L2/22-007R2

Avoiding Source Code Spoofing

From: Mark Davis, Robin Leroy, Peter Constable, Markus Scherer

Date: 2022-01-25

Summary

This is a proposal to form a working group / task force focusing on providing guidance for dealing with the so-called Trojan Source exploit, where a developer interprets what they see as a different sequence of tokens than what the compiler actually processes. While we have long-standing documentation on dealing with bidirectional behavior and confusables, this would focus on source code. This task force would deliver its results to the Properties and Algorithms working group of the Unicode Technical Committee for review, and eventually to the UTC for approval.

We would look to involve people who are experts in the Unicode encoding, bidirectional algorithm, and security concerns/mechanisms, as well as experts in programming language standardization and tooling and security experts. The proposal also provides a (rough draft) sketch of what such types of guidance could look like.

Note: Copying text from this PDF may result in incorrect bidi display.

Contents

Issue

There have been reports recently about problems in review of source code containing non-ASCII Unicode characters. The basic problem is that two ***different*** lines of code (in memory) can have the ***same*** (or confusingly similar) appearance on the screen. That is, the actual text is different from what the reader perceives it to be. The person reviewing a submission of code from a contributor could be fooled into thinking that the code was ok, when it was really malicious.

This can result from stateful bidirectional controls (used in Arabic, Hebrew, and other right-to-left scripts) that change the natural ordering of characters.

Misleading text can also result from a string containing Arabic or Hebrew characters, from "hidden" characters (such as a zero-width space), or from confusable characters or sequences of characters (Greek omicron vs Latin o). These are the known consequences of the human writing systems. Using BIDI or confusable characters to mislead users has been documented in a number of specifications and technical reports for many years: UAX #9 ("Bidi"), UAX #31 ("Identifiers"), UTR #36 ("Security"), and UTS #39 ("Security Mechanisms").

In particular, the stateful bidi controls behavior has been present and documented in the Unicode encoding for quite a while: since the very first version (1991), so it is not new.

https://xkcd.com/1137/

2012-11-22

Indeed, the developers of some tools have taken notice of these documents, and mitigated some of the issues described therein:

- The Rust compiler warns against confusable identifiers occurring in the same library[1]:
  CONFUSABLE_IDENTS in rustc_lint::non_ascii_idents - Rust.
  It also warns against scripts introduced into a library solely with the use of characters that are confusable with ones from already-used scripts: MIXED_SCRIPT_CONFUSABLES in rustc_lint::non_ascii_idents - Rust.
  It also warns against characters with Identifier_Status=Restricted, *i.e.*, it follows UTS #39 C1:
  UNCOMMON_CODEPOINTS in rustc_lint::non_ascii_idents - Rust
- An OCaml package that adds support for Unicode identifiers warns not just about confusable identifiers, but about identifiers that mix scripts without separating them by U+005F LOW LINE: whitequark/ocaml-m17n: Multilingualization for the OCaml source code.

[1] An exception is made for confusable Basic Latin, e.g., 0 and O, or I, l, and 1. This may mean that an alternate, narrower, definition of confusability is needed for source code, as suggested in a swift-evolution discussion when confusables were brought up for that language. Such a definition might treat p (Latin) and ρ (Greek) as non-confusable, though it would treat p (Latin) and p (Cyrillic) as confusable. Alternatively, it may mean that some exceptions in the case of ASCII are needed for other reasons. This needs further investigation.

- Visual Studio orders tokens[2] left-to-right[3], *i.e.*, it implements UAX #9 HL4, see the discussion above https://twitter.com/KhaledGhetas/status/993118048575021058.
- Both the Swift and Rust compilers make use of confusable detection to provide better error messages, *e.g.*, when General Punctuation “quotation marks” are mistakenly used in place of their Basic Latin "counterparts": https://bugs.swift.org/browse/SR-331.

The relative scarcity of these examples shows, however, that we can’t expect developers of compilers or source code editors to find that material and interpolate how it would apply to their specific domain. It is often tricky to find a balance between security measures and usability, when deep knowledge of Unicode is not normally a requirement for compiler developers. So we should do a better job about providing more focused documentation for such developers.

