Transparent file I/O using the original TEX program and the plain TEX format

Udo Wermuth

Abstract

Research papers demonstrate that it is possible to use a TEX file to distribute malware to a victim's system. Although it seems that no report has been published about a virus of this kind in a real attack, the potential danger to abuse a TEX source file to transport unfriendly code exists. This article explains an idea to make TEX's file I/O more transparent and develops requirements to turn the idea into TEX macros. Their application in a TEX file received from an untrusted source identifies all file names used for I/O operations. But the macros demand concentrated work with numerous text inputs and a non-beginner's knowledge of TEX. Furthermore, users should be patient, curious, and courageous.

1 Introduction

The usual input to TEX is a plain text file containing a few control sequences to instruct the program how to format the document. Through its macro capabilities TEX allows an author to increase the number of recognized control sequences, tailoring them to the needs of the text. But TEX does not forbid writing a macro like "`\def\useless{\useless}`" which generates an endless loop when `\useless` appears in the text. (Such endless loops are inherent for a macro expansion language [9, p. 659].) Similarly, some control sequences implemented directly in the TEX program — these are named *primitives* — must be used with care. For example, the simple "`\openout0=\jobname\bye`" truncates the file name to which `\jobname` expands, plus extension `.tex`, with zero bytes. As this is usually the file that TEX processes as the main file in the current run its original contents are gone.

Thus it's easy to waste CPU cycles by executing `\useless`. On a modern multiuser system the single-threaded TEX program occupies at most one CPU and a reasonably configured TEX system doesn't require much main memory. So other users are hardly affected in their own work unless many TEX programs run `\useless` in parallel. To produce a file that should be loaded by `\input` in a co-worker's TEX source file with the above `\openout` statement is a bad joke and might become a disaster if there is no backup of a laboriously created main file. (To protect yourself in such a case from this bad joke set your main file temporarily to read-only, for example, under Unix-like systems with `chmod u-w`.)

These examples raise the question: how brave or careful must one be to typeset a TEX file received from a friendly joker, a well-known silly person, an inexperienced beginner, a person known only by name, or an unknown individual who makes files available for downloading on the Internet. Is it possible that the TEX run of this plain text file results in a damaged or, worse, virus-infected system?

Unfortunately the answer is: Be careful! A TEX run using a specific prepared plain text file might delete important files, read private data, or infect your local system with a computer virus.

Published attacks. The thesis [13] uses LATEX and GNU Emacs to show in a feasibility study that a plain text file can contain code that spreads itself to other plain text files. In [1, 2] an ε-TEX source includes instructions to create during the compilation a JScript file in a certain directory. The execution of this file infects computers running MS Windows — the TEX source contains an absolute path that's only valid for this operating system (OS).

The attacks are possible as TEX contains commands to read from and write to any file. Some implementations of TEX restrict which directories are permitted for TEX's I/O primitives. Of course, every OS should protect itself and mechanisms are usually in effect for ordinary users. But what can be done if the user runs TEX with system administrator rights? Or when the system administrators of a multiuser system that provides a TEX service configured the system in a way that private information is accessible to users without a need to know [12]?

I found no report of any real attack in which someone was the victim of a TEX source file transporting a virus. This risk seems to be very small. But we can assume that some users have coded an endless loop and a few users have deleted an important file with an inappropriate file name for an `\openout`.

Is TEX an insecure program? No, definitely not. Both published attacks need supporting tools: the programmable GNU Emacs or a JScript file placed in an auto-start directory. Similar to an email, TEX source can be abused to transport malicious code. We avoid clicking on a link in an email sent by an unknown person and we must be cautious if we execute a TEX file received from an untrusted source. Sure, TEX could be more verbose with file names. But it doesn't help to learn which file was deleted and it's very cumbersome if TEX asks every time for the user's permission to process a file, as we will see.

It's somewhat pointless to ask today why TEX wasn't programmed with a more restricted access to files. I only provide three observations. First,

doi.org/10.47397/tb/43-1/tb133wermuth-trio

at the time TEX was designed, this program tried to achieve new inconceivable advancements in typesetting. The limits of the available computers were touched; for example, memory had to be conserved. Second, Don Knuth's intention, when he began the design, was to create a tool for his secretary and himself [9, p. 606; 10, p. 63]. There was no reason for mistrust, i.e., bad jokes were not expected. Third, the original TEX was reimplemented as TEX82 and at that time portability was a major concern [8, p. 254]. As file names are highly OS-dependent TEX's code cannot cover all possibilities and must be carefully customized through a *change file* [8, pp. 123–124].

Implementors often transfer TEX's archaic default file system into a nearly unrestricted model for the target OS. But excluding absolute paths or paths containing the short-cut for the parent directory (i.e., "`../`") inhibit the attacks of [1, 2]. The recommendation of [7, §511], to use portable file names built only from letters and digits may be too restrictive, yet reminds us to think about simplicity.

Other risks. Modern TEX implementations, not the original one that is used in this article, activate a communications mechanism to the OS; this feature uses the stream number 18 in `\write` statements. That such a communication makes the life easier for viruses and their developers or *crackers* (to name them in accordance with [15]) has been known for a long time (see [11, p. 454, no. 3]. Thus, the `\write18` feature is often disabled by default and must be explicitly switched on by the user.

A cracker might hide the use of a `\write18`. Therefore, always distrust tricky code without appropriate comments. For example, a single search for `\write18` fails with this obfuscated code; see [14].

```
\lccode`e=`r\lccode`q=`w\lccode`r=`t\lccode`u=`i
\lccode`w=`e\let\ea=\expandafter\lowercase{\ea
\global\ea\let\ea\trouble\csname qeurw\endcsn%
ame}\newcount\maker\maker=9 \multiply\maker by2
\immediate\trouble\maker{echo === GOTCHA ===}
```

All computer users know that all operating systems require regular updating to reduce the risk of a cracker getting into a system through security holes. Additional risks exist that stem from the installation of a distribution (see, for example, [16]) or that are given through the tools of the OS which are required to process a TEX source file and TEX's output; see section 10. From all I know, these risks are much higher than the danger coming from a plain text file containing TEX commands.

Unfriendly code can lurk everywhere. Even if you compile carefully inspected source code yourself, malicious code can be present [18].

Protection by inspection. The abovementioned articles about possible attacks need several lines of TEX code so a look at the source file might reveal the presence of instructions for a virus. But a cracker might try to hide the coded malware. Thus the TEX files one gets from an unknown or untrusted source must either be executed in a restricted environment or be the subject of a thorough visual inspection.

A journal or proceedings editor receives numerous source files and it's unlikely that all authors are known by the editor. On the other hand, the authors want to have their articles published and not be accused of spreading bad code. Nevertheless, an author might be a victim and unknowingly send out a TEX file transporting code for a virus.

Although it's a significant effort, editors should perform a visual inspection as part of the editorial work. I assume that they review text and code in most cases. Besides security, other reasons make this necessary as not all authors are willing to follow the instructions of the journal; some prefer to cheat. For example, look at the report [4] about problems with the length of submitted papers.

Protection by macros. This article describes a set of macros for the original TEX engine with the plain TEX format to make the file I/O operations more *transparent*. By this I mean that a user controls which files are processed when TEX executes `\input`, `\openin`, or `\openout`. The macros don't detect instructions for a virus or state that a file shouldn't be processed; they only report which file names occur and give the user a chance to change them. But they accomplish more: The instrumented source file cannot stealthily bypass their reporting.

One goal of the macros is to produce an identical DVI file compared to a run without the macros if the original source is error-free. Section 3 discusses why this goal cannot be reached for all plain TEX source files; a few eccentric constructions might fail.

Of course, the macros need a few resources. Besides memory space for the macros and other control sequences, the macro package declares five token registers. Thus, one cannot use the macros in the unlikely case that a source file requires more than 238 non-scratch token registers. Sure, the dreadful "TEX capacity exceeded" error message occurs earlier if the macros are used. But this is merely a theoretical problem as modern TEX installations set TEX's compile-time constants so high that it's doubtful that an error-free source reaches TEX's limits even if the macros of this article are active.

