PD ISO/IEC TR 24772-1:2019
Incorporating corrigendum April 2020

BSI Standards Publication

Programming languages — Guidance to avoiding vulnerabilities in programming languages

Part 1: Language-independent guidance

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TR 24772-1:2019

National foreword

This Published Document is the UK implementation of ISO/IEC TR 24772-1:2019. Together with PD ISO/IEC TR 24772-2:2020, they supersede PD ISO/IEC TR 24772:2013, which is withdrawn

The UK participation in its preparation was entrusted to Technical Committee IST/5, Programming languages, their environments and system software interfaces.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© The British Standards Institution 2020
Published by BSI Standards Limited 2020

ISBN 978 0 539 13541 1

ICS 35.060

Compliance with a British Standard cannot confer immunity from legal obligations.

This Published Document was published under the authority of the Standards Policy and Strategy Committee on 31 December 2019.

Amendments/corrigenda issued since publication

Date	Text affected
30 April 2020	Correction to supersession details in National Foreword

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TR 24772-1:2019

TECHNICAL REPORT

ISO/IEC TR 24772-1

First edition
2019-12-10

Programming languages — Guidance to avoiding vulnerabilities in programming languages —

Part 1: Language-independent guidance

Langages de programmation — Conduite pour éviter les vulnérabilités dans les langages de programmation —

Partie 1: Conduite indépendante du langage

Reference number
ISO/IEC TR 24772-1:2019(E)

Documents
Document Sharing Hub

© ISO/IEC 2019

Full document details: PD ISO/IEC TR 24772-1:2019

COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2019, Published in Switzerland

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TR 24772-1:2019

Contents

Page

Foreword ... **xv**

Introduction ... **xvi**

1 Scope ... **1**

2 Normative references ... **1**

3 Terms and definitions ... **1**
3.1 Terms related to communication ... 1
3.2 Terms related to execution model ... 2
3.3 Properties ... 4
3.4 Safety ... 4
3.5 Vulnerabilities ... 4

4 Applying this document ... **5**

5 Vulnerability issues and general avoidance mechanisms ... **7**
5.1 Predictable execution ... 7
5.2 Sources of unpredictability in language specification ... 8
5.2.1 Incomplete or evolving specification ... 8
5.2.2 Undefined behaviour ... 8
5.2.3 Unspecified behaviour ... 8
5.2.4 Implementation-defined behaviour ... 8
5.2.5 Difficult features ... 8
5.2.6 Inadequate language support ... 8
5.3 Sources of unpredictability in language usage ... 9
5.3.1 Porting and interoperation ... 9
5.3.2 Compiler selection and usage ... 9
5.4 Top avoidance mechanisms ... 9

6 Programming language vulnerabilities ... **11**
6.1 General ... 11
6.2 Type system [IHN] ... 11
6.2.1 Description of application vulnerability ... 11
6.2.2 Cross reference ... 11
6.2.3 Mechanism of failure ... 11
6.2.4 Applicable language characteristics ... 13
6.2.5 Avoiding the vulnerability or mitigating its effects ... 13
6.2.6 Implications for language design and evolution ... 13
6.3 Bit representations [STR] ... 14
6.3.1 Description of application vulnerability ... 14
6.3.2 Cross reference ... 14
6.3.3 Mechanism of failure ... 14
6.3.4 Applicable language characteristics ... 14
6.3.5 Avoiding the vulnerability or mitigating its effects ... 15
6.3.6 Implications for language design and evolution ... 15
6.4 Floating-point arithmetic [PLF] ... 15
6.4.1 Description of application vulnerability ... 15
6.4.2 Cross reference ... 15
6.4.3 Mechanism of failure ... 16
6.4.4 Applicable language characteristics ... 16
6.4.5 Avoiding the vulnerability or mitigating its effects ... 17
6.4.6 Implications for language design and evolution ... 17
6.5 Enumerator issues [CCB] ... 18
6.5.1 Description of application vulnerability ... 18
6.5.2 Cross reference ... 18
6.5.3 Mechanism of failure ... 18
6.5.4 Applicable language characteristics ... 19

AVA Documents
Document Sharing Hub

© ISO/IEC 2019 – All rights reserved

Full document details: PD ISO/IEC TR 24772-1:2019

6.5.5 Avoiding the vulnerability or mitigating its effects ... 19
6.5.6 Implications for language design and evolution ... 19
6.6 Conversion errors [FLC] ... 19
6.6.1 Description of application vulnerability ... 19
6.6.2 Cross reference ... 20
6.6.3 Mechanism of failure ... 20
6.6.4 Applicable language characteristics ... 20
6.6.5 Avoiding the vulnerability or mitigating its effects ... 21
6.6.6 Implications for language design and evolution ... 21
6.7 String termination [CJM] ... 21
6.7.1 Description of application vulnerability ... 21
6.7.2 Cross reference ... 21
6.7.3 Mechanism of failure ... 22
6.7.4 Applicable language characteristics ... 22
6.7.5 Avoiding the vulnerability or mitigating its effects ... 22
6.7.6 Implications for language design and evolution ... 22
6.8 Buffer boundary violation (buffer overflow) [HCB] ... 22
6.8.1 Description of application vulnerability ... 22
6.8.2 Cross reference ... 22
6.8.3 Mechanism of failure ... 23
6.8.4 Applicable language characteristics ... 23
6.8.5 Avoiding the vulnerability or mitigating its effects ... 24
6.8.6 Implications for language design and evolution ... 24
6.9 Unchecked array indexing [XYZ] ... 24
6.9.1 Description of application vulnerability ... 24
6.9.2 Cross reference ... 25
6.9.3 Mechanism of failure ... 25
6.9.4 Applicable language characteristics ... 25
6.9.5 Avoiding the vulnerability or mitigating its effects ... 26
6.9.6 Implications for language designers ... 26
6.10 Unchecked array copying [XYW] ... 26
6.10.1 Description of application vulnerability ... 26
6.10.2 Cross reference ... 26
6.10.3 Mechanism of failure ... 26
6.10.4 Applicable language characteristics ... 27
6.10.5 Avoiding the vulnerability or mitigating its effects ... 27
6.10.6 Implications for language design and evolution ... 27
6.11 Pointer type conversions [HFC] ... 27
6.11.1 Description of application vulnerability ... 27
6.11.2 Cross reference ... 28
6.11.3 Mechanism of failure ... 28
6.11.4 Applicable language characteristics ... 28
6.11.5 Avoiding the vulnerability or mitigating its effects ... 28
6.11.6 Implications for language design and evolution ... 28
6.12 Pointer arithmetic [RVG] ... 29
6.12.1 Description of application vulnerability ... 29
6.12.2 Cross reference ... 29
6.12.3 Mechanism of failure ... 29
6.12.4 Applicable language characteristics ... 29
6.12.5 Avoiding the vulnerability or mitigating its effects ... 29
6.12.6 Implications for language design and evolution ... 29
6.13 Null pointer dereference [XYH] ... 29
6.13.1 Description of application vulnerability ... 29
6.13.2 Cross reference ... 29
6.13.3 Mechanism of failure ... 30
6.13.4 Applicable language characteristics ... 30
6.13.5 Avoiding the vulnerability or mitigating its effects ... 30
6.13.6 Implications for language design and evolution ... 30