While the main focus is security, this guidance can also be useful in improving the usability of code editors and language tooling when non-ASCII characters can be in strings, comments, and identifiers such as variable names. Some people may be surprised that identifiers are included, but many modern programming languages allow for non-ASCII identifiers, including Java, Rust, Swift, Go, and many others.

[2] Technically units smaller than tokens: escape sequences within string literals are isolated and ordered as tokens would be.
[3] Right-to-left token order is not supported: https://docs.microsoft.com/en-us/visualstudio/ide/use-bidirectional-languages#right-to-left-reading-order.

Proposed Plan

We put together a work group / task force of experts to work on the following tasks. We create a Unicode email group where we can invite security experts and others to participate, and also schedule regular meetings for f2f discussion. It is anticipated that this task force would be of limited duration.

The goals would be to:

A. Engage with MITRE to get more accurate wording into the CVE records
B. Assemble documentation providing guidance for avoiding spoofing issues. Make that available for review and feedback.
C. Produce Unicode documentation, such as draft proposed updates of UAX #9 ("Bidi", aka UBA), UAX #31 ("Identifiers"), UTR #36 ("Security"), and UTS #39 ("Security Mechanisms") using the information in B, and post for comment.
    - We would not be changing the UBA, but otherwise exactly where in these documents additions would be made is as yet unclear. For example, #39 and #36 could have new sections, while the others would point to the new sections at appropriate places.
    - However, we may also add lengthier sections in multiple places, such as a new section of #9 describing issues in the display order of program text. The key will be to have new material in a location where the desired audience could find it.
D. In ICU, respond to tickets filed, and provide code snippets and/or APIs to implement utility functions that could be used directly to help avoid problems. (The implementations could also be ported to other languages.) Some areas are:
    - Determine where the ordering of programming language tokens on a line would not be visually monotonic.
    - Given the visual appearance of a line, determine where tokens would appear to have different boundaries.
    - Given a token, determine when a different memory representation could appear identical (or nearly so). For example, the visual token **a21א** could be the result of displaying three different in-memory tokens: **a21א**, **a2א1** and **aא21**.
E. Examine whether new properties and/or property values, or changes to values, would be useful.

Preliminary Ideas

We would expect to have progressive iterations of the documentation for coverage and comprehension among experts in Unicode and security. Here are some preliminary thoughts.

There are three different areas where problems can be addressed:

- **High-level linters** — These are linters that would check the contents of source code files in a repository, but with little to no knowledge of the particular programming language of the file, though they would at least need to know that it is programming language source code.
- **Compilers** — These include compilers and linters that are specific to particular programming languages.
- **Source Code Editors** — These include source-code editors and code review applications.

Some of the solutions can be implemented in multiple areas, not just one. In practice, it is best to implement whatever can be done in *all* of these areas, because source code can be in multiple repositories, be compiled by different compilers (even for the same programming language), and viewed in different source code editors.

Much of the text below focuses on viewing source code where the overall direction of text is left-to-right. Most of the discussion can also be applied where the overall direction of text is right-to-left, *mutatis mutandis*. There are, however, some additional wrinkles for right-to-left text in that most programming languages have ASCII keywords, forcing a larger percentage of text to be in the opposite direction of the overall direction.

However, since the goal is to prevent cases where the memory representation may be not what people expect, we have to take into account the fact that different people have different expectations. A person whose native language is written with right-to-left characters may have quite different expectations of order than one whose native language is written with left-to-right characters.

Notation

The examples are presented in the following format. The first section of three rows shows the in-memory representation of the line of text (also called the 'logical order'). The *memory* row has the characters, the *index* row has the starting index of each character in memory, and the *token#* row has a token number (a segment of text treated as atomic by a compiler), where the number is the number of the token in memory: 0 is first, 1 is second, and so on.