Usefulness of the macros. Above I wrote that the risk to become a victim of a virus that enters a

system via a plain TEX file is very small. Nevertheless it might be an interesting intellectual pastime to see how to protect a system with macros against malicious code. Moreover, such macros may reassure people and increase confidence in TEX's security.

A cracker might be aware that these macros exist and avoid conspicuous actions if they are present. Or, say, the code contains a test so that it gets executed only on Sundays and thus a check that runs on a Thursday doesn't detect it. Clearly the macros cannot help to protect a system if they are not active during all executions of a source file.

Although I think a cracker cannot circumvent the macros if the user follows all usage instructions carefully, everyone uses the macros at one's own risk.

2 Primitives requiring file names

With the procedure *scan_file_name* [7, §526] TEX scans in a system-independent way file names. Although file names are highly system dependent, this aspect is handled in other sections of the program. Here I use the convention that a file name consists of an optional path, the main part of the file name, and an optional extension. The path is a sequence of directories with a slash after each directory name; a period separates main part and extension. Spaces are forbidden in file names. A single period in the path, i.e., "`./`", stands for the current directory, and "`../`" represents the parent directory.

The above-mentioned procedure is used in the implementation of four primitives: `\input` in §537, `\font` in §1257, `\openin` in §1275, and `\openout` in §1351.

The primitive `\font` is somewhat special in this list. TEX expects a file name but replaces any extension with `tfm` (§563) as it reads for `\font` only files containing TEX font metric (TFM) data. It checks that the contents of the file with the constructed name obey the specifications of TFM files (§562).

Although this sounds simple it might be very hard to determine which font TEX loads. Above it was shown that the flexibility of TEX can be abused to hide what the code will do. File names are no exception, as the following input proves.

```
\def\gobble#1{r}\lccode`z=`f \lowercase{\edef
\word{zont}}\let\something\futurelet
\expandafter\expandafter\expandafter\let
\expandafter\expandafter\expandafter\futurelet
\expandafter\csname\word\endcsname\def
\lookatnext#1{\romannumeral100\romannumeral1000
\gobble\the\the\count18.\the#1}\futurelet\next
\lookatnext\linepenalty\let\futurelet\something
\lccode`z=`z \show\next
```

What does `\show\next` in the last line display?

I don't see any way to abuse the primitive `\font` to read a file that isn't a TFM file.

Three main primitives. The primitives `\input`, `\openin`, and `\openout` use the complete file name that they receive. They append the extension `.tex` if TEX doesn't find one [6, pp. 25, 217, 226]. With `\input` and `\output`, TEX prompts for a new file name if the file cannot be found or opened for writing, respectively [7, §530, §537, §1374]. The primitive `\openin` never asks the user to enter a new file name [8, p. 325, no. 582 of TEX's error log]. When TEX asks for another file name, the good news is that it displays first "! I can't find file" or "! I can't write on file" followed by the file name that it had scanned. Thus, even if the file name was entered in an obfuscated manner now the user sees the name.

3 Expected problems

Primitives and macros behave differently in a TEX run. If the three file I/O primitives are replaced by macros, under what circumstances does this influence the typesetting? Sure, a source file might test these command names and produce a different DVI file if one of them is a macro. In this case I only care about the result obtained with file I/O macros.

One important difference lies in the ability of macros to expand. The primitives `\openout` and `\openin` are allowed in an `\edef` (or `\xdef`, `\write`, etc.) so the macros should be accepted too. Thus the macros must either contain only expandable tokens and be quite simple or stop the expansion early.

The primitive `\input` is a special case as its acceptance in an `\edef` depends on the contents of the file that is input. TEX usually throws an error, as it treats the end of a file that's input similar to an outer macro [6, p. 206]. But TEX accepts a file that ends with the primitive `\noexpand`. Thus, the macro `\input` must be completely expanded and do its work. But if this macro, say, sets a Boolean flag from false to true, TEX runs into an error if `\input` is executed in an `\edef`. This is completely independent of the contents of the file that gets input.

This is expected, as `\input`'s expansion is null but TEX starts to read from the file [6, p. 214]. Thus, use of `\expandafter` will also give different results. For example, `\expandafter\show\input hello` displays "the letter H" if the file `hello.tex` contains the text "Hello TeX!". But a macro for `\input` expands just one level and TEX displays its first token, i.e., `\show` inactivates this token. (Our macro will start with `\begingroup`; so any control sequence between `\expandafter` and `\input` that reads at least one argument and doesn't open an unclosed

group gives an error.) Similar problems exist with the macros for \openin and \openout.

This "contents dependency" for the acceptance of the primitive \input makes it possible to place it between \csname and \endcsname. TEX allows this if the file that's input expands to character tokens only; \openin and \openout are always rejected. For example, the statement \csname\input hello \endcsname is a valid construction. Usually a macro fails in this scenario if it isn't very simple.

A similar situation occurs with the application of a prefix, \number, etc., to the primitive \input. The first token of the file that's input must accept this command or TEX displays an error; \openin and \openout don't accept such commands.

A reader might agree with me in finding some of these constructions weird and classify them as bad programming practice. Nevertheless the macros will address the four problems: the "\csname problem", the "\edef problem", the "\expandafter problem", and the "apply problem". Some can be solved interactively, others require a change of the source. The important point is: Be alert if a source file uses one of these unusual constructions and check the code carefully to convince yourself that it is required.

Note: The discussion concentrates on plain TEX but, for example, *TUGboat* uses its own macro package in which the command \input becomes a macro. Now, TEX always throws errors for the \edef and \csname problems but not for \global as the macro absorbs it; \long, \number, etc., give errors. Macros with at least three arguments in the \expandafter problem hinder \input.

Privacy. Let's state it frankly: It's not possible to hide the fact that file I/O primitives are replaced by macros. This doesn't mean that all macros must be made public but it means that I decided not to change, for example, \meaning, so a cracker can look at the macro \input. Thus, a cracker knows which control word was given the original meaning of the primitive as it is called in the macro.

The important question is, what can a cracker do with this information? It's suspicious to input a file without using the macro. A user sees on the terminal that TEX inputs a file except if \batchmode is active. My advice: Stop the execution if this happens without the approval through the procedure of the macro described in section 4. Thus the first statements of the macro package are

```
\let\batchmode=\scrollmode
\let\nonstopmode=\scrollmode
```

to make sure that no file can be input without a message on the terminal.

I deactivate \nonstopmode too in order to assure that TEX stops if it cannot find a file as I decided to let \input scan all file names with a trick that makes TEX prompt for a new file name. Then the user has the chance to check which file gets processed and to change the file name if necessary or to end the run. In a second step the file name is given to the primitive whose name occurs in the source to process the file, if the run wasn't canceled.

Another source file might redefine the primitives used in our macros and then they might not do what is intended. This problem gets solved in the usual manner: The used primitives are copied to new control words with a unique start sequence. I use the string "TRIO" for these copies and "TrIO" for all private macros. For example, instead of the primitive \begingroup I use \TRIObegingroup. The source might use the prefix TRIO too, for example,

```
\def\TRIObegingroup{% open three groups
  \begingroup \begingroup \begingroup}
```

(how likely is this?) and our own macro must get a new name, for example, \TRIxObegingroup.

Security. The primitives \openin and \openout are not as verbose as \input. They operate on a file without stating the file name on the terminal (or in the log file). The control words that save the meaning of these primitives must not be made public. Otherwise an evil-doer circumvents the macros and applies the original primitives under their new name.

Fortunately, none of our public macros require the control words with the original meaning of these two primitives as \input is executed first. As mentioned above the file name is read with a trick to make TEX ask for a new file name. The user must enter a special file name that in a next step contains control words that have received via \let the meaning of either \openin or \openout. Therefore these control words can be given what I call a *password-protected* name.

A password-protected name contains a string of at least six letters in upper- and lowercase and with one letter from the first third of the alphabet and another from the last third. If the six letters form neither an English word nor a word in the language of the user it is very unlikely that this control word can be guessed or computed by a cracker. (Six letters define the minimum; use more if you like. Shorter passwords might be discovered with TEX through a brute force attack.) For example, I use in this text the name \TRIOaAmNzZopenin in a \let assignment to save the meaning of the primitive \openin. Note, "aAmNzZ" is a placeholder that must be changed

by the user if the macros are used. First, it's the default that a cracker knows; second, it's much too simple to make a good password.

