bugs. If a library is not supported anymore by developers we are left to our own devices. We can try to fix a vulnerability if it is simply as in that case, but when bugs are more complicated we may need to be forced to get rid of the vulnerable library.
Written on February 24, 2020 by Michał Dardas

E-mail*

Telephone (optional)

Message*

I agree to the processing of my personal data and sending the offer.
Rules for the processing of personal data.

send

LogicalTrust sp. z o.o.
sp. k.
NIP: 8952177980
KRS: 0000713515
office@logicaltrust.net **(mailto:office@logicaltrust.net)**
Key: PGP/GPG (/logicaltrust.gpg)

al. Aleksandra Brücknera 25-43
51-411 Wrocław, Poland, EU
T.: +48 71 738 24 35 **(tel:+48717382435)**
K.: +48 514 812 431 **(tel:+48514812431)**

LOGICALTRUST