In the following, the characters in token #4 (indices 4-6) are Hebrew characters in memory order (Arabic would also work, except that the cursive nature of the characters makes it a bit less obvious what is going on). Highlighting is used to show areas of interest.

memory:	s		=		א	ב	ג	-	1	0	0	;
index:	0	1	2	3	4	5	6	7	8	9	10	11
token#:	0	1	2	3	4			5	6			7

visual:	s		=		1	0	0	-	ג	ב	א	;
index:	0	1	2	3	8	9	10	7	6	5	4	11
token#:	0	1	2	3	6			5	4			7

The second section shows the visual appearance. The visual row shows the order the user would see. The index row again shows the character indexes in memory : so the Hebrew character א again has the index 4. The *token#* row also shows the token with their in-memory order.

Notice that not only are the Hebrew characters reversed, but the tokens are also reordered on the line. Token 6 is coming right after token 3, then token 5, then token 4.

High-Level Linters

There are some changes that could be made at a very high level, perhaps without even knowing what the program language is. This includes raising errors on broad classes of characters present in the source files or in easily-identified sections thereof.

The goal of such high-level linters is to be able to quickly add support for many languages, making it possible to process large repositories in short order.

While it should be possible to have any Unicode character in source code, a high-level linter can require some of those characters to be escaped in programming language source code, which prevents their use for spoofing. For example, the stateful bidirectional characters are invisible, but affect the order of text around them.

Alternatively, a linter could disallow them except in end-of-line comments, where the reordering would be confined to the comment; this would make it possible to use those characters to get comments with bidirectional text to render as desired.

```
span.set('dir', 'ltr')  # הצג את הטקסט לפי UAX9, סעיף HL4.
```

The above comment reads ".HL4 section ,UAX9 display the text according to". Wrapping the comment in a right-to-left embedding (or isolate, if supported) fixes the problem:

```
span.set('dir', 'ltr')  # .HL4 סעיף ,UAX9 הצג את הטקסט לפי
```

One point many reports have overlooked is that stateful bidi controls are *not* the only cause of reordering issues — they do not have to be present for bidi reordering problems to occur! Problems can occur without them, as the examples under Compilers below illustrate. However, requiring the stateful controls to be escaped takes care of a large swath of possible ordering problems.

Importantly, the stateful bidi controls and other characters below are not in Unicode on a whim; they are needed to properly display plain text. So the escaping approach *must not* be applied to files that don't contain source code, such as plaintext or markdown.

The following are candidates for required escaping:

- Deprecated characters

- Controls (except common whitespace)
- Format characters (with certain exceptions)
- Private use
- Surrogates
- Unassigned code points (including non-characters)
- Misleading whitespace, including the "blank" Hangul fillers

Most of these are orthogonal to the reordering problems, but can also contribute to reducing confusables.[4] Note that only linters that are updated regularly to the latest version of Unicode should disallow unassigned code points.

There is a broad spectrum of levels of understanding of the programming language which can be used in such a linter.

The better the understanding of the language, the more actual issues can be detected, and the more remedies can be implemented without harming usability; some potentially interesting levels are:

1. Knowing nothing about the structure of the file, except that it is in a programming language for which this kind of linting is appropriate.
   - By definition, this is the minimum level of understanding at which something could be done.
2. Identifying comments and string literals;
   - A linter at this level is needed to diagnose situations where reordering causes comments or strings to look like executable code, and vice versa;
   - it could forbid more characters outside of comments or strings, while not getting in the way of legitimate usage of plain text within them.
   - ➢ This level remains very quick to implement, as only two types of tokens, both delimited, need to be taken into account for each language.
3. Identifying comments, string literals, and identifiers;
   - A linter at this level could start considering mixed-script or confusable detection;
   - it could, in most languages, check the exit directionality of all strongly directional tokens, avoiding token reordering (see next section).
   - ➢ Implementing a linter at this level requires implementing the definition of identifiers in each language, whether based on UAX#31 or something else, *e.g.*, general categories.
4. Full lexing.
   - ➢ Linters at that level (and above, if parsing or even semantic analysis are also used) are discussed in the next section.

[4] Note that restricting some of these characters in comments may make it more difficult to use natural text in right-to-left scripts, such as described in how-to-comment-in-a-right-to-left-language-in-visual-studio-ide. For example, users may need to reword and split comments to replace the use of RLE..PDF ranges.

Compilers

This level includes compilers, linters, and other programming language tooling that has a deep understanding of particular source code.

