Don't take LaTeX files from strangers

Steve Checkoway[1], Hovav Shacham[1], and Eric Rescorla[2]

[1]*University of California, San Diego*
[2]*Skype*

April 6, 2011

Abstract

TeX, LaTeX, and BibTeX files are a common method of collaboration for computer science professionals. It is widely assumed by users that LaTeX files are safe; that is, that no significant harm can come of running LaTeX on an arbitrary computer. Unfortunately, this is not the case: In this article we describe how to exploit LaTeX to build a virus that spreads between documents on the MiKTeX distribution on Windows XP as well as how to use malicious documents to steal data from web-based LaTeX previewer services.

1 Introduction

> "I wrote out what I thought I would like to type — how my electronic file should look. And then, I said, OK, that's my input, and here's my output — how do I get from input to output? And for this, well, it looks like I need macros."
>
> Donald Knuth [Thi96]

Donald Knuth's TeX is the standard typesetting system for documents in mathematics and computer science. However, like many other text processing systems designed by computer scientists (PostScript, troff, ...) what it really is is a general purpose programming language specialized for typesetting documents. This is a fact that most TeX users don't think about much, and they (we) tend to treat TeX documents the way they would treat text files — as something inherently safe. Many a user who would never consider downloading and running a random program the Internet doesn't think twice before feeding arbitrary data into his local copy of LaTeX.

TeX is extremely (legendarily) well-designed: Knuth actually gives out cash rewards to people who find bugs, and has made only a few minor changes to TeX in the last decade [Knu08]. As one would expect, TeX generally restricts the functionality that documents and the macros they define can invoke. Nevertheless, it allows macros to read and write arbitrary files. This single capability turns out to be enough to allow a carefully crafted document to completely escape TeX's sandbox. As a demonstration, we present a TeX virus that affects recent MiKTeX distributions on Windows XP, and that, with no user action beyond compiling an infected file, spreads to other TeX documents in the user's home directory. Our proof-of-concept virus carries no malicious payload beyond replicating itself, but it could just as easily download and execute binaries or undertake any other action.

The vulnerabilities exposed by TeX's file-IO capabilities extend beyond a user's personal computer. TeX is the *lingua franca* of mathematics and the mathematical sciences; its notation is frequently used even in communication (like email between collaborators) that isn't meant to be run through the TeX program. And TeX does such a good job of formatting mathematical formulae (and other programs do such a bad job) that it's common to write one's formulae in TeX, render them into images, and them embed them into a Web page, a Word document, or a PowerPoint presentation. A large number of Web-based TeX previewers exist to facilitate the process of turning TEX equations into an embeddable image or PDF. Unfortunately, many of these previewers fail to properly isolate the TEX program with the result that it is possible merely by sending them a malicious document to remotely download sensitive information such as the documents rendered by previous users or even — under the right conditions — the remote system's password file. (Even here, the danger is potentially more widespread. Because the TEX core has been stayed unchanged for many years, making TEX an archival format, many archive services, such as Cornell University's popular arXiv.org, accept submissions in TEX, which they compile to produce PDF.)

It is important to realize that the file IO capabilities at the heart of the vulnerabilities we identify are not bugs in TEX; rather, they are intended capabilities exposed by TEX's macro language that were not fully understood and accounted for by the designers of larger systems (such as online previewers) of which TEX is a component. In this way the vulnerability is of a different kind than the programming error frequently reported in image-handling software (including, in one notorious example, Microsoft Windows' handling of animated cursor files [Sot07]), in which the insufficient validation by the program of attacker-supplied input leads to memory corruption and arbitrary code execution. No such programming error is known in TEX, though Knuth, writing recently, did not disclaim their existence [Knu08]:

> Let me also observe that I never intended TEX to be immune to vicious "cracker attacks"; I only wish it to be robust under reasonable use by people who are trying to get productive work done. Almost every limit can be abused in extreme cases, and I don't think it useful to go to extreme pain to prevent such things. Computers have general protection mechanisms to keep buggy software from inflicting serious damage; TEX and METAFONT are far less buggy than the software for which such mechanisms were designed.

We believe that there are two important lessons to draw. First, one must be cautious about which TEX and LATEX files one compiles. This is actually harder than it sounds: While most people don't routinely compile LATEX source from untrusted sources they do compile BIBTEX entries. (For instance, ACM Portal provides BIBTEX entries for each of its articles.) Because BIBTEX entries can (surprise!) contain LATEX code, this is equally dangerous and much harder to verify, especially if you download large bibliography files such as Joe Hall's well-known election auditing bibliography [Hal10]. This brings us to the second lesson: Executable code is everywhere, even in formats that you would expect to just be passive data. And because it's so difficult to build an effective sandbox, our intuitions about what formats are inert (and hence safe) can lead us very far astray.