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

6.14 Dangling reference to heap [XYK] 30
6.14.1 Description of application vulnerability 30
6.14.2 Cross reference 31
6.14.3 Mechanism of failure 31
6.14.4 Applicable language characteristics 31
6.14.5 Avoiding the vulnerability or mitigating its effects 32
6.14.6 Implications for language design and evolution 32
6.15 Arithmetic wrap-around error [FIF] 32
6.15.1 Description of application vulnerability 32
6.15.2 Cross reference 32
6.15.3 Mechanism of failure 33
6.15.4 Applicable language characteristics 33
6.15.5 Avoiding the vulnerability or mitigating its effects 33
6.15.6 Implications for language design and evolution 33
6.16 Using shift operations for multiplication and division [PIK] 34
6.16.1 Description of application vulnerability 34
6.16.2 Cross reference 34
6.16.3 Mechanism of failure 34
6.16.4 Applicable language characteristics 34
6.16.5 Avoiding the vulnerability or mitigating its effects 34
6.16.6 Implications for language design and evolution 34
6.17 Choice of clear names [NAI] 35
6.17.1 Description of application vulnerability 35
6.17.2 Cross reference 35
6.17.3 Mechanism of Failure 35
6.17.4 Applicable language characteristics 36
6.17.5 Avoiding the vulnerability or mitigating its effects 36
6.17.6 Implications for language design and evolution 36
6.18 Dead store [WXQ] 36
6.18.1 Description of application vulnerability 36
6.18.2 Cross reference 36
6.18.3 Mechanism of failure 37
6.18.4 Applicable language characteristics 37
6.18.5 Avoiding the vulnerability or mitigating its effects 37
6.18.6 Implications for language design and evolution 37
6.19 Unused variable [YZS] 38
6.19.1 Description of application vulnerability 38
6.19.2 Cross reference 38
6.19.3 Mechanism of failure 38
6.19.4 Applicable language characteristics 38
6.19.5 Avoiding the vulnerability or mitigating its effects 38
6.19.6 Implications for language design and evolution 38
6.20 Identifier name reuse [YOW] 38
6.20.1 Description of application vulnerability 38
6.20.2 Cross reference 39
6.20.3 Mechanism of failure 39
6.20.4 Applicable language characteristics 40
6.20.5 Avoiding the vulnerability or mitigating its effects 40
6.20.6 Implications for language design and evolution 40
6.21 Namespace issues [BJL] 40
6.21.1 Description of application vulnerability 40
6.21.2 Cross-references 40
6.21.3 Mechanism of failure 41
6.21.4 Applicable language characteristics 41
6.21.5 Avoiding the vulnerability or mitigating its effects 41
6.21.6 Implications for language design and evolution 42
6.22 Initialization of variables [LAV] 42
6.22.1 Description of application vulnerability 42

AVA Documents
Document Sharing Hub

© ISO/IEC 2019 – All rights reserved

Full document details: PD ISO/IEC TR 24772-1:2019

6.22.2 Cross reference ... 42
6.22.3 Mechanism of failure ... 42
6.22.4 Applicable language characteristics ... 43
6.22.5 Avoiding the vulnerability or mitigating its effects ... 43
6.22.6 Implications for language design and evolution ... 44
6.23 Operator precedence and associativity [JCW] ... 44
6.23.1 Description of application vulnerability ... 44
6.23.2 Cross reference ... 44
6.23.3 Mechanism of failure ... 44
6.23.4 Applicable language characteristics ... 45
6.23.5 Avoiding the vulnerability or mitigating its effects ... 45
6.23.6 Implications for language design and evolution ... 45
6.24 Side-effects and order of evaluation of operands [SAM] ... 45
6.24.1 Description of application vulnerability ... 45
6.24.2 Cross reference ... 45
6.24.3 Mechanism of failure ... 46
6.24.4 Applicable language characteristics ... 46
6.24.5 Avoiding the vulnerability or mitigating its effects ... 46
6.24.6 Implications for language design and evolution ... 46
6.25 Likely incorrect expression [KOA] ... 47
6.25.1 Description of application vulnerability ... 47
6.25.2 Cross reference ... 47
6.25.3 Mechanism of failure ... 47
6.25.4 Applicable language characteristics ... 48
6.25.5 Avoiding the vulnerability or mitigating its effects ... 48
6.25.6 Implications for language design and evolution ... 48
6.26 Dead and deactivated code [XYQ] ... 48
6.26.1 Description of application vulnerability ... 48
6.26.2 Cross reference ... 48
6.26.3 Mechanism of failure ... 49
6.26.4 Applicable language characteristics ... 50
6.26.5 Avoiding the vulnerability or mitigating its effects ... 50
6.26.6 Implications for language design and evolution ... 50
6.27 Switch statements and static analysis [CLL] ... 50
6.27.1 Description of application vulnerability ... 50
6.27.2 Cross reference ... 50
6.27.3 Mechanism of failure ... 51
6.27.4 Applicable language characteristics ... 51
6.27.5 Avoiding the vulnerability or mitigating its effects ... 51
6.27.6 Implications for language design and evolution ... 51
6.28 Demarcation of control flow [EOJ] ... 52
6.28.1 Description of application vulnerability ... 52
6.28.2 Cross reference ... 52
6.28.3 Mechanism of failure ... 52
6.28.4 Applicable language characteristics ... 52
6.28.5 Avoiding the vulnerability or mitigating its effects ... 52
6.28.6 Implications for language design and evolution ... 53
6.29 Loop control variables [TEX] ... 53
6.29.1 Description of application vulnerability ... 53
6.29.2 Cross reference ... 53
6.29.3 Mechanism of failure ... 53
6.29.4 Applicable language characteristics ... 53
6.29.5 Avoiding the vulnerability or mitigating its effects ... 53
6.29.6 Implications for language design and evolution ... 54
6.30 Off-by-one error [XZH] ... 54
6.30.1 Description of application vulnerability ... 54
6.30.2 Cross reference ... 54
6.30.3 Mechanism of failure ... 54

Documents
Document Sharing Hub

© ISO/IEC 2019 – All rights reserved

Full document details: PD ISO/IEC TR 24772-1:2019

6.30.4 Applicable language characteristics ... 55
6.30.5 Avoiding the vulnerability or mitigating its effects ... 55
6.30.6 Implications for language design and evolution ... 55
6.31 Unstructured programming [EWD] ... 55
6.31.1 Description of application vulnerability ... 55
6.31.2 Cross reference ... 55
6.31.3 Mechanism of failure ... 56
6.31.4 Applicable language characteristics ... 56
6.31.5 Avoiding the vulnerability or mitigating its effects ... 56
6.31.6 Implications for language design and evolution ... 56
6.32 Passing parameters and return values [CSJ] ... 56
6.32.1 Description of application vulnerability ... 56
6.32.2 Cross reference ... 57
6.32.3 Mechanism of failure ... 57
6.32.4 Applicable language characteristics ... 58
6.32.5 Avoiding the vulnerability or mitigating its effects ... 58
6.32.6 Implications for language design and evolution ... 58
6.33 Dangling references to stack frames [DCM] ... 58
6.33.1 Description of application vulnerability ... 58
6.33.2 Cross reference ... 59
6.33.3 Mechanism of failure ... 59
6.33.4 Applicable language characteristics ... 59
6.33.5 Avoiding the vulnerability or mitigating its effects ... 60
6.33.6 Implications for language design and evolution ... 60
6.34 Subprogram signature mismatch [OTR] ... 60
6.34.1 Description of application vulnerability ... 60
6.34.2 Cross reference ... 60
6.34.3 Mechanism of failure ... 60
6.34.4 Applicable language characteristics ... 61
6.34.5 Avoiding the vulnerability or mitigating its effects ... 61
6.34.6 Implications for language design and evolution ... 61
6.35 Recursion [GDL] ... 61
6.35.1 Description of application vulnerability ... 61
6.35.2 Cross reference ... 62
6.35.3 Mechanism of failure ... 62
6.35.4 Applicable language characteristics ... 62
6.35.5 Avoiding the vulnerability or mitigating its effects ... 62
6.35.6 Implications for language design and evolution ... 62
6.36 Ignored error status and unhandled exceptions [OYB] ... 63
6.36.1 Description of application vulnerability ... 63
6.36.2 Cross reference ... 63
6.36.3 Mechanism of failure ... 63
6.36.4 Applicable language characteristics ... 64
6.36.5 Avoiding the vulnerability or mitigating its effects ... 64
6.36.6 Implications for language design and evolution ... 65
6.37 Type-breaking reinterpretation of data [AMV] ... 65
6.37.1 Description of application vulnerability ... 65
6.37.2 Cross reference ... 65
6.37.3 Mechanism of failure ... 65
6.37.4 Applicable language characteristics ... 66
6.37.5 Avoiding the vulnerability or mitigating its effects ... 66
6.37.6 Implications for language design and evolution ... 66
6.38 Deep vs. shallow copying [YAN] ... 66
6.38.1 Description of application vulnerability ... 66
6.38.2 Cross reference ... 67
6.38.3 Mechanism of failure ... 67
6.38.4 Applicable language characteristics ... 67
6.38.5 Avoiding the vulnerability or mitigating its effects ... 67