The macros contain several passwords and some are applied more than once. For example, every used TEX primitive has not only a copy with the prefix "TRIO" but also one with the prefix "TRIOhHJqsS" built with the password "hHJqsS" — again this is a placeholder which must be changed before the macro package is used. During the run a check procedure gets occasionally called to assure that both control words have the same meaning. At the start we define

```
\let\TRIOhHJqsSifx\TRIOifx
\let\TRIOhHJqsSelse\TRIOelse
\let\TRIOhHJqsSfi\TRIOfi
... % many more \let assignments
\def\TrIOhHJqsSstop#1{\TRIOhHJqsSerrmessage{TrIO
 ALERT !!! Don't trust the source (#1)}}
\def\TrIOdDjQwWcheck{% check that macros are OK
 \TRIOhHJqsSifx\TRIOhHJqsSifx\TRIOifx
 \TRIOhHJqsSelse\TrIOhHJqsSstop{TRIOifx}%
 \TRIOhHJqsSfi                  % \TRIOifx is OK
 \TRIOifx\TRIOhHJqsSelse\TRIOelse
 \TRIOhHJqsSelse\TrIOhHJqsSstop{TRIOelse}%
 \TRIOhHJqsSfi                 % \TRIOelse is OK
 \TRIOifx\TRIOhHJqsSfi\TRIOfi
 \TRIOelse\TrIOhHJqsSstop{TRIOfi}%
 \TRIOhHJqsSfi                  % \TRIOfi is OK
 ... }% many more \ifx tests
```

An undetectable problem. As mentioned above the macros for `\openin` and `\openout` input a special file. Changes in the category codes (or *catcodes*) of used characters might change what the file shall accomplish. Thus, I decided to reset all letters and some symbols to their default catcodes before the macros of the special file are executed. This — as well as other decisions like the use of `\count255` — requires executing the code of the macros most of the time inside a group. Sure, `\input` should not load the file inside a group. But `\openin` and `\openout` act globally and can be placed inside a group.

In order to keep such changes local to the group they must not be prefixed by `\global`. The problem occurs if the source sets `\globaldefs=1` because then every assignment, prefixed by `\global` or not, becomes global. Code like this is ok:

```
\begingroup\globaldefs=1 \input hello \endgroup
```

Our macro `\input` sets `\globaldefs=0`, executes its code, and sets `\globaldefs=1`. The first assignment to `\globaldefs` inside the macro, inside the group, is always global. Thus a problem occurs if `\globaldefs` was set to -1 before the above group as then `\globaldefs` is restored as 0 rather than -1 after `\endgroup`. Similarly the code `\globaldefs=1 \begingroup \input hello \endgroup` restores 0 not 1 for `\globaldefs` after `\endgroup`.

A positive `\globaldefs` is rare, and when it does occur it is usually in the good case above. But the problem that arises from the two bad cases can be neither solved nor detected. The macros can only report that `\globaldefs` is positive. The user must then carefully check the source to understand why this seldom-used integer parameter was set.

4 The macro `\input`

Do we need to make `\input` more transparent, as it writes the received file name to the terminal if `\batchmode` is inactive? It's easy to miss one file in a flood of output on the terminal. I prefer to check which files are input and I want to have the control to redirect the request. It is crucial for success to check which files are input. For example, a user must never allow that a source inputs any of the files of the macro package and continues the run.

The trick. How does the macro force TEX to ask for a new file name? A nonexistent path is placed in front of the given file name. For example, I define `\def\TrIOnosubdir{nosubdir/}` where `nosubdir/` must not exist as a directory in the current directory. Next, the macro changes `\input hello` into `\TRIOinput\TrIOnosubdir hello`.

This works fine as long as the file name doesn't start with "`../`" as this might undo in some TEX implementations the "`nosubdir/`" and the remaining path points to a file in the current directory that carries the same name as the file that should be found in the parent directory. In such a case an existing file is input without asking the user. The user sees on the terminal that TEX inputs a file without approval; stop the run and nothing dangerous can happen. Next the replacement text of `\TrIOnosubdir` should be changed to, for example, two nonexistent directories "`nosubdir/nosubdir/`" before a new run is started. It is unusual for the main source to input a file from the parent directory. Be alert if this happens; stop the execution if that still happens after two nonexistent directories are used. The code tries to cope with the definition of `\TrIOnosubdir`.

The macro. This is the main macro:

```
\def\input{% add nonexistent subdir; raise error
 \TRIObegingroup  % next line works in \edef too
 \TRIOafterassignment\TRIOnoexpand\TrIOempty
 \TRIOdef\TrIOskipXXXVIinsTrIOfixedef\TrIOempty
    {}\TrIOempty  \TrIOhandleglobaldefs
 \TRIOglobal\TRIOlet
    \TrIOskipXXXVIinsTrIOfixedef=\TRIOundefined
 \TrIOcountiocmd  \TrIOmessage{<<<}%
```

```
\TrIOsetcatcodes  \TRIOinput TrIOinput.tex
\TrIOinputmessage  \TrIOendgroup
\TRIOinput \TrIOnosubdir}% and file name: error
```

The first line (`\TRIObegingroup`) makes TEX stop if the "apply problem" occurs or if the macro is expanded inside a `\csname`/`\endcsname` structure. Line 2 switches off the application of a token held by the primitive `\afterassignment`; see section 5. The tricky code works in an `\edef` too; see below. The definition of an undefined control word catches the expansion of `\input` in an `\edef`. The macro `\TrIOhandleglobaldefs` handles the `\globaldefs` problem described in the previous section. All these technical parts are discussed in a moment.

The important parts: `\TrIOcountiocmd`, catcode changes in `\TrIOsetcatcodes`, the `TrIOinput` file, `\TrIOinputmessage`, and the last line's trick.

The first macro counts the number of times one of the three file I/O primitives is called.

```
\def\TrIOcnt{0 }\countdef\TrIOcount=255
\def\TrIOcountiocmd{% increment \TrIOcnt
 \TrIOcount=\TrIOcnt \TRIOadvance\TrIOcount by 1
 \TRIOxdef\TrIOcnt{\TRIOnumber\TrIOcount
    \TrIOspace}}
```

Together with information written to the terminal and the `log` file a simplified procedure for repeated execution of the source can be realized; see section 9.

```
\def\TrIOmessage{\TRIOimmediate\TRIOwrite16 }
\def\TrIOinputmessage{% what happens; what to do
 \TrIOmessage{TrIO >>> ( \TrIOcnt) Line
    \TRIOthe\TRIOinputlineno: input}%
 \TrIOmessage{>>> enter shown file name without
    `\TrIOnosubdir'.}\TrIOmessage{<<<}}
```

The catcode changes were mentioned in section 3. The macro prepares to load `TrIOinput.tex`.

```
\def\TrIOsetcatcodes{% establish a few \catcodes
 \TRIOedef\TrIOnext{\TRIOthe\TRIOcatcode`\%}%
 \TRIOcatcode`\%=12 \TRIOlet\%=\TRIOcatcode
 \%`\\=0 \%`\==12 \%`\`=12 \%`\1=12 \%`\2=12 }
```

These catcodes are fixed and build the base for the catcode changes in the file `TrIOinput.tex`:

```
\%`\0=12 \%`\3=12 ... \%`\9=12 \%`\a=11 \%`\b=11
... \%`\z=11 \%`\A=11 \%`\B=11 ... \%`\Z=11
\%`\%=\TrIOnext \TrIOdDjQwWcheck
```

Handling `\globaldefs`. The macro that checks the setting of `\globaldefs` clears it if it is positive as explained earlier. This macro de- and reactivates `\afterassignment` in case it holds a token: The macro `\TrIOsuspendafterassignment` blocks the application of this token after an assignment and the macro `\TrIOinitafterassignment` restores the default behavior. Finally, the macro defines the macro `\TrIOendgroup` that resets the integer parameter `\globaldefs` if necessary after it closes the group opened in the first line of `\input`.

```
\def\TrIOhandleglobaldefs{% inform about
% \globaldefs>0 and switch to \globaldefs=0
\TRIOifnum\TRIOglobaldefs>0 \TrIOmessage
    {TrIO Info: globaldefs is >0 (I/O)}%
 \TRIOafterassignment\TrIOsuspendafterassignment
 \TRIOglobaldefs=0 %       only this is global
 \TRIOdef\TrIOendgroup{\TRIOendgroup
  \TRIOafterassignment\TrIOinitafterassignment
  \TRIOglobaldefs=1 }%
\TRIOelse \TrIOsuspendafterassignment
 \TRIOglobaldefs=0 \TRIOdef\TrIOendgroup{%
  \TRIOendgroup\TrIOinitafterassignment}%
\TRIOfi}
```

A variant. To address some of the problems discussed in the previous section a second macro for `\input` is coded. It carries a password-protected name, `\TrIOcCkPxXinput`, to avoid its unnoticed use. It differs from the macro shown in two respects:

1. The message states "INPUT" instead of "input" to identify itself to the user.
2. In front of `\TRIOinput` in the last line the macro `\TrIOcCkPxXtransfer` appears.