2 How To Write a TEX Virus

In this section, we show how to write a virus that is carried in a TEX file.

As explained above, our virus is made possible by the file output capability exposed to TEX documents. Unlike other modern distributions of TEX (see Section 2.3), MiKTEX, the most common TEX distribution for Windows, does not place any meaningful restrictions on this capability.

Given the ability to overwrite system files, it is not surprising that TEX documents can compromise the security of the system on which they are compiled. For concreteness, we focus on one convenient target: On Windows XP, a JScript file written to a user's `Startup` directory will be executed by the Windows Script Host facility at login; the Windows Script Host exposes to scripts COM objects that allow easy manipulation of the filesystem.

Our JScript startup script, when run on the user's next login, seeks out other LATEX files on disk and infects them with our virus. The virus lifecycle is summarized in Figure F.

2.1 Writing the Malicious File

Writing the malicious JScript file is conceptually simple. The TEX `\write` primitive allows us to write data to a file, like so: `\write\file{foo}`. Since we have the malicious JScript embedded in our document we can just `\write` it to disk. However, there is one technical hurdle that must be overcome in order to write to the `Startup` directory: the full path of the directory is `C:\Documents and Settings\Administrator\Start Menu\Programs\Startup` but TEX does not ordinarily allow spaces in file paths (this does not appear to be

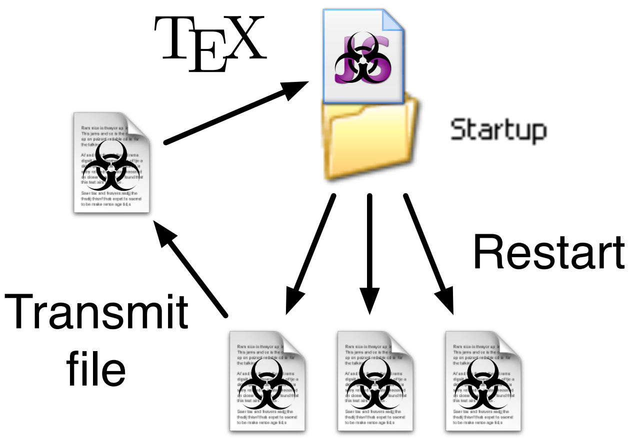

Figure F: LATEX virus lifecycle.

Listing L: Virus code with JScript omitted.

```
%%%%SPLOIT%%%%
{\newwrite\w\let\c\catcode\c`*13\def*{\afterassignment\d\count255"}\def\d{%
\expandafter\c\the\count255=12}{*0D\def\a#1^^M{\immediate\write\w{#1}}\c`^^M5%
\newread\r\openin\r=\jobname \immediate\openout\w=C:/WINDOWS/Temp/sploit.tmp
\loop\unless\ifeof\r\readline\r to\l\expandafter\a\l\repeat\immediate\closeout
\w\closein\r}{*7E*24*25*26*7B*7D\immediate\openout
\w=C:/DOCUME~1/ADMINI~1/STARTM~1/PROGRAMS/STARTUP/sploit.js \c`[1\c`]2\c`\@0
\newlinechar`\^^J\endlinechar-1*5C@immediate@write
@w[fso=new ActiveXObject("Scripting.FileSystemObject");foo=^^J
⟨11 lines of JScript omitted⟩
f(fso.GetFolder("C:\\Documents and Settings\\Administrator"));}m();]
@immediate@closeout@w]}%
%%%%SPLOIT%%%%
```

a security feature, just a functional defect). However, we can leverage Windows' compatibility with older programs that expect file and directory names in 8.3 format. For example, `Start Menu` can be specified as `STARTM~1`. This mechanism allows us to bypass the path restriction.

In addition to the JScript file, we also write a copy of the virus to the disk at an easily accessible location, for use by our JScript in viral spread. For convenience, we just write the entire original document, virus and all. For this, we take advantage of the fact that the TEX engine used in MiKTEX — and indeed in all modern TEX distributions — is pdfTEX which contains the ε-TEX extension `\readline` [TRH$^+$07]. We use `\readline` to read the document being compiled line by line and write an exact copy to `C:\WINDOWS\Temp\sploit.tmp`.