Documents
Document Sharing Hub

© ISO/IEC 2019 – All rights reserved

Full document details: PD ISO/IEC TR 24772-1:2019

6.38.6 Implications for language design and evolution ... 67
6.39 Memory leaks and heap fragmentation [XYL] ... 68
6.39.1 Description of application vulnerability ... 68
6.39.2 Cross reference ... 68
6.39.3 Mechanism of failure ... 68
6.39.4 Applicable language characteristics ... 68
6.39.5 Avoiding the vulnerability or mitigating its effects ... 68
6.39.6 Implications for language design and evolution ... 69
6.40 Templates and generics [SYM] ... 69
6.40.1 Description of application vulnerability ... 69
6.40.2 Cross reference ... 69
6.40.3 Mechanism of failure ... 70
6.40.4 Applicable language characteristics ... 70
6.40.5 Avoiding the vulnerability or mitigating its effects ... 70
6.40.6 Implications for language design and evolution ... 71
6.41 Inheritance [RIP] ... 71
6.41.1 Description of application vulnerability ... 71
6.41.2 Cross reference ... 71
6.41.3 Mechanism of failure ... 71
6.41.4 Applicable language characteristics ... 72
6.41.5 Avoiding the vulnerability or mitigating its effects ... 72
6.41.6 Implications for language design and evolution ... 73
6.42 Violations of the Liskov substitution principle or the contract model [BLP] ... 73
6.42.1 Description of application vulnerability ... 73
6.42.2 Cross reference ... 73
6.42.3 Mechanism of failure ... 73
6.42.4 Applicable language characteristics ... 74
6.42.5 Avoiding the vulnerability or mitigating its effects ... 74
6.42.6 Implications for language design and evolution ... 74
6.43 Redispatching [PPH] ... 74
6.43.1 Description of application vulnerability ... 74
6.43.2 Cross reference ... 74
6.43.3 Mechanism of failure ... 74
6.43.4 Applicable language characteristics ... 75
6.43.5 Avoiding the vulnerability or mitigating its effects ... 75
6.43.6 Implications for language design and evolution ... 75
6.44 Polymorphic variables [BKK] ... 75
6.44.1 Description of application vulnerability ... 75
6.44.2 Cross reference ... 76
6.44.3 Mechanism of failure ... 76
6.44.4 Applicable language characteristics ... 76
6.44.5 Avoiding the vulnerability or mitigating its effects ... 76
6.44.6 Implications for language design and evolution ... 77
6.45 Extra intrinsics [LRM] ... 77
6.45.1 Description of application vulnerability ... 77
6.45.2 Cross reference ... 77
6.45.3 Mechanism of failure ... 77
6.45.4 Applicable language characteristics ... 77
6.45.5 Avoiding the vulnerability or mitigating its effects ... 77
6.45.6 Implications for language design and evolution ... 78
6.46 Argument passing to library functions [TRJ] ... 78
6.46.1 Description of application vulnerability ... 78
6.46.2 Cross reference ... 78
6.46.3 Mechanism of failure ... 78
6.46.4 Applicable language characteristics ... 78
6.46.5 Avoiding the vulnerability or mitigating its effects ... 78
6.46.6 Implications for language design and evolution ... 79
6.47 Inter-language calling [DJS] ... 79

Documents
Document Sharing Hub

© ISO/IEC 2019 – All rights reserved

Full document details: PD ISO/IEC TR 24772-1:2019

6.47.1 Description of application vulnerability ... 79
6.47.2 Cross reference ... 79
6.47.3 Mechanism of failure ... 79
6.47.4 Applicable language characteristics ... 80
6.47.5 Avoiding the vulnerability or mitigating its effects ... 80
6.47.6 Implications for language design and evolution ... 80
6.48 Dynamically-linked code and self-modifying code [NYY] ... 80
6.48.1 Description of application vulnerability ... 80
6.48.2 Cross reference ... 81
6.48.3 Mechanism of failure ... 81
6.48.4 Applicable language characteristics ... 81
6.48.5 Avoiding the vulnerability or mitigating its effects ... 81
6.48.6 Implications for language design and evolution ... 81
6.49 Library signature [NSQ] ... 82
6.49.1 Description of application vulnerability ... 82
6.49.2 Cross reference ... 82
6.49.3 Mechanism of failure ... 82
6.49.4 Applicable language characteristics ... 82
6.49.5 Avoiding the vulnerability or mitigating its effects ... 82
6.49.6 Implications for language design and evolution ... 82
6.50 Unanticipated exceptions from library routines [HJW] ... 83
6.50.1 Description of application vulnerability ... 83
6.50.2 Cross reference ... 83
6.50.3 Mechanism of failure ... 83
6.50.4 Applicable language characteristics ... 83
6.50.5 Avoiding the vulnerability or mitigating its effects ... 83
6.50.6 Implications for language design and evolution ... 83
6.51 Pre-processor directives [NMP] ... 84
6.51.1 Description of application vulnerability ... 84
6.51.2 Cross reference ... 84
6.51.3 Mechanism of failure ... 84
6.51.4 Applicable language characteristics ... 85
6.51.5 Avoiding the vulnerability or mitigating its effects ... 85
6.51.6 Implications for language design and evolution ... 85
6.52 Suppression of language-defined run-time checking [MXB] ... 85
6.52.1 Description of application vulnerability ... 85
6.52.2 Cross reference ... 85
6.52.3 Mechanism of Failure ... 85
6.52.4 Applicable language characteristics ... 86
6.52.5 Avoiding the vulnerability ... 86
6.52.6 Implications for language design and evolution ... 86
6.53 Provision of inherently unsafe operations [SKL] ... 86
6.53.1 Description of application vulnerability ... 86
6.53.2 Cross reference ... 87
6.53.3 Mechanism of Failure ... 87
6.53.4 Applicable language characteristics ... 87
6.53.5 Avoiding the vulnerability ... 87
6.53.6 Implications for language design and evolution ... 87
6.54 Obscure language features [BRS] ... 87
6.54.1 Description of application vulnerability ... 87
6.54.2 Cross reference ... 87
6.54.3 Mechanism of failure ... 88
6.54.4 Applicable language characteristics ... 88
6.54.5 Avoiding the vulnerability or mitigating its effects ... 88
6.54.6 Implications for language design and evolution ... 88
6.55 Unspecified behaviour [BQF] ... 89
6.55.1 Description of application vulnerability ... 89
6.55.2 Cross reference ... 89