The advantages of changes on this level is that the help they provide is not limited to particular IDEs or editors. They can require escaping (as with High-Level Linters), but then can also apply some additional tests since they are aware of the particular syntax of the programming language in question.

In particular, they can apply tests in the following areas:

Bidirectional order

1. **Tokens out of order**

memory:	s		=		א	ב	ג	-	1	0	0	;
index:	0	1	2	3	4	5	6	7	8	9	10	11
token#:	0	1	2	3	4			5	6			7
visual:	s		=		1	0	0	-	ג	ב	א	;
index:	0	1	2	3	8	9	10	7	6	5	4	11
token#:	0	1	2	3	6			5	4			7

- In memory, the Hebrew identifier (token #4) is before 100 (token #6), but visually it appears *after*. That is, tokens 4 and 6 are swapped.
- For identifiers, this only affects programming languages that allow non-ASCII identifiers. However, the same effect can be produced with strings:

memory:	s		=		"	א	ב	ג	"		+		"	1	0	0	"	;
index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
token#:	0	1	2	3	4					5	6	7	8					9
visual:	s		=		"	1	0	0	"		+		"	ג	ב	א	"	;
index:	0	1	2	3	4	13	14	15	12	11	10	9	8	7	6	5	16	17
token#:	0	1	2	3	4	8				7	6	5	4				8	9

- In this case, tokens #4-#8 are reversed ***and*** tokens 4 and 8 are split

It is recommended that programming languages allow for insertion of LRM (*left-right-mark*) characters in whitespace (as some already do), see UAX31-R3 "Pattern_White_Space and Pattern_Syntax Characters". That provides a mechanism for users to correct the visual display of RTL text, and thus avoid raising errors in these cases (and look correct even when pasted into email, etc.). A sufficiently smart editor could handle this automatically.

2. **Token with ambiguous memory order**: two tokens appear identical visually, but are ordered differently in memory.
    - Line 1 — no reordering happens

memory:	s		=		a	1	א	;
index:	0	1	2	3	4	5	6	7
token#:	0	1	2	3	4	4	4	5
visual:	s		=		a	1	א	;
index:	0	1	2	3	4	5	6	7
token#:	0	1	2	3	4	4	4	5

    - Line 2 — the last two characters of an identifier are swapped in visual order

memory:	s		=		a	א	1	;
index:	0	1	2	3	4	5	6	7
token#:	0	1	2	3	4	4	4	5
visual:	s		=		a	1	א	;
index:	0	1	2	3	4	6	5	7
token#:	0	1	2	3	4	4	4	5

    - Note that what appears to be the same token visually (an identifier a1א on both visual lines) has two different memory representations (a1א and aא1)
3. **Token with unexpected boundaries:** A token encompasses more or fewer characters than expected.

memory:	p	r	i	n	t	f	(	R	"	(	א	(| ב |)	"	?	p	a	s	s	w	o	r	d	:	"	)	"	)	;
index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
token#:	0	0	0	0	0	0	1	2	3							4	5								6	7			8	9
visual:	p	r	i	n	t	f	(	R	"	(	)	ב	(	א	"	?	p	a	s	s	w	o	r	d	:	"	)	"	)	;
index:	0	1	2	3	4	5	6	7	8	9	13	12	11	10	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
token#:	0	0	0	0	0	0	1	2	3							4	5								6	7			8	9

    - This uses the 'raw string' construct in C++, R"(...)", but in the visual row it appears that the raw string goes out to position 27, printing a long literal. In fact, the raw string only goes out to position 14, so what it actually prints out is contents of the variable `password`: see https://gcc.godbolt.org/z/Y8dPhP49Y

Code to detect tokens out of order is relatively straightforward to implement using an API that handles the Unicode Bidirectional Algorithm, such as the Bidi class in ICU. Code to detect a token

with ambiguous memory order or with unexpected boundaries is trickier. In all cases we should supply utilities in ICU so that developers don't need to learn the intricacies of bidirectional ordering. Those utilities can also serve as a pattern for constructing similar utilities in other programming languages.

