INTERNATIONAL STANDARD

ISO IEC 24772-1
First Edition
2024

Amendment 1

Programming languages — Avoiding vulnerabilities in Programming languages — Part 1: Language independent catalogue of vulnerabilities

AMENDMENT 1: Code representation differs between compiler view and reader view

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International Organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

ISO draws attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO takes no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO had not received notice of (a) patent(s) which may be required to implement this document. However,

implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents. ISO shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see www.iso.org/iso/foreword.html.

This document was prepared by ISO/IEC/JTC 1 *Information Technology, Subcommittee SC 22,* Programming Languages

A list of all parts of ISO IEC 24772 can be found on the ISO website.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Programming Language — Programming language vulnerabilities — Part 1: Language independent catalogue of vulnerabilities

Clause 6.28 Non-demarcation of control flow [EOJ]

Add 6.28.1 after the last paragraph.

Some languages provide syntax to name a flow control construct as well as syntax to transfer control immediately after the named construct. Usage of such syntax can eliminate programming mistakes associated with control flow such as adding an additional loop and not preserving premature or normal termination conditions or loop exit conditions.

Clause 6.28 Non-demarcation of control flow [EOJ]

Add to 6.28.5

For languages that provide facilities to name control structures, utilize names for complicated nesting structures where early exit from one or more levels is necessary for the algorithm.

Justification for both changes:

The third bullet of 6.28.6

> "providing syntax to terminate named loops and conditionals and to determine if the structure as named matches the structure as inferred."

requires justification or removal. Removal would be counterproductive as the termination of the wrong loop(s) in a nested loop construct is a vulnerability.

Add the following new subclause to clause 6:

6.66 Code representation differs between compiler view and reader view [FPV]

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 differ 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 will visually expand the text preceding the `RLI` character and overwrite text that had already been written.

The following example, taken from [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
```

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 make part of a string or comment appear to be part executable code, or vice versa as shown above and also below using RLI in a string.

```
'''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-printing 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.

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.

Justification:

This vulnerability was recognized after the document was being published and was determined by the committee to be worthy of addition to the existing standard.

[1] Anderson, R. & Boucher, N. Trojan Source: Invisible Vulnerabilities, https://trojansource.codes/trojan-source.pdf