AVA Documents
Document Sharing Hub

© ISO/IEC 2019 – All rights reserved

Full document details: PD ISO/IEC TR 24772-1:2019

6.55.3 Mechanism of failure ... 89
6.55.4 Applicable language characteristics ... 89
6.55.5 Avoiding the vulnerability or mitigating its effects ... 90
6.55.6 Implications for language design and evolution ... 90
6.56 Undefined behaviour [EWF] ... 90
6.56.1 Description of application vulnerability ... 90
6.56.2 Cross reference ... 90
6.56.3 Mechanism of failure ... 90
6.56.4 Applicable language characteristics ... 91
6.56.5 Avoiding the vulnerability or mitigating its effects ... 91
6.56.6 Implications for language design and evolution ... 91
6.57 Implementation-defined behaviour [FAB] ... 91
6.57.1 Description of application vulnerability ... 91
6.57.2 Cross reference ... 91
6.57.3 Mechanism of failure ... 92
6.57.4 Applicable language characteristics ... 92
6.57.5 Avoiding the vulnerability or mitigating its effects ... 92
6.57.6 Implications for language design and evolution ... 93
6.58 Deprecated language features [MEM] ... 93
6.58.1 Description of application vulnerability ... 93
6.58.2 Cross reference ... 93
6.58.3 Mechanism of failure ... 93
6.58.4 Applicable language characteristics ... 94
6.58.5 Avoiding the vulnerability or mitigating its effects ... 94
6.58.6 Implications for language design and evolution ... 94
6.59 Concurrency — Activation [CGA] ... 94
6.59.1 Description of application vulnerability ... 94
6.59.2 Cross-references ... 94
6.59.3 Mechanism of Failure ... 95
6.59.4 Applicable language characteristics ... 95
6.59.5 Avoiding the vulnerability or mitigating its effects ... 95
6.59.6 Implications for language design and evolution ... 96
6.60 Concurrency — Directed termination [CGT] ... 96
6.60.1 Description of application vulnerability ... 96
6.60.2 Cross-references ... 96
6.60.3 Mechanism of failure ... 96
6.60.4 Applicable language characteristics ... 97
6.60.5 Avoiding the vulnerability or mitigating its effect ... 97
6.60.6 Implications for language design and evolution ... 97
6.61 Concurrent data access [CGX] ... 97
6.61.1 Description of application vulnerability ... 97
6.61.2 Cross-references ... 97
6.61.3 Mechanism of failure ... 98
6.61.4 Applicable language characteristics ... 98
6.61.5 Avoiding the vulnerability or mitigating its effect ... 98
6.61.6 Implications for language design and evolution ... 98
6.62 Concurrency — Premature termination [CGS] ... 99
6.62.1 Description of application vulnerability ... 99
6.62.2 Cross-references ... 99
6.62.3 Mechanism of failure ... 99
6.62.4 Applicable language characteristics ... 100
6.62.5 Avoiding the vulnerability or mitigating its effect ... 100
6.62.6 Implications for language design and evolution ... 100
6.63 Lock protocol errors [CGM] ... 100
6.63.1 Description of application vulnerability ... 100
6.63.2 Cross-references ... 101
6.63.3 Mechanism of failure ... 101
6.63.4 Applicable language characteristics ... 102

Documents
Document Sharing Hub

© ISO/IEC 2019 – All rights reserved

Full document details: PD ISO/IEC TR 24772-1:2019

6.63.5 Avoiding the vulnerability or mitigating its effect ... 102
6.63.6 Implications for language design and evolution ... 102
6.64 Reliance on external format strings [SHL] ... 103
6.64.1 Description of application vulnerability ... 103
6.64.2 Cross reference ... 103
6.64.3 Mechanism of failure ... 103
6.64.4 Applicable language characteristics ... 104
6.64.5 Avoiding the vulnerability or mitigating its effects ... 104
6.64.6 Implications for language design and evolution ... 104

7 Application vulnerabilities ... 104
7.1 General ... 104
7.2 Unrestricted file upload [CBF] ... 104
7.2.1 Description of application vulnerability ... 104
7.2.2 Cross reference ... 104
7.2.3 Mechanism of failure ... 105
7.2.4 Avoiding the vulnerability or mitigating its effects ... 105
7.3 Download of code without integrity check [DLB] ... 105
7.3.1 Description of application vulnerability ... 105
7.3.2 Cross reference ... 106
7.3.3 Mechanism of failure ... 106
7.3.4 Avoiding the vulnerability or mitigating its effects ... 106
7.4 Executing or loading untrusted code [XYS] ... 106
7.4.1 Description of application vulnerability ... 106
7.4.2 Cross reference ... 106
7.4.3 Mechanism of failure ... 106
7.4.4 Avoiding the vulnerability or mitigating its effects ... 107
7.5 Inclusion of functionality from untrusted control sphere [DHU] ... 107
7.5.1 Description of application vulnerability ... 107
7.5.2 Cross reference ... 107
7.5.3 Mechanism of failure ... 107
7.5.4 Avoiding the vulnerability or mitigating its effects ... 107
7.6 Use of unchecked data from an uncontrolled or tainted source [EFS] ... 108
7.6.1 Description of application vulnerability ... 108
7.6.2 Cross reference ... 108
7.6.3 Mechanism of failure ... 108
7.6.4 Avoiding the vulnerability or mitigating its effects ... 108
7.7 Cross-site scripting [XYT] ... 109
7.7.1 Description of application vulnerability ... 109
7.7.2 Cross reference ... 109
7.7.3 Mechanism of failure ... 109
7.7.4 Avoiding the vulnerability or mitigating its effects ... 111
7.8 URL redirection to untrusted site ("open redirect") [PYQ] ... 111
7.8.1 Description of application vulnerability ... 111
7.8.2 Cross reference ... 111
7.8.3 Mechanism of failure ... 111
7.8.4 Avoiding the vulnerability or mitigating its effects ... 111
7.9 Injection [RST] ... 112
7.9.1 Description of application vulnerability ... 112
7.9.2 Cross reference ... 112
7.9.3 Mechanism of failure ... 113
7.9.4 Avoiding the vulnerability or mitigating its effects ... 114
7.10 Unquoted search path or element [XZQ] ... 114
7.10.1 Description of application vulnerability ... 114
7.10.2 Cross reference ... 115
7.10.3 Mechanism of failure ... 115
7.10.4 Avoiding the vulnerability or mitigating its effects ... 115
7.11 Path traversal [EWR] ... 115
7.11.1 Description of application vulnerability ... 115

AVA Documents
Document Sharing Hub

© ISO/IEC 2019 – All rights reserved

Full document details: PD ISO/IEC TR 24772-1:2019

7.11.2 Cross reference 115
7.11.3 Mechanism of failure 116
7.11.4 Avoiding the vulnerability or mitigating its effects 117
7.12 Resource names [HTS] 117
7.12.1 7Description of application vulnerability 117
7.12.2 Cross reference 118
7.12.3 Mechanism of Failure 118
7.12.4 Avoiding the vulnerability or mitigating its effects 118
7.13 Resource exhaustion [XZP] 118
7.13.1 Description of application vulnerability 118
7.13.2 Cross reference 119
7.13.3 Mechanism of failure 119
7.13.4 Avoiding the vulnerability or mitigating its effects 119
7.14 Authentication logic error [XZO] 120
7.14.1 Description of application vulnerability 120
7.14.2 Cross reference 120
7.14.3 Mechanism of failure 120
7.14.4 Avoiding the vulnerability or mitigating its effects 121
7.15 Improper restriction of excessive authentication attempts [WPL] 121
7.15.1 Description of application vulnerability 121
7.15.2 Cross reference 121
7.15.3 Mechanism of failure 122
7.15.4 Avoiding the vulnerability or mitigating its effects 122
7.16 Hard-coded credentials [XYP] 122
7.16.1 Description of application vulnerability 122
7.16.2 Cross reference 122
7.16.3 Mechanism of failure 122
7.16.4 Avoiding the vulnerability or mitigating its effects 123
7.17 Insufficiently protected credentials [XYM] 123
7.17.1 Description of application vulnerability 123
7.17.2 Cross reference 123
7.17.3 Mechanism of failure 123
7.17.4 Avoiding the vulnerability or mitigating its effects 123
7.18 Missing or inconsistent access control [XZN] 124
7.18.1 Description of application vulnerability 124
7.18.2 Cross reference 124
7.18.3 Mechanism of failure 124
7.18.4 Avoiding the vulnerability or mitigating its effects 124
7.19 Incorrect authorization [BJE] 124
7.19.1 Description of application vulnerability 124
7.19.2 Cross reference 124
7.19.3 Mechanism of failure 125
7.19.4 Avoiding the vulnerability or mitigating its effects 125
7.20 Adherence to least privilege [XYN] 125
7.20.1 Description of application vulnerability 125
7.20.2 Cross reference 125
7.20.3 Mechanism of failure 125
7.20.4 Avoiding the vulnerability or mitigating its effects 125
7.21 Privilege sandbox issues [XYO] 126
7.21.1 Description of application vulnerability 126
7.21.2 Cross reference 126
7.21.3 Mechanism of failure 126
7.21.4 Avoiding the vulnerability or mitigating its effects 127
7.22 Missing required cryptographic step [XZS] 127
7.22.1 Description of application vulnerability 127
7.22.2 Cross reference 127
7.22.3 Mechanism of failure 127
7.22.4 Avoiding the vulnerability or mitigating its effects 127