Confusables

We already have descriptions and ICU code and data to support confusable detection, mixed scripts, and so on. We need to have a description of the application to source code, with examples. At a compiler level there is enough context that the compiler knows the identifiers that are in scope at any particular location, and can test whether a new identifier declaration is confusable with one of the previous ones.

Source Code Editors

Spoofing problems can also be addressed at the editing stage. This also applies to viewing applications, such as with code review diffs. But for brevity we'll subsume those under the term ' code editor'.

Bidirectional order

There are a few techniques that can be applied. Most of these involve knowledge of the particular programming language that is being edited or used.

Syntax highlighting can help to make clear when the visual appearance is misleading. For example, in the following example, part of a comment is reordered to appear as if it were code, and vice-versa.

```
return "1000";  // ;"A thousand."
```

However, it cannot be the only solution, because there are reordering examples where two strings are simply swapped, and plain syntax highlighting would not help for those. Syntax highlighting *could* expose the reordering issues discussed under Compilers, but can also be confusing (and color is less effective for the color-blind).

```
S : constant String := 7 * "0" & "1";  -- Actually 11111110!
```

Visual escaping is where a character is displayed as a "chit" instead of its normal appearance. That chit can be a hex representation (eg 202A) or a mnemonic (eg LRE). In either case, it is helpful to have a mouse hover or other mechanism that can have a longer description (eg "left-to-right embedding: changes the text order around it"). The text should be ordered as if the chit were a single neutral character, thus preventing reordering via stateful controls. Visual escaping can help to expose 'hidden' characters like the stateful bidi controls, and reduce the opportunity for reordering. It does require extra work in the code editor, but many code editors provide a 'show hidden' mode which can be extended.

The above example displays as follows on godbolt.org:

```
S : constant String := 7 * [U+200F]"1" & "0"[U+200F];
```

However, visual escaping of individual characters is not sufficient. In the Python example below, there are reordering problems despite there being no 'hidden' characters.

```
three_alefs_and_a_tav = 3 * 'ת' + 'א'  # This is אאאת, not תתתא!
```

Such visual escaping may even be counterproductive if the invisible characters are being used to *restore* desired display order in the presence of RTL text, as is possible in languages conforming to UAX31-R3 with no profile.

For instance, the broken rendering (1) below may be fixed in Rust by inserting a left-to-right mark after every identifier, and a right-to-left mark at the end of the comment (2):

```
(1) return תרבוע גאוסיאני // ;(1.0 .. 1.0- ,אינטגרל(פונקציה.
(2) return אינטגרל(פונקציה, -1.0 .. 1.0); // .תרבוע גאוסיאני
```

However, visual escaping makes the result hard to read—and incorrect in the case of the comment—:

```
return אינטגרל[U+200E](פונקציה[U+200E], -1.0 .. 1.0); // תרבוע גאוסיאני.[U+200F]
```

The use of directionally neutral "chits", e.g., a middle dot, even undoes the fix entirely, as in this screenshot from the Rust playground (using the ace editor):

```
return תרבוע גאוסיאני // ;(1.0 .. 1.0- ,·(פונקציה)·אינטגרל·.·
```

A fuller solution involves lexing the line for tokens, and presenting those tokens in a monotonic order. Typically that is left-to-right, but specialized or more sophisticated editors would also provide for a right-to-left mode. Note that something that does enough syntax highlighting most likely already does the bulk of the work of this approach. And if it is rendering it in HTML, it just needs a bunch of dir=ltr spans to achieve the desired behaviour.

The Python example above displays as follows in Microsoft Visual Studio, which implements such a solution:

```
three_alefs_and_a_tav·=·3·*·'א'·+·'ת'··#·This·is·אאאת,·not·תתתא!
```

That reduces the attack surface dramatically, but not completely, because there can still be tokens that are visual spoofs of others.

Compare the renderings of the following identifiers:

— `UAX9_סעיף_HL4` (memory order `UAX9_section_HL4`);
— `UAX9_סעיף_HL4` (memory order `UAXsection_9_HL4`).

Confusables (homoglyphs)

An editor that performs tokenization, *e.g.*, for syntax highlighting, could detect confusable identifiers and flag them.