The variant is called if the source file contains `\TrIOcCkPxXmove`. The user must enter this macro into the source to fix some of the discussed problems.

```
\def\TrIOcCkPxXmove#1\input{% transfer tokens
 \def\TrIOcCkPxXtransfer{#1}\TrIOcCkPxXinput}
```

Use this macro only if you are convinced that a `\csname`, `\expandafter`, or "prefix" is required and the source cannot extract the password in the name.

An example. Most macros of this article are bundled in the file `TrIOmacros.tex`. This file is input in the first line of the source file that should be checked.

```
\input TrIOmacros
\batchmode \input hello \errorstopmode
\csname \input hello \endcsname
\TrIOcCkPxXmove\global\input hello
\expandafter\show\input hello
\edef\csone{\input hello }\show\csone\bye
```

When this file is executed TEX displays the messages of the macro followed by an error:

```
<<<
(TrIOinput.tex)
TrIO >>> ( 1 ) Line 2: input
>>> enter shown file name without `nosubdir/'.
<<<
! I can't find file `nosubdir/hello.tex'.
l.2 \batchmode \input hello
                           \errorstopmode
Please type another input file name:
```

This is the normal case: First, a user should check that `TrIOinput.tex` was input, then the macro re-

ports that the first I/O command was found in line 2 and that this command is `\input`, and finally the macro displays what to do next.

We enter "`hello`" as the new file name. TEX shows in the next line that it inputs `hello.tex`. But then an error message pops up.

```
(hello.tex)
! Missing \endcsname inserted.
<to be read again>
                   \TRIObegingroup
\input ->\TRIObegingroup
                          \TRIOafterassignment...
l.3 \csname \input
                   hello \endcsname
?
```

This error message signals the `\csname`/`\endcsname` problem. The answer to the question mark is to type "`42`", then to insert the correct code, i.e., `I\csname`, at the next prompt. Finally, enter the file name.

```
? 42
\input ...\TrIOendgroup
                          \TRIOinput \TrIOnosubdir
l.3 \csname \input
                   hello \endcsname
? I\csname
! I can't find file 'nosubdir/hello.tex'.
l.3 \csname \input hello
                          \endcsname
Please type another input file name: hello
(hello.tex)
```

In this example the `\csname` problem was fixed successfully. But, for example, the code `\csname AA \input hello \endcsname` would create a different typesetting result compared to the original source. Check carefully if the macro `\TrIOcCkPxXmove` can be inserted, if the contents of the file can be typed in, or if the source file should be rejected.

The next line represents such an insertion by the user. Now the "normal" case occurs except that the word "INPUT" signals the use of the macro. (Note the "`2`" as the "`42`" skipped the counting.)

```
<<<
(TrIOinput.tex)
TrIO >>> ( 2 ) Line 4: INPUT
>>> enter shown file name without 'nosubdir/'.
<<<
! I can't find file 'nosubdir/hello.tex'.
l.4 \TrIOcCkPxXmove\global\input hello

Please type another input file name: hello
(hello.tex
```

without the macro TEX reports "! You can't use a prefix with '`\begingroup`'." and the fix is to enter "`42`" and "`I\global`". The apply problem can always be solved in this way.

Next, an error message appears as the contents of `hello.tex` doesn't start with an assignment; it's an error in the original source: "! You can't use a prefix with 'the letter H'.".

After pressing RETURN TEX displays

```
> \TRIObegingroup=\begingroup.
\input ->\TRIObegingroup
                          \TRIOafterassignment...
l.5 \expandafter\show\input
                             hello
```

which is not an error message but the result of the primitive `\show`. Nevertheless the macro `\input` lost its first token. Without intervention TEX will display an error message as soon as it reads the corresponding `\endgroup`. This time the interactive fix is to type "`41`" followed by "`I\expandafter\show`".

With a macro that reads arguments instead of the non-typesetting command `\show` such a fix is not possible. Edit the source and use `\TrIOcCkPxXmove` except in cases like `\expandafter{\input hello }`, in which the `\expandafter` should be deleted.

The macros in `TrIOmacros.tex` are designed in a way that all errors in the original source produce errors in the instrumented file, although the error messages and/or recovery might be different. An erroneous source might lead to an instrumented source in which it is impossible to recover from an error during the execution.

The last line in the above source gives an example of such an error. In the original source TEX displays "Runaway definition?" but the instrumented source shows first "! Undefined control sequence."

```
! Undefined control sequence.
\input ... \TrIOskipXXXVIinsTrIOfixedef
                                          \TrIO...
l.6 \edef\csone{\input
                        hello }\show\csone\bye
```

The name of the undefined control sequence informs the user what to do: Skip 36 tokens and insert then `\TrIOfixedef`. Doing so and after entering "`hello`" TEX displays the original error message.

```
? 36
\input ...\TrIOendgroup
                          \TRIOinput \TrIOnosubdir
l.6 \edef\csone{\input
                        hello }\show\csone\bye
? I\TrIOfixedef
! I can't find file 'nosubdir/hello.tex'.
l.6 \edef\csone{\input hello
                             }\show\csone\bye
Please type another input file name: hello
(hello.tex)
Runaway definition?
->\TRIObegingroup \TRIOafterassignment \ETC.
! File ended while scanning definition of \csone
```

```
<inserted text>
                }
l.6 \edef\csone{\input hello
                               }\show\csone\bye
```

Next TEX complains about too many closing curly braces as in the original source.

The `\show\csone` displays:

```
\TRIObegingroup \TRIOafterassignment \TrIOempty
 \TRIOdef \TrIOempty {}\TRIOendgroup Hello TeX!
```

because of the trick in line 2 and this definition

```
\def\TrIOfixedef{% fix \edef problem for \input
   \TRIOnoexpand\TrIOempty{}\TRIOendgroup}
```

so that `\csone` contains more material than in the source file; a prefix or `\number`, etc., now gives a new error if the original accepts this in front of `\csone`.

Summary: A user can fix the apply problem interactively, but not always the `\csname` and the `\expandafter` problems; one can try to fix them in the source. The `\edef` problem must be fixed interactively but the defined macro has additional tokens.

5 Macro for `\afterassignment`

Next, let's look at the support macros that we need to handle the primitive `\afterassignment`. This primitive stores a single token that isn't expanded [6, p. 215]; thus it can hold an undefined macro and execute it after it was defined. To reproduce this behavior the macro must store the token in a token register and not via a `\let` assignment. On the other hand, a curly brace cannot be placed in a token register; this requires `\let`. To distinguish these cases the macro sets a flag. (`\afterassignment` cannot appear in a `\csname`/`\endcsname` construction or with a prefix like `\global`.)

```
\newif\ifTrIOsavedtoken % true: token is stored
\newif\ifTrIOblockafterassignment % true: don't
          % insert a token after an assignment
\newif\ifTrIOusetokenlist % true: use token reg
\newtoks\TrIOtoken         % the token register
```

A second difficulty is that `\afterassignment` can be used in an `\edef` or `\xdef` but the macro would fail if it is fully expanded. Therefore a second token register is declared to stop the expansion.

```
\let\TRIOafterassignment=\afterassignment
\newtoks\TrIOtrafterassignment % stops expansion
% the replacement of the primitive
\def\afterassignment{% \edef expands one level
 \the\TrIOtrafterassignment}
\TrIOtrafterassignment={\TrIOafterassignment}
```

For the rest of the article — and already in the code just above — I omit the initial "TRIO" if a primitive is meant and no macro replaces it. For example, above I wrote `\the` instead of `\TRIOthe`, but I will still write `\TRIOinput` since the `\input` primitive has been replaced by a macro.

