low level TEX security

Contents

1 Preamble

Here I will discuss a moderate security subsystem of LuaMetaTEX and therefore ConTEXt LMTX. This is not about security in the sense of the typesetting machinery doing harm to your environment, but more about making sure that a user doesn't change the behavior of the macro package in ways that introduce interference and thereby unwanted side effect. It's all about protecting macros.

This is all very experimental and we need to adapt the ConTEXt source code to this. Actually that will happen a few times because experiments trigger that. It might take a few years before the security model is finalized and all files are updated accordingly. There are lots of files and macros involved. In the process the underlying features in the engine might evolve.

2 Flags

Before we go into the security levels we see what flags can be set. The TEX language has a couple of so called prefixes that can be used when setting values and defining macros. Any engine that has traditional TEX with ε-TEX extensions can do this:

```
                \def\foo{foo}
\global         \def\foo{foo}
\global\protected\def\foo{foo}
```

And LuaMetaTEX adds another one:

```
      \tolerant          \def\foo{foo}
\global\tolerant         \def\foo{foo}
\global\tolerant\protected\def\foo{foo}
```

What these prefixes do is discussed elsewhere. For now is is enough to know that the two optional prefixes `\protected` and `\tolerant` make for four distinctive cases of macro calls.

But there are more prefixes:

`frozen`	a macro that has to be redefined in a managed way
`permanent`	a macro that had better not be redefined
`primitive`	a primitive that normally will not be adapted
`immutable`	a macro or quantity that cannot be changed, it is a constant
`mutable`	a macro that can be changed no matter how well protected it is
`instance`	a macro marked as (for instance) be generated by an interface
`noaligned`	the macro becomes acceptable as `\noalign` alias
`overloaded`	when permitted the flags will be adapted
`enforced`	all is permitted (but only in zero mode or ini mode)
`aliased`	the macro gets the same flags as the original

These prefixed set flags to the command at hand which can be a macro but basically any control sequence.

To what extent the engine will complain when a property is changed in a way that violates the above depends on the parameter `\overloadmode`. When this parameter is set to zero no checking takes place. More interesting are values larger than zero. If that is the case, when a control sequence is flagged as mutable, it is always permitted to change. When it is set to immutable one can never change it. The other flags determine the kind of checking done. Currently the following overload values are used:

	immutable	**permanent**	**primitive**	**frozen**	**instance**
1 warning	*	*	*		
2 error	*	*	*		
3 warning	*	*	*	*	
4 error	*	*	*	*	
5 warning	*	*	*	*	*
6 error	*	*	*	*	*

The even values (except zero) will abort the run. In ConTEXt we plug in a callback that deals with the messages. A value of 255 will freeze this parameter. At level five and above the `instance` flag is also checked but no drastic action takes place. We use this to signal to the user that a specific instance is redefined (of course the definition macros can check for that too).

So, how does it work. The following is okay:

```
\def\MacroA{A}
\def\MacroB{B}
```

```
\let\MyMacro\MacroA
\let\MyMacro\MacroB
```

The first two macros are ordinary ones, and the last two lines just create an alias. Such an alias shares the definition, but when for instance `\MacroA` is redefined, its new meaning will not be reflected in the alias.

```
\permanent\protected\def\MacroA{A}
\permanent\protected\def\MacroB{B}
\let\MyMacro\MacroA
\let\MyMacro\MacroB
```

This also works, because the `\let` will create an alias with the protected property but it will not take the `permanent` propery along. For that we need to say:

```
\permanent\protected\def\MacroA{A}
\permanent\protected\def\MacroB{B}
\permanent\let\MyMacro\MacroA
\permanent\let\MyMacro\MacroB
```

or, when we want to copy all properties:

```
\permanent\protected\def\MacroA{A}
\permanent\protected\def\MacroB{B}
\aliased\let\MyMacro\MacroA
\aliased\let\MyMacro\MacroB
```

However, in ConTEXt we have commands that we like to protect against overloading but at the same time have a different meaning depending on the use case. An example is the `\NC` (next column) command that has a different implementation in each of the table mechanisms.

```
\permanent\protected\def\NC_in_table   {...}
\permanent\protected\def\NC_in_tabulate{...}
\aliased\let\NC\NC_in_table
\aliased\let\NC\NC_in_tabulate
```

Here the second aliasing of `\NC` fails (assuming of course that we enabled overload checking). One can argue that grouping can be used but often no grouping takes place when we redefine on the fly. Because `frozen` is less restrictive than `primitive` or `permanent`, and of course `immutable`, the next variant works:

```
\frozen\protected\def\NC_in_table   {...}
```

```
\frozen\protected\def\NC_in_tabulate{...}
\overloaded\let\NC\NC_in_table
\overloaded\let\NC\NC_in_tabulate
```

However, in practice, as we want to keep the overload checking, we have to do:

```
\frozen\protected\def\NC_in_table   {...}
\frozen\protected\def\NC_in_tabulate{...}
\overloaded\frozen\let\NC\NC_in_table
\overloaded\frozen\let\NC\NC_in_tabulate
```

or use `\aliased`, but there might be conflicting permissions. This is not that nice, so there is a kind of dirty trick possible. Consider this:

```
\frozen\protected\def\NC_in_table   {...}
\frozen\protected\def\NC_in_tabulate{...}
\def\setNCintable   {\enforced\let\frozen\let\NC\NC_in_table}
\def\setNCintabulate{\enforced\let\frozen\let\NC\NC_in_tabulate}
```

When we're in so called `initex` mode or when the overload mode is zero, the `\enforced` prefix is internalized in a way that signals that the follow up is not limited by the overload mode and permissions. This definition time binding mechanism makes it possible to use `permanent` macros that users cannot redefine, but existing macros can, unless of course they tweak the mode parameter.

Now keep in mind that users can always cheat but that is intentional. If you really want to avoid that you can set the overload mode to 255 after which it cannot be set any more. However, it can be useful to set the mode to zero (or some warning level) when foreign macro packages are used.

3 Complications

One side effect of all this is that all those prefixes can lead to more code. On the other hand we save some due to the extended macro argument handling features. When you take the size of the format file as reference, in the end we get a somewhat smaller file. Every token that you add of remove gives a 8 bytes difference. The extra overhead that got added to the engine is compensated by the fact that some macro implementations can be more efficient. In the end, in spite of these new features and the more extensive testing of flags performance is about the same.[1]

[1] And if you wonder about memory, by compacting the used (often scattered) token memory before dumping I manages to save some 512K on the format file, so often the loss and gain are somewhere else.

4 Introspection

In case you want to get some details about the properties of a macro, you can check its meaning. The full variant shows all of them.

```
% a macro with two optional arguments with optional spacing in between:

\permanent\tolerant\protected\def\MyFoo[#1]#*[#2]{(#1)(#2)}

\meaningless\MyFoo\par
\meaning    \MyFoo\par
\meaningfull\MyFoo\par
```

macro:[#1]#*[#2]->(#1)(#2)
tolerant protected macro:[#1]#*[#2]->(#1)(#2)
permanent tolerant protected macro:[#1]#*[#2]->(#1)(#2)