For instance, it could warn on the coexistence of the following identifiers within a given scope, source file, or library:

1. HTTP_Сервер (HTTP_*Server*);
2. HTTP_Сервер (all-Cyrillic *NTTR_Server*).

Systematically flagging mixed scripts in identifiers may be a problem, since English technical terms and Latin acronyms abound in source code (see HTTP above). Whether it is beneficial to require a boundary (such as "_" or a change of case) between scripts, or whether confusable detection (and possibly flagging of characters with Identifier_Status=Restricted) suffices is yet unclear.

As mentioned in a footnote above, flagging confusable identifiers in the broad sense currently used by UTS #39 may also be undesired in some cases; for instance, it would result in warnings about the following snippets:

— `v_k = v_l + v_1;`
— `Pressure p; Density ρ;`

Confusable detection made without awareness of the lexical structure of program text is very likely to be unacceptable for users. See, *e.g.*, twitter.com/marinintim/status/1482095046635704322, wherein *individual* Cyrillic letters confusable with Latin are being highlighted in string literals containing Russian text.

ICU provides utilities which could assist in implementing some of these diagnostics, with some adaptations for source code.

Properties

There are a few cases where we might consider some changes to properties. *This is not a proposal for specific additions of properties or property values, but rather ideas that the working group could discuss.*

Stateful Format Characters

Stateful controls or format characters have an extended visual effect between them (or until the end of a paragraph.

There is currently no property that indicates which characters are stateful, which is important information. We could consider an additional binary property for that, such as the following:

U+202A	LEFT-TO-RIGHT EMBEDDING
U+202B	RIGHT-TO-LEFT EMBEDDING
U+202C	POP DIRECTIONAL FORMATTING
U+202D	LEFT-TO-RIGHT OVERRIDE
U+202E	RIGHT-TO-LEFT OVERRIDE
U+2066	LEFT-TO-RIGHT ISOLATE
U+2067	RIGHT-TO-LEFT ISOLATE
U+2068	FIRST STRONG ISOLATE
U+2069	POP DIRECTIONAL ISOLATE
U+206A	⛔ INHIBIT SYMMETRIC SWAPPING
U+206B	⛔ ACTIVATE SYMMETRIC SWAPPING
U+206C	⛔ INHIBIT ARABIC FORM SHAPING
U+206D	⛔ ACTIVATE ARABIC FORM SHAPING
U+206E	⛔ NATIONAL DIGIT SHAPES
U+206F	⛔ NOMINAL DIGIT SHAPES
U+FFF9	INTERLINEAR ANNOTATION ANCHOR
U+FFFB	INTERLINEAR ANNOTATION TERMINATOR
U+13437	EGYPTIAN HIEROGLYPH BEGIN SEGMENT
U+13438	EGYPTIAN HIEROGLYPH END SEGMENT
U+1D173	MUSICAL SYMBOL BEGIN BEAM
U+1D174	MUSICAL SYMBOL END BEAM
U+1D175	MUSICAL SYMBOL BEGIN TIE
U+1D176	MUSICAL SYMBOL END TIE
U+1D177	MUSICAL SYMBOL BEGIN SLUR

U+1D178	MUSICAL SYMBOL END SLUR
U+1D179	MUSICAL SYMBOL BEGIN PHRASE
U+1D17A	MUSICAL SYMBOL END PHRASE

[\u202A-\u202E\u2066-\u206F\uFFF9\uFFFB\x{13437}\x{13438}\x{1D173}-\x{1D179}]

The characters marked with ⛔ are deprecated, and thus should not in any event be used. Note that the following is not in the list, since it is a separator.

U+FFFA	INTERLINEAR ANNOTATION SEPARATOR

The above is a very preliminary list. The following need to be checked.