The main macro blocks the usual work of the primitive `\afterassignment` and then fetches via `\futurelet` the token that should be stored. Two of the other three user macros were shown earlier. One sets the flag to block `\afterassignment`, the second removes this block. The third uses the original primitive to call our own insertion macro.

```
\def\TrIOafterassignment{% first save a token
 \begingroup\endgroup          % stop \global
 \TrIOglobaldefs \TrIOsavedtokentrue
 \futurelet\TrIOsavedtoken\TrIOchecktoken}
% user commands for those who know the macros
\def\TrIOsuspendafterassignment{% switch off
 \TrIOblockafterassignmenttrue}
\def\TrIOresumeafterassignment{% switch on
 \TrIOblockafterassignmentfalse % remove block
 \TrIOinitafterassignment}
\def\TrIOinitafterassignment{% init exec macro
 \TRIOafterassignment\TrIOAFTERASSIGNMENT}
```

Again `\globaldefs` must be checked. This is similar to the procedure used for `\input` but here no group must be closed so `\TrIOresetglobaldefs` is defined. It's called when a token must be stored.

```
\def\TrIOglobaldefs{% inform about \globaldefs>0
% and switch to \globaldefs=0 for the macros
\ifnum\globaldefs>0 \TrIOmessage{TrIO Info:
   globaldefs is >0 (store)}%
 \TRIOafterassignment\TrIOsuspendafterassignment
 \globaldefs=0 \def\TrIOresetglobaldefs{%
  \TrIOblockafterassignmentfalse
  \TRIOafterassignment\TrIOinitafterassignment
  \globaldefs=1 }%
\else\ifnum\globaldefs<0 % no group, do a reset
 \TRIOafterassignment\TrIOsuspendafterassignment
 \globaldefs=0 \def\TrIOresetglobaldefs{%
  \TrIOblockafterassignmentfalse
  \TRIOafterassignment\TrIOinitafterassignment
  \globaldefs=-1 }%
\else \TrIOsuspendafterassignment  % switch off
 \def\TrIOresetglobaldefs{% and switch on again
  \TrIOresumeafterassignment}%
\fi\fi}
```

The next macro determines the type of the token and stores it either in a token register or via a `\let` assignment.

```
\def\TrIOchecktoken{% check token, store a macro
 \ifcat\noexpand\TrIOsavedtoken\relax
  \let\TrIOnext=\TrIOstoresavedtoken % a macro
 \else % otherwise remove token from the input
  \let\TrIOnext=\TrIOremovesavedtoken
 \fi \TrIOnext}
\def\TrIOstoresavedtoken#1{% #1: cs in token reg
 \let\TrIOnext=\undefined \TrIOusetokenlisttrue
 \TrIOtoken={#1}\TrIOresetglobaldefs}
```

```
\def\TrIOremovesavedtoken{% remove a token
 \let\TrIOnext=\undefined \TrIOusetokenlistfalse
 \TRIOafterassignment\TrIOresetglobaldefs
 \let\TrIOsavedtoken=}
```

The application macros just test the flags.

```
\def\TrIOAFTERASSIGNMENT{% use the stored token
 \ifTrIOblockafterassignment% true nothing to do
 \else  % otherwise output token if one is saved
  \ifTrIOsavedtoken \ifnum\globaldefs>0
    \TrIOmessage{TrIO Info: globaldefs is 1
         (apply)}\globaldefs=0 %        clear it
    \TrIOsavedtokenfalse \globaldefs=1 % & reset
   \else \TrIOsavedtokenfalse \fi
  \expandafter\expandafter          % get rid of
  \expandafter\TrIOoutputtoken       % the 2 \fi
  \expandafter\fi        % with 3+1 \expandafter
 \fi}
\def\TrIOoutputtoken{% output token (check type)
 \ifTrIOusetokenlist       % true: use token reg
  \expandafter\the\expandafter\TrIOtoken
 \else           % otherwise use the saved token
  \expandafter\TrIOsavedtoken
 \fi}%     no need to change \ifTrIOusetokenlist
```

6 Macro for \openin

Let's repeat what we already know about `\openin`. It's nicer than `\input` as it can't occur in a `\csname`/`\endcsname` construction. Moreover, it can't be prefixed by `\global` as the equals sign here does not mean an assignment is performed; it's an association between a stream number and a file name. This association acts globally so that we can execute `\openin` inside a group. To solve the `\expandafter` problem in the source just delete this token. But `\openin` might be part of an `\edef`. Thus, the technique of the previous section is applied for `\openin` too.

But `\openin` is also much more unpleasant than `\input`. It operates without stating the file name on the terminal or in the `log` file. Thus, the control word that saves the meaning of the primitive must not be made public. Otherwise an evil-doer could circumvent the macro and apply the original primitive under its new name. Therefore the copy of the primitive is assigned a password-protected name: `\TRIOaAmNzZopenin`.

The macro `\openin` first reads the stream number, next a test is made to see if the optional equals sign follows, and third `\TRIOinput` is called with the trick so that TEX asks for a new file name. But this time the user enters two file names. First, a generic file name — for `\openin` it's by default `openin` — and then the file name that should be processed by the primitive `\openin`. The file `openin.tex` contains several password-protected macros that do the important work. Please remember: A user must never allow `TrIOmacros.tex` or any other file of this package, such as `openin.tex`, to be processed by the original source.

All aspects of the following macros are either well-known or have been discussed.

```
\newtoks\TrIOtropenin % token register for \edef
\def\openin{\the\TrIOtropenin}% expand one level
\TrIOtropenin={\TrIOopenin}% call the main macro
\def\TrIOopenin{\begingroup
 \TrIOhandleglobaldefs \TrIOcountiocmd
 \TRIOafterassignment\TrIOOpenIn \TrIOcount=}
\def\TrIOOpenIn{% remove an optional =
 \TRIOafterassignment\TrIOOPENIN
 \let\TrIOnxt=}
\def\TrIOOPENIN{% add nonexistent directory
 \TrIOmessage{<<<}%     first: the instructions
 \TrIOmessage{TrIO >>> (\TrIOcnt) Line
  \the\inputlineno: openin \the\TrIOcount}%
 \TrIOmessage{>>> If you accept that the file
  (without \TrIOnosubdir) is read}%
 \TrIOmessage{>>> enter 'openin' and
  follow the instructions.}\TrIOmessage{<<<}%
 \ifx=\TrIOnxt \def\TrIOnxt{}%
 \fi    % otherwise \TrIOnxt <> '='; so keep it
 \TrIOsetcatcodes     % required for openin.tex
 \TRIOinput\TrIOnosubdir\TrIOnxt}
```

In `openin.tex`, private information is used: a kind of signature that it is the user's `openin.tex` and not one by a cracker. A user should change the text to make it unique for each installation. But of course, use only characters whose category codes are known, i.e., set in the list. As `\TrIOnext` becomes undefined in the macro `\TrIOaAmNzZopenin` the message stays private.

```
\%‘\0=12 \%‘\3=12 ... \%‘\9=12 \%‘\a=11 ...
\%‘\z=11 \%‘\A=11 ... \%‘\Z=11 \%‘\>=12
\%‘\{=1 \%‘\}=2 \%‘\%=\TrIOnext \TrIOdDjQwWcheck
\TRIOgGKptTpausing=1 \def\TrIOnext{My message
Enter 1> return 2> file name}\TRIOgGKptTpausing0
\TrIOaAmNzZopenin
```

The mentioned macro prompts for the file name and calls the password-protected primitive using the stream number stored in the register `\TrIOcount`.

```
\def\TrIOaAmNzZopenin{% get file name from user
 \read16 to \FilenameOPENIN
 \TRIOaAmNzZopenin\number\TrIOcount=
                                \FilenameOPENIN
 \let\FilenameOPENIN=\undefined
 \let\TrIOnext=\undefined
 \let\TrIOnxt=\undefined
 \TrIOendgroup}% see \TrIOhandleglobaldefs
```

7 Macros for \openout and \immediate

The macros to replace the primitive `\openout` are very similar to the ones used for `\openin`; and the

file `openout.tex` is similar to `openin.tex`. The only aspect not yet discussed is the "prefix" `\immediate`.

