Draft amendment to ISO/IEC 24772-1 to add vulnerability on source code representation

Formatted: Normal

6.66 Code representation differs between compiler view and reader view

6.66.1 Description of application vulnerability

The ISO/IEC 10646:2020 character set includes characters that can cause a reordering of the displayed source code, such that the semantics of the displayed code differs from the semantics of the executed code. Such characters set text display direction left-to-right or right-to-left but are invisible unless the editor or display program is instructed to mnemonically display them. If left-to-right is the current default direction and a right-to-left character (`RLI`) is used, subsequent text can visually expand the text preceding the `RLI` character.

Deleted: actually

Deleted: will

The following example, taken from ISO/IEC 24772-4[1], shows code with the invisible characters denoted visibly by `+LRI, +PDI, +RLO`, where these denotations stand for the zero-space Unicode control characters:

```
<LRI> Left-to-Right Isolate
<PDI> Pop Directional Isolate
<RLO> Right-to-Left Overwrite
```

Deleted:

Deleted:

Due to the direction-changing characters, the following code

```
alvl = 'user'
if alvl != 'none+RLO+LRI': #Check if admin+PDI+LRI' and alvl!= 'user’
    print('You are an admin.')
```

will be displayed to the human reader in some editors as:

```
alvl = 'user'
if alvl != 'none’ and alvl!= 'user' #Check if admin
    print('You are an admin.')
```

but execute as:

```
alvl = 'user'
if alvl != 'none’:  #Check if admin' and alvl!= 'user’
```

```
print('You are an admin.')
```

as the second condition shown in the visual representation is really part of the comment in the actual code.

Some languages restrict the use of direction-changing control characters to comments or strings. Nevertheless, malicious use can cause a string or comment to appear to change executable code, as shown above and also below using RLI in a string.

Deleted: change

Deleted: into

```
'''Subtract funds from account then  RLI      ''' ; return
'''LRI'''
```

This line can display as, depending on the text editor used;

```
'''Subtract funds from bank account then return;'''
```

but executes as

```
; return
```

A similar situation arises from the use of the carriage return <CR> and line feed <LF> characters, depending upon the environment where the code is executed.

Example

```
Blow_Up(); <CR> BeReallyNice()
```

The lack of a <LF> can cause the code (e.g in UNIX-based systems) to be displayed as

```
BeReallyNice()
```

while the code executes as

```
Blow_Up(); BeReallyNice()
```

because some environments will overwrite the displayed line if the <LF> is not included with the <CR>.

6.66.2 Related coding guidelines

6.66.3 Mechanism of failure

The examples in 6.66.1 show how readers of code can be misled about the actual code that will be executed once the code is compiled or interpreted. Seemingly executed code can be

effectively added or subtracted in the visual display of the program. Thus these Unicode characters are a simple means to cause the execution of malicious code.

Additionally, the end-of-line issue can be a source of unintentional errors and a difficult search for the origin of unexpected program behaviour, when executed code is accidentally not shown in the displayed source code.

6.66.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that permit non-printing Unicode control characters causing differences between displayed code and executed code as part of program code, string literals or comments.
- Languages that permit arbitrary sequences of <CR> and <LF> characters.

6.66.5 Avoiding the vulnerability or mitigating its effect

To avoid the vulnerability or mitigate its ill effects, software developers can:

- Carefully manage and thoroughly review the use of any characters that can in any way hide the functionality and representation of program code.
- Avoid reliance on simple visual inspection of code; instead use tools to reveal dangerous control characters.
- Always use static analysis tools that identify all occurrences of non-visible and hidden characters within a program.
- Use tools to confirm that program code conforms to the end-of-line convention of the environment in which code is edited and compiled.
- Use only editors that are capable of revealing the hidden Unicode (zero-space) control characters and ensure that the appropriate editor setting is enabled.
- Avoid copying code from untrusted sources unless the code is thoroughly checked as described above.

Deleted: Refrain

Deleted: from

Deleted: and pasting

6.66.6 Implications for language design and evolution

In future language design and evolution activities, language designers should consider the following items:

- Flagging all occurrences of Unicode control characters that are capable of causing displayed code to be different from executed code.
- Excluding <CR> and <LF> characters from strings and comments.
- Diagnosing mismatches of program code with end-of-line conventions of the compilation environment.