The above list

- [\u202A-\u202E\u2066-\u206F\uFFF9\uFFFB\x{13437}\x{13438}\x{1D173}-\x{1D179}]

Other Cf characters

- [\p{Cf}-[\u202A-\u202E\u2066-\u206F\uFFF9\uFFFB\x{13437}\x{13438}\x{1D173}-\x{1D179}]]

Some non-Cf characters

- Some non-Cf characters may also be stateful, and need investigation. For example, enclosed characters are indicated by https://www.unicode.org/versions/Unicode14.0.0/ch11.pdf (search for *Enclosures*), but unfortunately the full list of such enclosures is not made explicit in that document. The following are possibilities for investigation but there may be others:

13258 EGYPTIAN HIEROGLYPH O006A
• beginning of hwt or serekh enclosure
→ 13282 egyptian hieroglyph o033a

13259 EGYPTIAN HIEROGLYPH O006B
• beginning of low hwt enclosure

1325A EGYPTIAN HIEROGLYPH O006C
• beginning of high hwt enclosure

1325B EGYPTIAN HIEROGLYPH O006D
• end of high hwt enclosure

1325C EGYPTIAN HIEROGLYPH O006E
• end of low hwt enclosure

1325D EGYPTIAN HIEROGLYPH O006F
• end of hwt enclosure

13286 EGYPTIAN HIEROGLYPH O036A
• beginning of fortified wall cartouche

13287 EGYPTIAN HIEROGLYPH O036B
• end of fortified wall cartouche

13288 EGYPTIAN HIEROGLYPH O036C
• beginning of fortified wall enclosure

13289 EGYPTIAN HIEROGLYPH O036D
• end of fortified wall enclosure

13379 EGYPTIAN HIEROGLYPH V011A
• beginning of cartouche

1337A EGYPTIAN HIEROGLYPH V011B
• end of cartouche

13282 EGYPTIAN HIEROGLYPH O033A • end of serekh enclosure → 13258 egyptian hieroglyph o006a	1337B EGYPTIAN HIEROGLYPH V011C • end of knotless cartouche

White Space

The following characters are categorized as gc=Lo, but have no appearance and are typically displayed as whitespace or invisible. Here an appearance is shown in «...», but the precise appearance depends on the font and rendering system: when not part of a Hangul syllable they could be invisible, look like a space, or have a special appearance.

« »	U+115F	HANGUL CHOSEONG FILLER
« »	U+1160	HANGUL JUNGSEONG FILLER
« »	U+3164	HANGUL FILLER
« »	U+FFA0	HALFWIDTH HANGUL FILLER

Typically characters like these are either Cf or Z, so people overlook them when considering security issues. There may be others of this type as well.

We should consider how we can categorize these characters in a way that highlights their nature. Options:

1. Have a special property for them?
2. Re categorize them as Cf? (And make sure they work properly in word-break, etc.)
3. Other options?

Requires Bidi

An important optimization is to quickly check for the following set of characters; if none are found then implementations don't have to invoke any special machinery for bidirectional handling.

[\p{bc=AL}\p{bc=AN}\p{bc=LRE}\p{bc=RLE}\p{bc=LRO}\p{bc=RLO}\p{bc=PDF}\p{bc=FSI}\p{bc=RLI}\p{bc=LRI}\p{bc=PDI}\p{bc=R}]

However, this expression is easy for people to get wrong, so we might consider whether there is some way to make that easier and less error-prone.

Option 1: Document this (and its inverse) explicitly in UAX #9 and also in #39.

Option 2: Provide a new property RequiresBidi that has the contents above.

Principles

As part of this work, we'll be developing principles to help guide the work. The following is an incomplete, rough draft of what they might look like.

Notation

In examples throughout this section, transliterations to Latin script and translations to English are used to clarify the memory order of bidirectional program text, and to disambiguate confusables. *Italics* are used for transliterated and translated words.

Glossary

- token = lexical element; but is treated broadly. For example, it includes "whitespace" tokens.
- in-memory order = logical order

Lines are treated independently, as separate paragraphs with respect to the BIDI algorithm. For this purpose, a lexical element that spans multiple lines is treated as multiple tokens.

List of principles

P0. The desired rendering of program text never splits a token visually.

> Example: The following rendering of a line of Rust is undesired—the string literal is not visually contiguous:
>
> println!("{} 1729 , "מוניות"); println!("{} *taxis*", 1729);
>
> Neither of the following renderings violates P0; the first has LTR token order, the second RTL.
>
> println!("{} מוניות", 1729);
>
> ;(1729 ,"מוניות {}")!println

P1. The desired rendering of program text displays tokens in a visual order that matches the in-memory order.