TEX allows an `\immediate` everywhere without raising an error for the next token. This is in contrast to, for example, the prefix `\long` that, after expansion of the next token, requires a definition primitive (`\def`, etc.) or another prefix (`\global`, etc.). Although `\immediate` never complains, it influences the next token after expansion only if it is one of `\openout`, `\write`, or `\closeout`.

The way `\immediate` operates means that the macro that replaces the primitive cannot simply set a flag that signals that it was seen. For example, the sequence "`\immediate\begingroup\openout`" would then faultily apply `\immediate` to `\openout`. Can we just test if the macro `\openout` follows the macro `\immediate`? I decided to put an identification primitive at the start of `\openout` so that `\TrIOopenout` doesn't start with `\begingroup` but with the sequence "`\TRIOimmediate\begingroup`".

The macros for `\immediate`. As indicated above, the first part of the macros uses the known structure. Only the last line of `\TrIOImmediate` contains a new technique (or trick).

```
\newif\ifTrIOimoo    % true: \immediate\openout
\newtoks\TrIOtrimmediate % token reg. for \edef
\def\immediate{\the\TrIOtrimmediate}% one level
\TrIOtrimmediate={\TrIOImmediate}%    expansion
\def\TrIOImmediate{% expand the following token
 \begingroup \TrIOhandleglobaldefs
 \TRIOafterassignment\TrIOIMMEDIATE
 \TrIOcount=‘x}% the trick; explanation follows
```

TEX treats the alphabetic constant `‘x` like a number and digests a space after such a number [7, §442]. To check if a space follows, tokens are expanded (§443) but TEX doesn't add anything to the alphabetic constant. Thus TEX assigns the value 120 to `\TrIOcount` after it determines the first token of the expansion of the token that follows `\immediate`. Only if this first token is `\TRIOimmediate` does the source contain `\openout` as an interim next token for `\immediate` during the expansion.

```
\def\TrIOIMMEDIATE#1{% #1: a token; it’s tested
 \ifx#1\TRIOimmediate    % true: macro \openout
  \global\TrIOimootrue      % follows; set flag
 \else \global\TrIOimoofalse \fi \TrIOendgroup
 \TRIOimmediate#1}% apply the primitive
```

A cracker might set the flag (either directly or via `\TRIOimmediate` as the names aren't protected) to confuse the user. The next `\openout` will use the flag even if no `\immediate` precedes it. Stop the execution if `TrIOmacros` reports "immediate openout" but the source file seems to have no `\immediate` in front of `\openout`. Then check the source carefully.

The macros for `\openout`. As written above, the macros for `\openout` are so similar that they aren't shown here in detail. Besides the wording "openout" instead of "openin" and "created" instead of "read" in the messages there are two differences:

1. `\TrIOopenout` starts with `\TRIOimmediate`;
2. the first message in `\TrIOOPENOUT` contains now "`\ifTrIOimoo immediate \fi`" in front of the string "`openout`".

A new password-protected macro is called in `openout.tex`; it makes use of the new flag. Otherwise `openout.tex` is identical to `openin.tex`.

```
\def\TrIObBlOyYopenout{% get file name from user
 \read16 to \FilenameOPENOUT
 \ifTrIOimoo \global\TrIOimoofalse
  \let\TrIOnext=\TRIOimmediate % use \immediate
 \else \let\TrIOnext=\relax \fi
 \TrIOnext\TRIObBlOyYopenout
    \number\TrIOcount=\FilenameOPENOUT
 \let\FilenameOPENOUT=\undefined
 \let\TrIOnext=\undefined
 \let\TrIOnxt=\undefined  \TrIOendgroup}
```

8 The virus example

The following instructions are a modified version of the code containing the virus shown in [1] and [2]. This badly formatted, comment-free but obfuscated code should alert everyone who sees it. (I changed the original source so that it can be executed under plain TEX. Moreover, the original file names and in one case the contents of a file were changed.)

```
1. \input TrIOmacros % new 1st line; see below
2. \newif\ifcontinue \continuetrue
3. \def\uncat{\def\do##1{\c‘##1=12 }\dospecials
4. \do\^^M\do\*}\def\nice{\endlinechar=‘\^^M
5. \uncat}\def\readline#1to#2{\begingroup\nice
6. \global\read#1to#2\endgroup}%
7. {\newwrite\w\let\c\catcode\c‘*13\def
8. *{\afterassignment\d\count255"}\def\d{%
9. \expandafter\c\the\count255=12}{*0D\def%
10. \a#1^^M{\immediate\write\w{#1}}\c‘^^M5%
11. \newread\r\openin\r=\jobname
12. \immediate\openout\w=../justafile.tex
13. \loop\ifeof\r\continuefalse\fi\ifcontinue
14. \readline\r to\l\expandafter\a\l\repeat
15. \immediate\closeout
16. \w\closein\r}{*7E*24*25*26*7B*7D\immediate
17. \openout\w gotcha.tex \c‘[1\c‘]2\c‘\@0
18. \newlinechar‘\^^J\endlinechar-1*5C@immediate
19. @write@w[What have I done?]@immediate
20. @closeout@w]}%
21. \bye
```

As in the example of section 4 the file got a new first line "`\input TrIOmacros`". Next we run TEX on this file, which I call `danger.tex`. TEX quickly

stops to display a message. (Some lines are broken for *TUGboat*'s column width, and the identifying password in the name for `\pausing` was deleted.)

```
<<<
TrIO >>> ( 1 ) Line 11: openin 0
>>> If you accept that the file (without
    nosubdir/) is read
>>> enter 'openin' and follow the instructions.
<<<
! I can't find file 'nosubdir/danger.tex'.
<to be read again>
                   \begingroup
\TrIOImmediate ->\begingroup
                              \TrIOhandlegl...
l.12 \immediate
               \openout\w=../justafile.tex
Please type another input file name:
```

Don't get confused by the shown source lines. TEX detects that it has the complete file name only after seeing the `\immediate` in line 12. The "`TrIO >>>`" line shows the number of the I/O command, the line number in which it was found, and the command itself. The first file I/O is in line 11 and the command is `\openin` with stream number 0. After the instructions TEX displays the file name that it read plus the nonexistent subdirectory that our macros added. Here the source looks for the file `danger.tex`, i.e., itself. Although I find it weird for a file to read itself, this process is harmless compared to a file that wants to destroy itself. So I continue; that is, I enter "openin", press return, check my private message, press return, and enter the file name.

```
Please type another input file name: openin
(openin.tex
Enter 1> return 2> file name}\TRIO...pausing0=>
\FilenameOPENIN=danger
```

Next, TEX stops again. As expected it is the second file I/O command and this time it's `\immediate \openout` with stream number 0. The source wants to write a file in the parent directory. This is very strange and shouldn't be allowed. I prefer to create a subdirectory `trioo/` and to redirect all output files to this directory. Of course, the user must remember which files are placed in this subdirectory if the source wants to read one of them again.

```
<<<
TrIO >>> ( 2 ) Line 12: immediate openout 0
>>> If you accept that the file (without
    nosubdir/) is created
>>> enter 'openout' and follow the instructions.
<<<
! I can't find file 'nosubdir/../justafile.tex'.
l.12 \immediate\openout\w=../justafile.tex

Please type another input file name: openout
(openout.tex
Enter 1> return 2> file name}\TRIO...pausing0=>
\FilenameOPENOUT=trioo/justafile
```

Maybe you directly saw in the source that a path contains two periods. To avoid the case that TEX inputs an existing file `justafile.tex` in the current directory, add in front of `\input TrIOmacros` `\let\twonosubdirs=y` to have `\def\TrIOnosubdir {nosubdir/nosubdir/}` as explained earlier.