AVA Documents
Document Sharing Hub

© ISO/IEC 2019 – All rights reserved

Full document details: PD ISO/IEC TR 24772-1:2019

7.23 Improperly verified signature [XZR] 128
7.23.1 Description of application vulnerability 128
7.23.2 Cross reference 128
7.23.3 Mechanism of failure 128
7.23.4 Avoiding the vulnerability or mitigating its effects 128
7.24 Use of a one-way hash without a salt [MVX] 128
7.24.1 Description of application vulnerability 128
7.24.2 Cross reference 128
7.24.3 Mechanism of failure 128
7.24.4 Avoiding the vulnerability or mitigating its effects 129
7.25 Inadequately secure communication of shared resources [CGY] 129
7.25.1 Description of application vulnerability 129
7.25.2 Cross-references 129
7.25.3 Mechanism of failure 129
7.25.4 Avoiding the vulnerability or mitigating its effect 130
7.26 Memory locking [XZX] 130
7.26.1 Description of application vulnerability 130
7.26.2 Cross reference 130
7.26.3 Mechanism of failure 131
7.26.4 Avoiding the vulnerability or mitigating its effects 131
7.27 Sensitive information not cleared before use [XZK] 131
7.27.1 Description of application vulnerability 131
7.27.2 Cross reference 131
7.27.3 Mechanism of failure 131
7.27.4 Avoiding the vulnerability or mitigating its effects 132
7.28 Time consumption measurement [CCM] 132
7.28.1 Description of application vulnerability 132
7.28.2 Cross-references 132
7.28.3 Mechanism of failure 132
7.28.4 Avoiding the vulnerability or mitigating its effect 132
7.29 Discrepancy information leak [XZL] 133
7.29.1 Description of application vulnerability 133
7.29.2 Cross reference 133
7.29.3 Mechanism of failure 133
7.29.4 Avoiding the vulnerability or mitigating its effects 133
7.30 Unspecified functionality [BVQ] 134
7.30.1 Description of application vulnerability 134
7.30.2 Cross reference 134
7.30.3 Mechanism of failure 134
7.30.4 Avoiding the vulnerability or mitigating its effects 134
7.31 Fault tolerance and failure strategies [REU] 134
7.31.1 Description of application vulnerability 134
7.31.2 Cross reference 135
7.31.3 Mechanism of failure 136
7.31.4 Avoiding the vulnerability or mitigating its effects 136
7.32 Distinguished values in data types [KLK] 137
7.32.1 Description of application vulnerability 137
7.32.2 Cross reference 137
7.32.3 Mechanism of failure 137
7.32.4 Avoiding the vulnerability or mitigating its effects 138
7.33 Clock issues [CCI] 138
7.33.1 Description of application vulnerability 138
7.33.2 Cross-references 139
7.33.3 Mechanism of failure 139
7.33.4 Avoiding the vulnerability or mitigating its effect 140
7.34 Time drift and jitter [CDJ] 140
7.34.1 Description of application vulnerability 140
7.34.2 Cross-references 141

AVA Documents
Document Sharing Hub

© ISO/IEC 2019 – All rights reserved

Full document details: PD ISO/IEC TR 24772-1:2019

7.34.3 Mechanism of failure ... 141
7.34.4 Avoiding the vulnerability or mitigating its effect ... 141

8 New vulnerabilities ... 142
8.1 General ... 142
8.2 Modifying constants [UJO] ... 142
8.2.1 Description of application vulnerability ... 142
8.2.2 Cross reference ... 142
8.2.3 Mechanism of failure ... 142
8.2.4 Applicable language characteristics ... 143
8.2.5 Avoiding the vulnerability or mitigating its effects ... 143
8.2.6 Implications for language design and evolution ... 143

Annex A (informative) **Vulnerability taxonomy and list ... 144**

Annex B (informative) **Selected guidance to language designers ... 152**

Annex C (informative) **Language-specific vulnerability template ... 154**

Bibliography ... 157

Index 159

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub

Full document details: PD ISO/IEC TR 24772-1:2019

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

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 document 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).

Attention is drawn to the possibility that some of the elements of this document can be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see http://patents.iec.ch).

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 assessment, 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 Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

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.

This first edition cancels and replaces ISO IEC TR 24772:2013, which has been split into several parts.

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

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

Introduction

All programming languages contain constructs that are incompletely specified, exhibit undefined behaviour, are implementation-dependent, or are difficult to use correctly. The use of those constructs can therefore give rise to *vulnerabilities*, as a result of which software programs can execute differently than intended by the writer. In some cases, these vulnerabilities can endanger the safety of a system or be exploited by attackers to compromise the security or privacy of a system.

This document provides users of programming languages with a language-independent overview of potential vulnerabilities in their usage and ways to avoid or mitigate them. Other parts of the ISO/IEC 24772 series describe how the general observations apply to specific languages.

This document is intended to provide guidance spanning multiple programming languages, so that application developers will be better able to avoid the programming constructs that lead to vulnerabilities in software written in their chosen language and their attendant consequences. This guidance can also be used by developers to select source code evaluation tools that can discover and eliminate some constructs that could lead to vulnerabilities in their software or to select a programming language that avoids anticipated problems.

The intended audience for this document are those who are concerned with assuring the predictable execution of the software of their system; that is, those who are developing, qualifying, or maintaining a software system and need to avoid language constructs that can cause the software to execute in a manner other than intended.

Developers of applications that have clear safety, security or mission-criticality requirements are expected to be aware of the risks associated with their code and can use this document to ensure that their development practices address the issues presented by the chosen programming languages, for example by subsetting or providing coding guidelines.

Specific audiences for this document include in particular developers, maintainers and regulators of:

— safety-critical applications that can cause loss of life, human injury, or damage to the environment;

— security-critical applications that need to ensure properties of confidentiality, integrity, and availability;

— mission-critical applications that need to avoid loss or damage to property or finance;

— business-critical applications where correct operation is essential to the successful operation of the business;

— scientific, modelling and simulation applications that require high confidence in the results of possibly complex, expensive and extended calculation.

However, it should not be assumed that other developers can ignore this document. A weakness in a non-critical application can provide the route by which an attacker gains control of a system or otherwise disrupts co-hosted applications that are critical. It is hoped that all developers would use this document to ensure that common vulnerabilities are removed or at least minimized from all applications.

While this document does not discuss specification or design issues, there is recognition that boundaries among the various activities are not clear-cut. This document seeks to avoid the debate about where low-level design ends and implementation begins by treating selected issues that some can consider design issues rather than coding issues.

It should be noted that this document is inherently incomplete. It is not possible to provide a complete list of programming language vulnerabilities because new weaknesses are discovered continually. Any such report can only describe those that have been found, characterized, and determined to have sufficient probability and consequence.