The complete source for the TEX portion of our virus is given in Listing L. We give the details of how it accomplishes the tasks listed above in our technical article [CSR10].

2.2 Spreading the Disease

The second phase, written in JScript, is automatically executed by Windows when the user next logs in. It reads the `sploit.tmp` file, extracts from it the TEX virus code, finds all the files in the `Administrator` directory with the extension `.tex`, and appends the virus onto each of them. To manipulate the filesystem, it instantiates Microsoft's convenient `FileSystemObject`, which exposes a programmatic interface for filesystem search and manipulation.

In total, the virus requires two marker lines and 21 80-column lines of TEX. Listing L omits most of the JScript, in the interest of not providing a complete, working virus; but the remaining code is straightforward and we have tested it in our own systems.

We stress that JScript code run from the filesystem is unsandboxed. Our virus could manipulate the filesystem however it wishes, or download an aribtrary program from the Internet and cause it to be executed. The damage caused by the vulnerability could in principle be far greater than just modifying LaTeX files on disk.

2.3 Applications Outside of Windows

While Windows is the easiest platform to exploit, exploits on other platforms are still possible. As an example, consider the TeX Live distribution popular on UNIX platforms (including Mac OS X). Like MiKTeX, TeX Live allows any file to be read. Unlike MiKTeX, in its default configuration TeX Live prohibits TeX documents from writing to "dotfiles" (files whose names start with a dot, such as `~/.login`, the user startup script for Bourne-derived shells) or files not in the current directory or its subdirectories.

Even with these restrictions, however, there may be avenues for attack. For instance, if a Makefile is being used to run LaTeX then the attacker can overwrite it, inducing arbitrary behavior the next time the `make` program is run. In addition, the popular Emacs-based TeX editing environment AucTeX writes Emacs Lisp cache files to the local directory; an attacker who overwrites these files can execute arbitrary Lisp code inside of Emacs, which itself is Turing-complete and unsandboxed. (For an earlier example of a TeX virus that used Emacs for propagation, see [McM94].)

3 Attacks on Previewers

We now turn our attention to a slightly harder target. There are more than a dozen Web-based services that compile LaTeX files on users' behalf and make the resulting PDFs available. While some of the operators of these sites seem to be dimly aware that attacks may be possible, in nearly every case we were able to read server files remotely and in many cases we were able to write loops that could be used for denial of service via resource consumption. The one previewer we were unable to attack, MathTran [The], uses Secure plain TeX, a reimplementation of plain TeX that prevents using any control sequence other than those meant for typesetting.

We have designed successful exfiltration and denial of service attacks on most of the LaTeX previewer services we studied. Moreover, the filtering mechanisms devised by these services were largely ineffective against our attacks. We disclosed the vulnerabilities of the affected services we found to the operators, with universally positive responses. As a result, a number of operators changed their security policy or removed the previewer altogether.

In the rest of this section we describe some of the details of our attacks.

3.1 Exfiltrating Data

Our key insight is this: Any data that can be read by the TeX script being compiled can be incorporated the PDF file that is its output. When that PDF file is made available to the attacker, he can read it to recover the data. A data exfiltration vulnerability is thus created whenever Web-based TeX previewers allow scripts to read files on disk that are not otherwise made public by the Web server.

This attack can be implemented in a number of ways. The most obvious way uses `\input` to interpolate the text of the file being read into the TeX input and hence the output document. A minor problem with this approach is that it loses line breaks in the input file, since TeX will treat them as spaces in the usual manner. To avoid this, we can instead use the ε-TeX `\readline` extension, as we did in our virus; see Section 2.1. Using this (rarely-used) control sequence also evades any blacklisting of `\input` by the preview service's developers.

In principle, the procedure is straight forward. Our malicious TeX program opens the sensitive file for reading and, in a loop, reads and typesets each line. When the preview service displays the output in the attacker's browser, the contents of the sensitive file are exposed.

For the preview services we examined, the procedure was, in some cases, slightly more complicated. The first barrier to overcome is that many of these previewers are designed to typeset a single equation, and, as a consequence, interpolate the user input into a mathematics environment in an otherwise-complete LaTeX document for processing. Similar in vein to basic SQL injection attacks, the attacker must escape math mode to perform some operations. A further barrier is that some of the preview services explicitly disallow some control sequences such as `\input` or `\include` — rightly recognizing their potential for misuse. This is a very natural defense; however, the availability of other macros for file IO and the malleability of LaTeX code make possible a host of techniques for defeating blacklist or whitelist filters, ranging from using equivalently powerful internal LaTeX macros to exploiting the way TeX parses its input and, in particular, how it decides what is a control sequence. Again, see our technical article [CSR10] for more details.