The third stop is similar to the second except one should check that `\immediate` occurs at the end of line 16. Again I use the output directory `trioo`.

```
<<<
TrIO >>> ( 3 ) Line 17: immediate openout 0
>>> If you accept that the file (without
    nosubdir/) is created
>>> enter 'openout' and follow the instructions.
<<<
! I can't find file 'nosubdir/gotcha.tex'.
l.17 \openout\w=gotcha.tex
                           \c'[1\c']2\c'\@0
Please type another input file name: openout
(openout.tex
Enter 1> return 2> file name}\TRIO...pausing0=>
\FilenameOPENOUT=trioo/gotcha
```

At the end of the run the user should check the files in the subdirectory `trioo`. This reveals that `justafile.tex` is a copy of `danger.tex`.

9 Repeated executions

Although the macros work well, a user needs to concentrate during the stop-and-go operation and thus it's easy to make mistakes. A run is ruined if the user enters, for example, the file name instead of `openout` at a stop for `\openout`. No harm to the system is done as TEX reads the file; the creation of a file is only possible through the file `openout.tex`.

As soon as one manages to finish a successful run the package provides macros to avoid the input of file names in subsequent runs if the I/O commands and the file names aren't changed from run to run. These macros use the I/O commands with the file names entered in the successful run in exactly the order they occured previously. A run is deemed successful if and only if TEX doesn't report an error that was interactively fixed. To activate the macros for repeated executions a user has to do the following.

1. Copy the `.log` file of the successful run. For example, copy `danger.log` to `danger.trio`.
2. Run a `sed` command on the copied `log` file. Use `TrIOlineno.sed` (or `TrIOextract.sed`) to create another TEX file called `TrIOnames.tex`. For example, enter: `sed -f TrIOlineno.sed danger.trio > TrIOnames.tex`.

3. Change the first line of the instrumented source file; replace `TrIOmacros` by `TrIOauto`.

The `log` file contains in the lines that start with "`TrIO >>> ...`", "! I can't find file ...", and "`\FilenameOPEN...`" all the data needed to create a case statement in TEX, in which for each sequence number the line number, the I/O command, and the file name can be combined to do the file I/O automatically; the uppercase form of "input" is thereby changed to "`\TrIOcCkPxXtransfer \TRIOinput`".

The difference between the two `sed` files is that in one the new line number and the line number of the successful run are compared. This exact replication of the successful run might be too strict if the user has to edit the text but doesn't change the sequence of I/O commands. A user can create a new `TrIOnames.tex` by using `TrIOextract.sed` instead of `TrIOlineno.sed` in step 2 of the above list.

The case statement is placed in a password-protected macro stored in `TrIOnames.tex`. Here is the structure of this file from the run of section 8.

```
\def\TrIOeEMnvVfilenames{% use files of prev run
 \ifcase\TrIOcnt \iffalse   % a technicality
  \else\TrIOstop{case ( \TrIOcnt) in auto}\fi
 \or\ifnum\TrIOcount=11 %             case 1
   \def\TrIOiocmd{\TRIOaAmNzZopenin 0}%
   \TrIOenvopen \def\TrIOfile{danger}%
   \TrIOmessage{TrIO >>> ( 1 ) Line 11:
      openin 0 \TrIOfile}%
  \else \TrIOstop{case ( \TrIOcnt) in auto}\fi
 \or\ifnum\TrIOcount=12 %             case 2
 ...
 \else
  \TrIOstop{unknown case ( \TrIOcnt) in  auto}%
 \fi \TrIOfFLouUexecute}
```

The macro `\TrIOenvopen` provides some definitions for an "environment" to end the current group for `\openin` and `\openout`. For `\input` the group must end before it gets active.

```
\def\TrIOenvopen{\let\TrIOleft=\relax
 \let\TrIOright=\TrIOendgroup}
\def\TrIOenvinput{\let\TrIOleft=\TrIOendgroup
 \let\TrIOright=\relax}
```

The new macros. The file `TrIOauto.tex` contains simplified macros for `\input`, `\openin`, and `\openout`. It uses the file `TrIOopen.tex` to load and write the files in `TrIOnames.tex`. The new file `TrIOopen.tex` is like `openin.tex` or `openout.tex` except that it doesn't contain a personal message and that it calls `\TrIOeEMnvVfilenames`, not the password-protected copies of `\openin` or `\openout`.

The macro for `\input` no longer writes terminal messages with `\TrIOmessage`; this also applies to all other file I/O macros in `TrIOauto.tex`.

```
\def\input{\begingroup \TrIOhandleglobaldefs
 \TrIOcountiocmd \TrIOsetcatcodes
 \TrIOcount=\inputlineno   % see \TrIOfilenames
 \let\TrIOnxt==%         needed in \TrIOexecute
 \TRIOinput TrIOopen.tex }
```

The variant with a password-protected name, `\TrIOcCkPxXinput`, isn't needed anymore because the macro `\TrIOcCkPxXmove`, which might still occur in the source, now calls `\input`.

For `\openin`, two of the four macros are unchanged. In `\TrIOopenin` the line number is saved (as in `\input`) so that it becomes available in the macro `\TrIOeEMnvVfilenames`. The other changes in this set of macros are similar to the changes seen in the new macro `\input`.

```
\def\openin{\the\TrIOtropenin}
\TrIOtropenin={\TrIOopenin}
\def\TrIOopenin{\begingroup
 \TrIOhandleglobaldefs \TrIOcountiocmd
 \xdef\TrIOnext{\TrIOcount=\the\inputlineno}%
 \TRIOafterassignment\TrIOOpenIn \TrIOcount=}
\def\TrIOOpenIn{\TRIOafterassignment\TrIOOPENIN
 \global\let\TrIOnxt=}
\def\TrIOOPENIN{\TrIOnext \TrIOsetcatcodes
 \TRIOinput TrIOopen.tex }
```

The macros for `\openout` and `\immediate` receive drastic changes: `\openout` becomes identical to `\openin` and `\immediate` isn't replaced by a macro.

The execution macro. The last line in the macro of `TrIOnames.tex`, i.e., in `\TrIOeEMnvVfilenames`, calls a password-protected macro that executes the stored file I/O command.

```
\def\TrIOfFLouUexecute{% prepare I/O execution
 \ifx=\TrIOnxt \gdef\TrIOnext{TrIO_}%
 \else \gdef\TrIOnext{TrIO_\TrIOnxt}\fi
 \TRIOafterassignment\TrIOfFLouUdoiocmd % exec
 \font\unused=\TrIOnext}% remove file name
```

The last line might be a surprise. Why do we need a `\font` command here? Now that the file name from the input isn't used for an I/O command the source contains an unread file name. I decided to read and display the file name so that a user can check that the file name agrees with the one used in `TrIOnames.tex`. It's possible that a cracker codes something like "`\input\myfile`" and changes file names in `\myfile` from run to run. Although our macros use a name that was approved they can still help the user to identify such sources.

Thus the file name should be displayed. But with `\input` and the trick the user must enter another file name, for example, `null`. To reduce this to a simple `return` I apply the primitive `\font` and a prefix for the file name to avoid loading a TFM file if the file name is, for example, called `cmr10.tex`.

TEX raises an error message that shows the file name without extension; see section 2. After a quick check that the main parts of the known file name and the shown one without `TrIO_` agree, the user continues the run by pressing return. Next the I/O command is executed; as mentioned earlier, `\input` outside the group, `\openin` and `\openout` inside the group.

```
\def\TrIOfFLouUdoiocmd{% execute the I/O command
 \let\TrIOnext=\undefined
 \TrIOresumeafterassignment
 \ifx\TrIOright\relax \expandafter\TrIOleft
  \expandafter\TrIOiocmd \expandafter\TrIOfile
 \else \TrIOiocmd\TrIOfile\TrIOright \fi}
```

For example, TEX's first message for the source `danger.tex` of section 8 with `TrIOauto.tex` is:

```
(TrIOopen.tex
TrIO >>> ( 1 ) Line 11: openin 0 danger
)
! Font \TRIOunused=TrIO_danger not loadable:
Metric (TFM) file not found.
<to be read again>
                   \immediate
l.12 \immediate
               \openout\w=../justafile.tex
```

Although it is quite unusual the source might contain something like "`\input file.tex at`" and then TEX interprets the "`at`" as a keyword if the input `file.tex` is treated as the name of a font. In such a case the user should change the source and place the "`at`" in curly braces; treat the keyword "`scaled`" in the same way. With `TrIOauto.tex` the repeated execution isn't a big problem.