> Example: The following rendering of a line of C++ is undesired; the underlined tokens are displayed right-to-left in an otherwise left-to-right line.
>
> std::vector<מיאו> חתול; std::vector<*meow*> *cat*;
>
> Neither of the following renderings[5] violates P1.
>
> std::vector<מיאו> חתול;
>
> ;חתול <מיאו>vector::std

P2. For program text having tokens with initial and terminal markers (such as string literals, end-of-line comments, and block comments), the desired rendering of those markers is that they appear at the start and end of the token (in the in-memory order).

[5] Neither of these renderings is achievable in plain text in C++, since they both require the use of implicit directional marks, and C++ does not conform with UAX #31 R3 with no profile.

Example: The following left-to-right rendering[6] of a C++ raw string literal is undesired; while the initial marker R"(appears correctly to the left, the terminal marker)" does not appear to the right of the token:

R"(בא)". R"(*a*(*b*)".

The following rendering[7] does not violate P2:

R"(בא)".

[6] This is the rendering under UAX #9 v. 6.3 or later.

[7] Not achievable in plain text: a left-to-right mark had to be inserted inside the string.

Reports

Some of the reports of problems include:

- https://www.trojansource.codes/
- https://www.kb.cert.org/vuls/id/999008
- https://www.rapid7.com/blog/post/2021/11/04/trojan-source-cve-2021-42572/
- https://www.python.org/dev/peps/pep-0672/
- And others

More examples

Robin has done some in-depth analysis of examples and behavior in different programming languages, and we could add more.

Usability

The use of strongly right-to-left text in identifiers or string literals can easily make source code illegible.

C# 1.0/1.2 (ISO/IEC 23270:2003) or later:

```
Console.WriteLine("הודעה ,"({1}) {0} :השתבש, this);
```

parses—and is typed—as

```
Console.WriteLine("Error: {0} ({1})", message, this);
```

Ada 2005 (ISO/IEC 8652:1995 with Technical Corrigendum 1 (COR 1:2001) and Amendment 1 (AMD 1:2007)) or later:

```
<<השתבש משהו -- ;(הודעה) רשם <<שגיאה.
```

parses—and is typed—as

```
<<Error>> Log (Message); -- Something went wrong.
```

Python 3.0 (2008) or later:

```
return אינטגרל(lambda 1=ל ,0=מ ,2 ** א :א)
```

parses—and is typed—as

```
return integral(lambda a: a ** 2, from_=0, to=1)
```

C++11 (ISO/IEC 14882:2011) or later:

```
std::vector<قطة <مواء;
```

parses—and is typed—as

```
std::vector<meow> cat;
```

In the same language,

```
return u8"رسالة العنصر النائب //  ;"مواء.
```

parses—and is typed—as

```
return u8"meow";  // Placeholder message.
```

Rust 1.53.0 (2021) or later:

```
fn אינטגרל<פונקציה: Fn(f64) -> f64>(
    קטע ,פונקציה :אינטגרנד: std::ops::Range<f64>) -> f64 {
```

parses—and is typed—as

```
fn integral<Function: Fn(f64) -> f64>(
    integrand: Function, interval: std::ops::Range<f64>) -> f64 {
```

Security

The examples in this section don't simply seem confusing or illegible as the ones above do; instead the reader would be expected to think that they are programs with different semantics. While we provide examples involving literals or identifiers in strongly right-to-left scripts, we also provide some that have neither (and instead use the invisible implicit directional marks), as these may not look suspicious even in a codebase that does not use right-to-left scripts.

C++98 or later:

```
std::cerr << "encountered " << (errors == 0 ? " 0 " : "")
          << "errors";
```

Will print "encountered errors" if and only if errors = 0, and "encountered 0 errors" otherwise.

Ada 2005 or later:

```
for Hebrew_Letter in Wide_Character range 'א' .. 'ת' loop
```

While it may seem like it loops over the Hebrew alphabet (from alef to tav), this is actually dead code (looping over the empty range from tav to alef).

Rust 1.9 (2016) or later:

```
return x >> 8;
```

While this looks like a right shift by eight bits, it is a left shift by eight bits.