How to deal with Trojan Source vulnerability

11/15/2021 12:08 AM - duerst (Martin Dürst)

Status: Feedback
Priority: Normal
Assignee: 
Target version: 

Description
The "Torjan Source" vulnerability recently has caught some attention.

The vulnerability involves using certain combinations of Unicode characters to let source code look like it is correct (and therefore pass code review,...) but actually do something else than intended.


I contacted the Ruby security list, which was already aware of the issue, and we agreed to discuss this here because the vulnerability is already public.

The paper focuses on the use of [A] Directional Formatting Characters (*1) in string constants, comments, and similar constructs to change the visual appearance of code outside these constructs. There are related vulnerabilities, namely the use of [B] non-spacing (and therefore mostly invisible) characters e.g. in variable names, and the use of [C] mixed-script identifiers, which also lets some variable names look identical even if they are not.

Some languages, such as Rust, have addressed [A] (see https://blog.rust-lang.org/2021/11/01/cve-2021-42574.html) by requiring escapes to be used for the relevant characters in source. On the other hand, people such as Russ Cox think compilers are the wrong place to address the issue; it should be addressed in editors and similar tools (see https://research.swtch.com/trojan). Github now warns about

The question is what Ruby should do, if anything. Addressing [A] similar to how Rust does it can be done relatively easily. If that's done, I'd prefer to only reject incomplete Bidi control sequences, which is a bit more complicated. In particular, string interpolation needs a very careful analysis.

For [B], I'll open a separate issue.

For [C], we have all data about scripts, but the way it's currently structured makes finding out which character a script belongs to quite inefficient.

(*1) "Directional Formatting Character" is the official Unicode term (see https://www.unicode.org/reports/tr9/#Directional_Formatting_Characters). The terms "Bidi/Bidirectional control" or "Bidi/Bidirectional control character" are also used. Overall, there are 9 such characters. Unfortunately, both the paper and KrebsonSecurity use the term "Bidi Override", which is highly misleading. The term "Bidi Override" is reserved for two characters only: LRO, U+202D, Left-to-Right Override, and RLO, U+202E, Right-to-Left Override (see Table 1 in the paper). It is also used for the phenomenon associated with these two characters, a “hard” override (i.e. affecting all characters including e.g. the Latin alphabet), and mechanisms in other technology that achieve the same (e.g. the HTML bdo element (https://html.spec.whatwg.org/#the-bdo-element) or the ‘bidi-override’ value of the unicode-bidi property in CSS (https://www.w3.org/TR/CSS2/visuren.html#propdef-unicode-bidi)).
Just FYI: Rubocop has an issue to address this issue, [https://github.com/rubocop/rubocop/issues/10226](https://github.com/rubocop/rubocop/issues/10226)

#4 - 11/15/2021 10:19 AM - duerst (Martin Dürst)
mame (Yusuke Endoh) wrote in #note-3:

I'm afraid if prohibiting or warning bidi characters may bother programmers who use Arabic and/or Hebrew.

This is a very good point. I'm not an actual user of the Arabic or Hebrew script (or some other RTL (right-to-left) script such as Syriac,...), but I have done standards work and research in this area, so I'll answer from this perspective.

First, the bidi control characters are not needed to just write a comment in any of these scripts. A comment purely in Arabic or a string purely in Hebrew will automatically be displayed RTL. Bidi controls may be needed if the comment also contains LTR (left-to-right) characters (e.g. Latin or Kanji/Kana), but still can mostly be avoided.

Bidi embeddings (U+202A/U+202B) are only needed if you have a structure of LTR inside RTL inside LTR, but that shouldn't be needed for most comments, and if it looks like it may be needed, it should be possible to avoid this by using more than one line.

Bidi overrides (U+202D/U+202E) are only needed for fixed hard ordering, with the main application I have heard of being part numbers that may contain characters from various scripts. They may also be helpful e.g. if somebody wants to nail down the exact visual output expected.

Bidi isolates (U+2066/U+2067/U+2068) are a relatively new addition. Their main use is as replacement for bidi embeddings, or to isolate pieces with a fixed internal order from outside texts. Their typical use is e.g. when adding items from a database into text in an user interface. So I expect them to appear quite a bit in string interpolations, but in that context, having them escaped would probably help the programmer.

Also, a big advantage is that program text does not get reflowed. Bidi controls are much more important for reflowed text (e.g. in documents or web pages); for texts with fixed linebreaks, some "cheating" is possible (just put pieces of text on the line so they show up the way you want).

On the other hand, programming includes a lot of symbol characters. Many if not most of these symbol characters are 'weak' in the bidi algorithm, i.e. they take their directionality from the surrounding alphabetic characters. But in programming languages, these symbol characters are actually the characters that determine the overall syntax.

This may lead to problems in comments that could be addressed by using bidi controls. More often, it may lead to problems if RTL text is used e.g. for variable names. As an example, the following Ruby fragment has one comment and one variable name in Arabic. The comment looks fine, the assignment of 20 to the variable ﻹ٠ will need some time to get used to. But that's not the problem in this issue.

```
book = 20
kitb = 20
```

The original paper says that if the bidi control characters are nicely grouped (i.e. each of the opening characters mentioned above is followed by the respective closing character (either PDF, U+202C, or PDI, U+2069), then there is no vulnerability. So it may be possible for comments and strings without interpolations to check for that condition. But I would first like to verify this claim; bidi can be quite tricky.

#5 - 11/22/2021 02:55 AM - duerst (Martin Dürst)

- Status changed from Open to Feedback

We discussed this at the developers' meeting on 2021/11/18. No final decision was taken. We think that this issue should primarily be addressed by editors and similar tools, by making the relevant characters visible.

We will see what other languages do; currently, the picture is mixed, with a tendency to leave it to editors,... The only language that we know of that has reacted is Rust. Any feedback is appreciated.

#6 - 11/23/2021 08:39 PM - Dan0042 (Daniel DeLorme)

In a sense it's true this is the responsibility of the editor, but I also think it's ok to have defense in depth. I would support some form of customizable blacklist of "dangerous" unicode characters that are not allowed in source code, with some sane default.

```ruby
$UNICODE_BLACKLIST #=> #<Set: {0x202D, 0x202E}> #by default blacklist bidi overrides?
$UNICODE_BLACKLIST << 0x3164 #worried about that "invisible variable" exploit
$UNICODE_BLACKLIST.delete(0x202D).delete(0x202E) #if you need bidi overrides
```