10 Treatment of `\special`

The previous sections introduce macros that allow a user to control which external files TEX reads and writes. But by default TEX writes data to two other files: the `log` file and the DVI file.

The `log` file is a plain text file like the TEX source. It is neither interpreted nor compiled.

The DVI file is a binary file that must be interpreted by a device driver. Most of its content is determined by the encoding of the text which TEX has to typeset. But TEX also contains the primitive `\special` that is able to write any data to the DVI file. The device drivers must know what to do with this data.

Some device drivers support a `\special` string being executed as a shell command; this scenario has the same risks as the `\write18`. Or the device driver may interpret data as PostScript instructions. PostScript code can delete files, spread a virus, or hide private data inside the PostScript file — later the author can extract this information if the user returns its output; see [5, chap. 4]. The macros of this article cannot control the actions of shell scripts or PostScript code.

It is strongly recommended to activate the security options of the device driver if a DVI file from an untrusted source is processed even if the source was compiled by oneself. For example, use `-safer` in `xdvi` [3] and `-R2` for the DVI-to-PostScript translator `dvips` [17].

Macros for `\special`. By default the macros assume that the user configures the device drivers to protect the system. That is, `TrIOmacros.tex` and `TrIOauto.tex` keep the primitive `\special` active.

But the macros offer a way to look at the data contained in a `\special` without touching the primitive. TEX puts a marker for the `\special` and the associated token list into a so-called *whatsit* [6, p. 226] that appears in the box that TEX ships out. TEX writes all token lists into the `log` file (sometimes in an abbreviated form, see [7, §292]) with:

```
\tracingoutput=1
\showboxdepth=10000 \showboxbreadth=10000
```

The `log` file might now become very large! The user must search or extract the data to check what the unknown token lists contain. For example,

`grep -e'^\.\.*\\special'` ⟨*logfile*⟩

extracts the beginning of the token lists of all specials in the `log` file ⟨*logfile*⟩.

Of course, the source might set the above integer parameters to other values and we disable this by assigning `\tracinglostchars` via `\let` to the three parameters. But a source file that, for example, relies on the fact that one of the values of the three integer parameters has its default value — 0, 3, or 5, respectively — might now produce unintended output. Again an unusual case; reject the source.

Besides the possibilities of keeping the primitive untouched in TEX or tracing `\special`'s actions, the package offers to deactivate `\special` and to trace all complete token lists in the `log` file.

```
\def\TrIOwlog{\TRIOimmediate\write-1 }
\def\special{\TrIOwlog{<<< TrIO >>>
   Line \the\inputlineno: special}\TrIOwlog}
```

A user starts the described tracing via either `\let\disablespecial=n` or `y` before reading one of `TrIOmacros.tex` or `TrIOauto.tex`, with or without executing the primitive `\special`.

11 Final remarks

The shown code snippets introduce all password-protected names, in total eight. The package consists of ten main files and to change these passwords

in all of them is therefore a laborious job. To automate this task I added two more files: a `sed` file to change the passwords and a shell script to apply the `sed` file to the ten files. Remember: It's crucial that each installation has its own passwords.

Before files of your run are returned to the author (1) delete the new first line and all inserted macros `\TrIOcCKpxXmove` in the source; (2) check the `log` file for tracing output containing password-protected macro names; (3) look at the DVI output to avoid the unlikely case that it contains information about the new macros.

I described scenarios in which the macros fail but remember these are all exotic cases — the author is playing tricks on you. That's why I wrote to inspect or reject the source file. I assume a cracker avoids these exotic cases; no one wants to attract attention to one's harmful code.

If you want to use the macros and you provide a macro package to authors think about code like

```
\let\TeX@input=\input \let\globaldefs=\undefined
\def\input{\begingroup\def\undefinedinput{}%
 \endgroup\TeX@input}
```

so that then error-free sources avoid most problems.

Can the program TEX adopt these ideas? No. We can't deactivate `\batchmode` or stop the run to reenter a file name for `\input` without violating the TRIP test [9, p. 572]. But it's okay to exclude certain paths and to reenter names of certain files. Only when a file with such an excluded path occurs is the user asked to enter a new name or reenter the then-accepted file name that appeared in the TEX file.

References

[1] Stephen Checkoway, Hovav Shacham, and Eric Rescorla, "Are Text-Only Data Formats Safe? Or, Use This LaTeX Class File To Pwn Your Computer", *Proceedings of LEET '10, USENIX* (2010), 8 pp. usenix.org/legacy/events/leet10/tech/full_papers/Checkoway.pdf

[2] Stephen Checkoway, Hovav Shacham, and Eric Rescorla, "Don't take LaTeX files from strangers", *;LOGIN:* **35**:1 (2010), 17–22. usenix.org/system/files/login/articles/73506-checkoway.pdf

[3] Eric Cooper, Bob Scheifler, Mark Eichin, Paul Vojta, et al., *Xdvi man page*, Xdvik version 22.87.04, February 29, 2020, 34 pp. tug.org/texlive/Contents/live/texmf-dist/doc/man/man1/xdvi.man1.pdf

[4] Wouter Duivesteijn, Sibylle Hess, Xin Du, "How to cheat the page limit", *WIREs Data Mining and Knowledge Discovery* 2020;10:e1361, 9 pp. doi.org/10.1002/widm.1361

[5] Markus Dürmuth, *Novel Classes of Side Channels and Covert Channels*, Ph. D. thesis, Saarland University, Saarbrücken (2009), 146 pp. publikationen.sulb.uni-saarland.de/bitstream/20.500.11880/26018/1/Dissertation_1920_Duer_Mark_2009.pdf

[6] Donald E. Knuth, *The TEXbook*, Volume A of *Computers & Typesetting*, Boston, Massachusetts: Addison-Wesley, 1984.

[7] Donald E. Knuth, *TEX: The Program*, Volume B of *Computers & Typesetting*, Boston, Massachusetts: Addison-Wesley, 1986.

[8] Donald E. Knuth, *Literate Programming*, Stanford, California: Center for the Study of Language and Information, CSLI Lecture Notes No. 27, 1992.

[9] Donald E. Knuth, *Digital Typography*, Stanford, California: Center for the Study of Language and Information, CSLI Lecture Notes No. 78, 1999.

[10] Donald E. Knuth, *Companion to the Papers of Donald Knuth*, Stanford, California: Center for the Study of Language and Information, CSLI Lecture Notes No. 202, 2011.

[11] Joachim Lammarsch, "VM/CMS site report", *TUGboat* **11**:3 (1990), 454–455. tug.org/TUGboat/tb11-3/tb29site.pdf

[12] Guilhem Lacombe, Kseniia Masalygina, Anass Tahiri, Carole Adam, Cédric Lauradoux, "Can You Accept LaTeX Files from Strangers? Ten Years Later", arXiv:2102.00856v1 [cs.CR], 2021, 10 pp. arxiv.org/abs/2102.00856

[13] Keith Allen McMillan, *A platform independent computer virus*, M. Sc. thesis, University of Wisconsin, Milwaukee (1994), ix+28 pp. ftp://coast.cs.purdue.edu/pub/doc/viruses/KeithMcMillan-PlatformIndependantVirus.ps

[14] Scott Pakin, reply to "Malicious commands in LaTeX", `comp.text.tex`, August 7, 2008. groups.google.com/g/comp.text.tex/c/epWW3eV9udw

[15] Eric S. Raymond with Guy L. Steele Jr., eds., *The New Hacker's Dictionary*, 3rd ed., Cambridge, Massachusetts: MIT Press, 1996. catb.org/esr/jargon/

[16] Red Hat Customer Portal: *RHSA-2012:0137 – Security Advisory*, 15 February 2012. access.redhat.com/errata/RHSA-2012:0137

[17] Tomas Rokicki, *Dvips: A DVI-to-PostScript Translator*, version 2021.1, February 2021, 62 pp. ctan.org/pkg/dvips

[18] Ken Thompson, "Reflections on Trusting Trust", *CACM* **27**:8 (1984), 761–763. dl.acm.org/doi/pdf/10.1145/358198.358210

◇ Udo Wermuth
Dietzenbach, Germany
u dot wermuth (at) icloud dot com