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

TECHNICAL REPORT **ISO/IEC TR 24772-1:2019(E)**

Programming languages — Guidance to avoiding vulnerabilities in programming languages —

Part 1: Language-independent guidance

1 Scope

This document specifies software programming language vulnerabilities to be avoided in the development of systems where assured behaviour is required for security, safety, mission-critical and business-critical software. Language-specific descriptions of these vulnerabilities are provided in other parts of the ISO/IEC 24772 series.

It is applicable to the software developed, reviewed, or maintained for any application.

This document does not address software engineering and management issues such as how to design and implement programs, use configuration management tools, use managerial processes, and perform process improvement. Furthermore, the specification of properties and applications to be assured are not treated.

Vulnerabilities are described in a generic manner that is applicable to a broad range of programming languages.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 2382, *Information technology — Vocabulary*

ISO 80000-2, *Quantities and units — Part 2: Mathematics*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382, ISO 80000-2 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

3.1 Terms related to communication

3.1.1
protocol
set of rules and supporting structures for the interaction of threads

Note 1 to entry: A protocol can be tightly embedded and rely on data in memory and hardware to control interaction of threads or can be applied to more loosely coupled arrangements, such as message communication spanning networks and computer systems.

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

3.1.2
stateless protocol
communication or cooperation between threads where no state is preserved in the protocol itself (example HTTP or direct access to a shared resource)

Note 1 to entry: Since most interaction between threads requires that state be preserved, the cooperating threads needs to use values of the resources(s) themselves or add additional communication exchanges to maintain state. Stateless protocols require that the application provide explicit resource protection and locking mechanisms to guarantee the correct creation, view, access to, modification of, and destruction of the resource – for example, the state needed for correct handling of the resource.

3.2 Terms related to execution model

3.2.1
thread
sequential stream of execution

Note 1 to entry: Although the term thread is used here and the context portrayed is that of shared-memory threads executing as part of a process, everything documented applies equally to other variants of concurrency such as interrupt handlers being enabled by a process, processes being created on the same system using operating system routines, or processes created as a result of distributed messages sent over a network. The mitigation approaches will be similar to those listed in the relevant vulnerability descriptions, but the implications for standardization would be dependent on how much language support is provided for the programming of the concurrent system.

3.2.2
thread activation
creation and setup of a thread up to the point where the thread begins execution

Note 1 to entry: A thread can depend on one or more other threads to define its access to other objects to be accessed and to determine its duration.

3.2.3
activated thread
thread that is created and then begins execution as a result of thread activation

3.2.4
activating thread
thread that exists first and makes the library calls or contains the language syntax that causes the activated thread to be activated

Note 1 to entry: The activating thread may wait, or not, for the activated thread to finish activation and may or may not check for errors if the activation fails. The activating thread can be permitted, or not, to terminate until after the activated thread terminates.

3.2.5
static thread activation
creation and initiation of a thread by program initiation, an operating system or runtime kernel, or by another thread as part of a declarative part of the thread before it begins execution

Note 1 to entry: In static activation, a static analysis can determine exactly how many threads will be created and how much resource, in terms of memory, processors, CPU cycles, priority ranges and inter-thread communication structures, will be needed by the executing program before the program begins.

3.2.6
dynamic thread activation
creation and initiation of a thread by another thread (including the main program) as an executable, repeatable command, statement or subprogram call

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

3.2.7
thread abort
request to stop and shut down a thread immediately

Note 1 to entry: The request is asynchronous if from another thread, or synchronous if from the thread itself. The effect of the abort request (such as whether it is treated as an exception) and its immediacy (that is, how long the thread can continue to execute before it is shut down) depend on language-specific rules. Immediate shutdown minimizes latency but can leave shared data structures in a corrupted state.

3.2.8
termination-directing thread
thread (including the OS) that requests the abortion of one or more threads

3.2.9
thread termination
completion and orderly shutdown of a thread, where the thread is permitted to make data objects consistent, release any acquired resources, and notify any dependent threads that it is terminating

Note 1 to entry: There are a number of steps in the termination of a thread as listed below, but depending on the multithreading model, some of these steps can be combined, explicitly programmed, or missing:

— the termination of programmed execution of the thread, including termination of any synchronous communication;

— the finalization of the local objects of the thread;

— waiting for any threads that can depend on the thread to terminate;

— finalization of any state associated with dependent threads;

— notification that finalization is complete, including possible notification of the activating task;

— removal and clean-up of thread control blocks and any state accessible by the thread or by other threads in outer scopes.

3.2.10
terminated thread
thread that has been halted from any further execution

3.2.11
master thread
thread which needs to wait for a terminated thread before it can take further execution steps (including termination of itself)

3.2.12
process
single execution of a program, or portion of an application

Note 1 to entry: Processes do not normally share a common memory space, but often share:

— processor;

— network;

— operating system;

— filing system;

— environment variables; or

— other resources.

Processes are usually started and stopped by an operating system and may or may not interact with other processes. A process can contain multiple threads.

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

3.3 Properties

3.3.1
software quality
degree to which software implements the requirements described by its specification and the degree to which the characteristics of a software product fulfil its requirements

3.3.2
predictable execution
property of the program such that all possible executions have results that can be predicted from the source code

3.4 Safety

3.4.1
safety hazard
potential source of harm

Note 1 to entry: IEC 61508-4[20] defines a *Hazard* as a *potential source of harm*, where *harm* is *physical injury or damage to the health of people either directly or indirectly as a result of damage to property or to the environment*. Some derived standards, such as UK Defence Standard 00-56, broaden the definition of *harm* to include *material and environmental damage (not just harm to people caused by property and environmental damage*).

3.4.2
safety-critical software
software for applications where failure can cause very serious consequences such as human injury or death

Note 1 to entry: IEC 61508-3[20] defines *Safety-related software* as *software that is used to implement safety functions in a safety-related system*. Although, in some domains a distinction is made between safety-related (can lead to any harm) and safety-critical (life threatening), this document uses the term *safety-critical* for all vulnerabilities that can result in safety hazards.

3.5 Vulnerabilities

3.5.1
application vulnerability
security vulnerability or safety hazard, or defect

3.5.2
language vulnerability
property (of a programming language) that can contribute to, or that is strongly correlated with, application vulnerabilities in programs written in that language.

Note 1 to entry: The term *property* can mean *the presence or the absence of a specific feature, used singly or in combination*. As an example of the absence of a feature, encapsulation (control of where names can be referenced from) is generally considered beneficial since it narrows the interface between modules and can help prevent data corruption. The absence of encapsulation from a programming language can thus be regarded as a vulnerability. Note that a property together with its complement can both be considered language vulnerabilities. For example, automatic storage reclamation (garbage collection) can be a vulnerability since it can interfere with time predictability and result in a safety hazard. On the other hand, the absence of automatic storage reclamation can also be a vulnerability since programmers can mistakenly free storage prematurely, resulting in dangling references.

3.5.3
security vulnerability
weakness in an information system, system security procedures, internal controls, or implementation that can be exploited or triggered by a threat

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

3.5.4
failure
malfunction of the system or component which has as subcategories omission failure, commission failure, timing failure and value failure

3.5.5
omission failure
service that is requested but never rendered

Note 1 to entry: The client can wait forever or can be notified about the failure (termination) of the service.

3.5.6
commission failure
service that initiates unexpected actions, e. g., communication that is unexpected by the receiver

Note 1 to entry: The service can wait forever, causing omission failures for subsequent calls by clients. The receiver can be hindered to do its legitimate actions in time. At a minimum, resources are consumed that are possibly needed by others.

3.5.7
timing failure
service that is not rendered before an imposed deadline

Note 1 to entry: This results in a system response that is (too) late, causing corresponding damages to the real world affected by the system.

3.1.5.8
value failure
service delivers incorrect or tainted results

Note 1 to entry: The client continues computations with these corrupted values, causing a spread of consequential application errors.