3.2 Denial of Service

Any previewer that allows the TeX looping construct `\loop`...`\repeat` or the definition of new macros is at risk of a denial of service attack. One can create a simple loop: `\loop\iftrue\repeat`. Or one can define a recursive macro like `\def\nothing{\nothing}`. In the absence of imposed resource limits, enough such loops executed in parallel will slow the server machine to a crawl and no more useful work will be possible until the processes are killed. One extension of this attack is to cause TeX to produce very large files, potentially filling up the disk.

4 The Origins of Insecurity in the Breakdown of the Code/Data Distinction

The vulnerabilities described in the previous sections are an example of a much broader problem: the big shift towards active content. It's common to think of there being a sharp distinction between "code" and "data": code expresses behavior or functionality to be carried out by a computer; data encodes and describes an object that is conceptually inert, and examined or manipulated by means of appropriate code. Programs (Web browsers, word processors, spreadsheets, etc.) are code. Documents (Web pages, text documents, spreadsheet files, etc.) are data, and data is safe.

This distinction is increasingly false. All of the "document" formats mentioned above routinely contain active content (JavaScript, macros, etc.) which is run in the context of whatever program you use to work with the data. When those programs do not properly sandbox the active content, then viewing a seemingly inert document can be just as dangerous as directly executing a program from an unknown source. For example, PDF files can embed JavaScript, which allows PDF files that include malicious JavaScript to exploit bugs in Adobe's Acrobat; by one report [Sca09], some 80% of exploits in the fourth quarter of 2009 used malicious PDF files. Unfortunately, as long experience has shown, proper sandboxing is very hard.

The insecurity we have identified in TeX is one more example of the weakness of this kind of thinking. In TeX, we have a piece of extremely well written software designed for a superficially safe activity (text processing). What's more, whereas PDF files and most other media formats are binary and opaque, the input file formats associated with TeX are all plain text and thus, naïvely, transparent and auditable. Nevertheless, executing TeX files from untrustworthy sources is fundamentally unsafe: Compiling a document with standard TeX distributions allows total system compromise on Windows and information leakage on UNIX. Simply put, every time you compile someone else's LaTeX file or cut-and-paste a BibTeX entry from a Web site you are engaging in unsafe computing.[1] You would do well, as Knuth suggested, to avail yourself of those operating-system protection mechanisms designed "to keep buggy software from inflicting serious damage."

References

[CSR10] Stephen Checkoway, Hovav Shacham, and Eric Rescorla. Are text-only data formats safe? Or, use this LaTeX class file to pwn your computer. In Michael Bailey, editor, *Proceedings of LEET 2010*. USENIX, April 2010.

[Hal10] Joseph Lorenzo Hall. Election auditing bibliography, version 3.8, February 2010. Online: `http://josephhall.org/eamath/bib.pdf` and `http://josephhall.org/eamath/eamath.bib`.

---

[1] The LaTeX source for this article is available from the authors upon request.

[Knu08] Donald E. Knuth. The TEX tuneup of 2008. *TUGboat*, 29(2):233–38, 2008. Online: `http://www.tug.org/TUGboat/Articles/tb29-2/tb92knut.pdf`.

[McM94] Keith Allen McMillan. A platform independent computer virus. Master's thesis, The University of Wisconsin—Milwaukee, April 1994. Online: `http://vx.netlux.org/lib/vkm00.html`.

[Sca09] ScanSafe. Annual global threat report. Online: `http://www.scansafe.com/downloads/gtr/2009_AGTR.pdf`, 2009.

[Sot07] Alexander Sotirov. Windows ANI header buffer overflow, March 2007. Online: `http://www.phreedom.org/research/vulnerabilities/ani-header/`.

[The] The Open University. MathTran – Online translation of mathematical content. `http://mathtran.open.ac.uk`.

[Thi96] Christina Thiele. Knuth meets NTG members, March 13th, 1996. *MAPS*, 16:38–49, 1996. Online: `http://www.ntg.nl/maps/16/15.pdf`.

[TRH+07] Hàn Thế Thành, Sebastian Rahtz, Hans Hagen, Harmut Henkel, Pawł Jackowski, and Margin Schröder. *The pdfTEX user manual*, January 2007.