

Hacking PDF & Javascript for fun and profit

ALEXANDER KLINK
alech@pdftracker.de
<http://www.pdftracker.de>

01.09.2005

Introduction

Using Javascript embedded in PDFs and recent Adobe products, HTTP requests can be sent without user interaction. This is not actually breaking news, but has been known for quite a while. There are even some commercial companies that use these “features” to provide PDF tracking services. After some research, I found out that PDF tracking is not that complicated and tried to implement it. Thus, www.pdftracker.de was born, a site that enables users to embed Javascript code into their documents and track them using a web interface.

If you are interested to see that it works and what is possible, please visit [pdftracker.de](http://www.pdftracker.de) and see for yourself. As an example you can see who has viewed this document at <http://www.pdftracker.de/cgi/view.cgi?documentid=pdftracking>. In the following section, I will present some of the more technical implementation details. I assume that the reader has some knowledge of HTTP, CGI and Javascript.

Implementation

Starting the Javascript code

So you might have guessed that Javascript is mostly used with PDF forms. There, calculations can be made, data can be checked or animations can be played. But then there is always a trigger, like clicking on a button or entering a form field. So how do you start some Javascript code when opening the document? Luckily, there is a way to start a Javascript function without user interaction. Since quite a while, there are so-called additional-actions dictionaries defined in the PDF standard for various PDF objects, such as annotations, pages or even for the entire document. These allow the PDF

writer to specify actions that are taken when an event occurs. For the document, one can define triggers for document opening, closing, printing, saving, etc. On a page level, one can define triggers for opening and closing. This is how pdftracker does it at the moment: once the first page is opened, a Javascript function is called that does all the work. Adding a trigger for document printing or opening would be interesting as well, but this seems to be more difficult to achieve using pdf~~AT~~X.

Sending off the data, one by one

Javascript can be used to submit PDF forms, even if none are embedded in the document. The Javascript function `this.submitform()` is your friend. Called with a URL, it sends a POST request to the server with the form data (in our case, none at all) embedded into an FDF (Form Definition File Format) file. So POST seems to be useless to transfer the data. But we can still use the URL and thus use GET to send information away. Why one by one? Because somehow the clients don't like huge URLs and pop up an error message if the URL is too big (I'd guess that the limit is probably 255 chars, but I have not verified this). We can send away all the data we can get using Javascript, which is quite a lot – see for yourself in the Javascript reference or have a look what pdftracker sends away at the moment at pdftracker.de.

Say cheesecake ^Wcookie!

One interesting feature I was not aware of before is that one can make Javascript variables persistent over application sessions. The client saves them in a special Javascript file which is interpreted at client startup. In this way, a client can be traced over different documents and different networks, similar to the way done with HTTP cookies.

Email for me

Nicely enough, `this.submitform()` also accepts a `mailto:-URL`. Using a session ID and a persistent ID, one can construct an email address where mail is shipped to. This gives us the user's email-address as well as his mail header, which may contain interesting information as well. Doing this without any user interaction works probably only under Unix, where some of the MUAs don't pop up a GUI.

The magic HTTP answer

Once the HTTP request is sent, the server should probably provide an answer. Creating an answer that does not cause the client to do something turned out to be quite tricky. Depending on whether it runs in a browser or not, the behaviour is different. In a browser, the viewer actually loads the returned page in a browser window. HTTP 204

No-Content proved usable there, but created some problems with a stand-alone viewer. The stand-alone viewer does not care that there is “no content” but goes on to read the header, stumbles over a “Content-type: text/html” line and complains that it can not deal with it. Interestingly enough, the only way that it does not complain is when it receives a HTTP 500 response, even though HTML was send back in this case as well. But it looked way to inelegant to distinguish between the different types and act accordingly.

Fortunately, the already mentioned FDF files can also be used to update data in a PDF form. So the idea arose to send back an empty FDF file and hope that the client does not do anything with it. This works fine once you have figured out that the form URL has to end with “#FDF” to enable this feature (even though the “#FDF” is not part of the request the viewer sends away). If you leave it out, the client downloads the file and asks for the document for which this data update is, thus alerting the user of something going on.

Providing regular updates

Nicely enough, Javascript provides a function that enables you to call a function every few milliseconds – `app.setInterval()`. Using this function, you can send out information every few seconds, for example the current page number.

Glueing it all together using pdf_{TeX}, Perl & friends

But how does pdftracker actually create the documents? This is where my favourite typesetting system comes into play – \TeX . Using the `insdljs` package from the Acro \TeX bundle for pdf \TeX , one can insert document-level JavaScript, and using `pdfpages` one can insert PDFs. I also had a look at `PDF::Reuse` for a while, which seems quite nice but I could not figure out how to insert the additional actions dictionary, so I stuck with what I knew best.

The interface and all the rest is a matter of glue, i.e. HTML, Perl, procmail, SQLite, etc.

Conclusion

So what do we learn from this? Javascript in PDFs is evil, disable it or delete the plugin directory. Of course it provides some nice features as well, but its usage is far less widespread than on the web, so disabling should cause little harm.

Note that the warning pop up that is embedded in the documents from pdftracker.de is completely optional – somebody could have been doing this all along without you noticing it. Use an alternative viewer whenever possible.