4 Applying this document

This document has been written with several possible usages in mind:

— programmers familiar with the vulnerabilities of a specific language can reference the guide for more generic descriptions and their manifestations in less familiar languages;

— tool vendors can use the three-letter codes as a succinct way to "profile" the selection of vulnerabilities considered by their tools;

— individual organizations can wish to write their own coding standards intended to reduce the number of vulnerabilities in their software products. the guide can assist in the selection of vulnerabilities to be addressed in those standards and the selection of coding guidelines to be enforced;

— organizations or individuals selecting a language for use in a project can want to consider the vulnerabilities inherent in various candidate languages;

— scientists, engineers, economists, statisticians, or others who write computer programs as tools of their chosen craft can read this document to become more familiar with the issues that can affect their work.

There are a number of ways to avoid a vulnerability:

— avoiding the particular coding constructs that are found to be problematic;

— using static analysis tools to detect anomalous situations, including usage of a compiler that provides warnings if a construct is problematic;

— choosing a programming language that avoids or mitigates a class of vulnerabilities;

AVA Documents
Document Sharing Hub

© ISO/IEC 2019 – All rights reserved

Full document details: PD ISO/IEC TR 24772-1:2019

— writing specific runtime checks to detect situations that can lead to problematic behaviour;

— used automated analysis tools to enforce coding standards;

— undertaking verification and validation methods such as focused human review of code.

This document gathers descriptions of programming language vulnerabilities, as well as selected application vulnerabilities, which have occurred in the past and are likely to occur again. Every vulnerability discussed here has been experienced in at least one programming language or runtime environment. Some vulnerabilities occur in all programming languages, while others are mitigated by the features or capabilities of some programming environments.

Each vulnerability and its possible mitigations are described in this document in a language-independent manner, though illustrative examples can be language-specific. In addition, other parts of the ISO/IEC 24772 series for particular languages describe the vulnerabilities and their mitigations in a manner specific to the language.

Clause 5 explains how many of the vulnerabilities common to programming languages occur. The issues discussed are not vulnerabilities but language characteristics that lead to mistakes and vulnerabilities that can be exploited. Table 1 provides a summary list of the top 21 approaches to avoiding the most common vulnerabilities with references to Clauses 6 and 7. For many that cannot invest the resources to research all of the vulnerabilities documented in Clauses 6, 7, and 8, implementing the guidance in 5.4 already provides significant benefit to their projects.

Clause 6 provides language-independent descriptions of vulnerabilities in programming languages that can lead to application vulnerabilities. Each description provides a summary of the vulnerability, characteristics of languages where the vulnerability can be found, typical mechanisms of failure, techniques that programmers can use to avoid the vulnerability, and ways that language designers can modify language specifications in the future to help programmers mitigate the vulnerability.

In using Clause 6, it is important to be aware of how a listed vulnerability is presented by the programming language, tool environment, and operating system that is being used. To help, this document is supported by ISO/IEC TR 24772-2 (for Ada), ISO/IEC TR 24772-3 (for C), etc. Each part of the ISO/IEC 24772 series:

— is named for a particular programming language, lists the vulnerabilities described in Clause 6;

— describes how each vulnerability appears (or does not appear) in that specific language; and

— specifies how it can be mitigated in that language, whenever possible.

Clause 7 provides descriptions of selected vulnerabilities which:

— have been found and exploited in a number of applications;

— have well known mitigation techniques; and

— result from design decisions made by coders in the absence of suitable language library routines or other mechanisms.

For these vulnerabilities, each description provides:

— a summary of the vulnerability;

— typical mechanisms of failure; and

— techniques that programmers can use to avoid the vulnerability.

Mitigations for vulnerabilities listed in Clause 7 do not include the use of programming language-specific features or choices but consist of alternate design choices or programming techniques.

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

Clause 8 provides new vulnerabilities that have not yet had corresponding programming language-specific text developed. The form and material matches the vulnerabilities of Clause 6, but vulnerability writeups in the language-specific parts may not yet exist.

Annex A is a categorization of the vulnerabilities of this document in the form of a hierarchical outline and a list of the vulnerabilities arranged in alphabetic order by their three-letter code.

Annex B summarizes the guidance to language designers found in Clauses 6.X.6.

Annex C is a template for the writing of programming language-specific parts of the ISO/IEC 24772 series that explain how the vulnerabilities from Clause 6 are realized in that programming language (or show how they are absent), and how they can be mitigated in language-specific terms.

5 Vulnerability issues and general avoidance mechanisms

5.1 Predictable execution

There are many reasons why software does not execute as expected by its developers, its users or other stakeholders. Reasons include incorrect specifications, configuration management errors and a myriad of others. This document focuses on one cause: the usage of programming languages in ways that render the execution of the code less predictable.

Predictable execution is a property of a program such that all possible executions have results that can be predicted from examination of the source code. Achieving predictability is complicated by that fact that software can be used:

— on unanticipated platforms (for example, ported to a different processor);

— in unanticipated ways (as usage patterns change);

— in unanticipated contexts (for example, software reuse and system-of-system integrations); and

— by unanticipated users (for example, those seeking to exploit and penetrate a software system).

Furthermore, today's ubiquitous connectivity of software systems virtually guarantees that most software will be attacked — either because it is a target for penetration or because it offers a springboard for penetration of other software. Accordingly, today's programmers need to take additional care to ensure predictable execution despite the new challenges.

Software vulnerabilities are unwanted characteristics of software that can allow software to execute in ways that are unexpected. Programmers introduce vulnerabilities into software by using language features that are inherently unpredictable in the variable circumstances outlined above or by using features in a manner that reduces what predictability they can offer. Of course, complete predictability is an ideal (particularly because new vulnerabilities are often discovered through experience), but any programmer can improve predictability by carefully avoiding the introduction of known vulnerabilities into code.

This document focuses on a particular class of vulnerabilities, *language vulnerabilities*. These are properties of programming languages that can contribute to (or are strongly correlated with) *application vulnerabilities* — security weaknesses, safety hazards, or defects. An example may clarify the relationship. The programmer's use of a string copying function that does not check length can be exploited by an attacker to place incorrect return values on the program stack, hence passing control of the execution to code provided by the attacker. The string copying function is the language vulnerability and the resulting weakness of the program in the face of the stack attack is the application vulnerability. The programming language vulnerability enables the application vulnerability. The language vulnerability can be avoided by using a string copying function that does set appropriate bounds on the length of the string to be copied. By using a bounded copy function, the programmer improves the predictability of the code's execution.

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

The primary purpose of this document is to survey common programming language vulnerabilities (see Clause 6). Each description explains how an application vulnerability can result. In Clause 7, a few additional application vulnerabilities are described. These are selected because they are associated with language weaknesses even if they do not directly result from language vulnerabilities. For example, a programmer can have stored a password in plain text (see 7.16) because the programming language did not provide a suitable library function for storing the password in a non-recoverable format.

In addition to considering the individual vulnerabilities, it is instructive to consider the sources of uncertainty that can decrease the predictability of software. These sources are briefly considered in the remainder of this clause.

5.2 Sources of unpredictability in language specification

5.2.1 Incomplete or evolving specification

The design and specification of a programming language involves considerations that are very different from the use of the language in programming. Language specifiers are often required by customers to maintain compatibility with older versions of the language — even to the extent of retaining inherently vulnerable features. Sometimes, the semantics of new or complex features are not completely known, especially when used in combination with other features.

5.2.2 Undefined behaviour

It is simply not possible for the specifier of a programming language to describe every possible behaviour. For example, the result of using a variable to which no value has been assigned is left undefined by many languages. In such cases, a program can do anything — including crashing with no diagnostic or executing with wrong data, leading to incorrect results.

5.2.3 Unspecified behaviour

The behaviour of some features may be incompletely defined. The language implementer would have to choose from a finite set of choices, but the choice may not be apparent to the programmer. In such cases, different compilers can lead to different results.

5.2.4 Implementation-defined behaviour

In some cases, the results of execution can depend on characteristics of the compiler that was used, the processor on which the software is executed, or the other systems with which the software has interfaces. In principle, one can predict the execution with sufficient knowledge of the implementation, but such knowledge is sometimes difficult to obtain. Furthermore, dependence on a specific implementation-defined behaviour will lead to problems when a different processor or compiler is used — sometimes even if different compiler switch settings are used.

5.2.5 Difficult features

Some language features can be difficult to understand or to use appropriately, either due to complicated semantics (for example, floating point in numerical analysis applications) or human limitations (for example, deeply nested program constructs or expressions). Sometimes simple typing errors can lead to major changes in behaviour without a diagnostic (for example, typing “=” for assignment when one really intended “==” for comparison).

5.2.6 Inadequate language support

No language is suitable for every possible application. Furthermore, programmers sometimes do not have the freedom to select the language that is most suitable for the task at hand. In many cases, libraries are used to supplement the functionality of the language. Then, the library itself becomes a potential source of uncertainty reducing the predictability of execution.

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

5.3 Sources of unpredictability in language usage

5.3.1 Porting and interoperation

When a program is recompiled using a different compiler, recompiled using different switches, executed with different libraries, executed on a different platform, or even interfaced with different systems, its behaviour will change. Such changes result from different choices for unspecified and implementation-defined behaviour, differences in library function, and differences in underlying hardware and operating system support. The problem is far worse if the original programmer chose to use implementation-dependent extensions to the language rather than staying with the standardized language.

5.3.2 Compiler selection and usage

Nearly all software has bugs and compilers are no exception. They should be carefully selected from trusted sources and qualified prior to use. Perhaps less obvious, though, is the use of compiler switches. Different switch settings can result in differences in generated code. A careful selection of settings can improve the predictability of code, for example, a setting that causes the flagging of any usage of an implementation-defined behaviour.

5.4 Top avoidance mechanisms

Each vulnerability listed in Clauses 6 and 7 provides a set of ways that the vulnerability can be avoided or mitigated. Many of the mitigations and avoidance mechanisms are common. Table 1 provides the most effective and the most common mitigations, together with references to which vulnerabilities they apply. The references are hyperlinked to provide the reader with easy access to those vulnerabilities for rationale and further exploration.

The expectation is that users of this document will develop and use a coding standard based on this document that is tailored to their risk environment.

Table 1 — Recommended avoidance mechanisms

Number	Recommended avoidance mechanism	References
1	Validate input. Do not make assumptions about the values of parameters. Check parameters for valid ranges and values in the calling and/or called functions before performing any operations.	6.6 7.13 7.18 7.28
2	When functions return error values, check the error return values before processing any other returned data.	6.36 6.60
3	Enable compiler static analysis checking and resolve compiler warnings.	6.8 6.10 6.14 6.15 6.16 6.17 6.18 6.19 6.22 6.25 6.26 6.27 6.29 6.30 6.34 6.36 6.38 6.39 6.47 6.54 6.56 6.57 6.60 6.61 6.62 7.28.

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

Number	Recommended avoidance mechanism	References
4	Run a static analysis tool to detect anomalies not caught by the compiler.	6.3 6.6 6.7 6.8 6.10 6.14 6.15 6.16 6.17 6.18 6.19 6.22 6.25 6.26 6.27 6.29 6.30 6.34 6.36 6.38 6.39 6.47 6.54 6.56 6.57 6.60 6.61 6.62 7.28
5	Perform explicit range checking when it cannot be shown statically that ranges will be obeyed, when range checking is not provided by the implementation, or if automatic range checking is disabled.	6.6 6.8 6.16
6	Allocate and free resources, such as memory, threads or locks, at the same level of abstraction.	6.14
7	Avoid constructs that have unspecified but bounded behaviour, and if the construct is needed, test for all possible behaviours.	6.24 6.56
8	Make error detection, error reporting, error correction, and recovery an integral part of a system design.	6.36
10	Use only those features of the programming language that enforce a logical structure on the program.	6.31
11	Avoid using features of the language which are not specified to an exact behaviour or that are undefined, implementation-defined or deprecated.	6.55 6.56 6.57 6.58 6.59
12	Avoid using libraries without proper signatures.	6.34
13	Do not modify loop control variables inside the loop body.	6.29
14	Do not perform assignments within Boolean expressions, even if allowed by the language.	6.25
15	Do not depend on side effects of a term in the expression itself.	6.31 6.24
16	Use names that are clear and visually unambiguous. Be consistent in choosing names.	6.17
17	Use careful programming practice when programming border cases.	6.6 6.29 6.30
18	Be aware of short-circuiting behaviour when expressions with side effects are used on the right side of a Boolean expression such as if the first expression evaluates to `false` in an and expression, then the remaining expressions, including functions calls, will not be evaluated.	6.24 6.25
19	Avoid fall-through from one case (or switch) statement into the following case statement: if a fall-through is necessary then provide a comment to inform the reader that it is intentional.	6.27
20	Do not use floating-point arithmetic when integers would suffice, especially for counters associated with program flow, such as loop control variables.	6.4
21	Sanitize, erase or encrypt data that will be visible to others (for example, freed memory, transmitted data).	7.11 7.12

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

6 Programming language vulnerabilities

6.1 General

This clause provides language-independent descriptions of vulnerabilities in programming languages that can lead to application vulnerabilities. Each description provides:

— a summary of the vulnerability;

— characteristics of languages where the vulnerability can be found;

— typical mechanisms of failure;

— techniques that programmers can use to avoid the vulnerability; and

— ways that language designers can modify language specifications in the future to help programmers mitigate the vulnerability.

Descriptions of how vulnerabilities are manifested, in particular, programming languages are provided in other parts of the ISO/IEC 24772 series. In each case, the behaviour of the language is assumed to be as specified by the standard cited in the respective part. Clearly, programs can have different vulnerabilities in a non-standard implementation. Examples of non-standard implementations include:

— compilers written to implement some specification other than the standard;

— use of non-standard vendor extensions to the language; and

— use of compiler switches providing alternative semantics.

The following descriptions are written in a language-independent manner except when specific languages are used in examples. The language-specific parts can be consulted for language-specific descriptions.

In general, this clause uses the terminology that is most natural to the description of each individual vulnerability. Therefore, terminology can differ from description to description.

6.2 Type system [IHN]

6.2.1 Description of application vulnerability

When data values are converted from one data type to another, even when done intentionally, unexpected results can occur.

6.2.2 Cross reference

JSF AV Rules[31]: 148 and 183

MISRA C[35]: 4.6, 10.1, 10.3, and 10.4

MISRA C++[36]: 3-9-2, 5-0-3 to 5-0-14

CERT C guidelines[38]: DCL07-C, DCL11-C, DCL35-C, EXP05-C and EXP32-C

Ada Quality and Style Guide[1]: 3.4

6.2.3 Mechanism of failure

The *type* of a data object informs the compiler on how values should be represented and which operations are legal for the type. The *type system* of a language is the set of rules used by the language to structure and organize its collection of types. Any attempt to manipulate data objects with

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

inappropriate operations is a *type error*. A program is said to be *type safe* (or *type secure*) if it can be demonstrated that it has no type errors[27].

Every programming language has some sort of type system. A language is *statically typed* if the type of every expression is known at compile time. The type system is said to be *strong* if it guarantees type safety and *weak* if it does not. There are strongly typed languages that are not statically typed because they enforce type safety with runtime checks[27].

In practical terms, nearly every language falls short of being strongly typed (in an ideal sense) because of the inclusion of mechanisms to bypass type safety in particular circumstances. For that reason, and because every language has a different type system, this description focuses on taking advantage of whatever features for type safety are available in the chosen language.

Sometimes it is appropriate for a data value to be converted from one type to another *compatible* one. For example, consider the following program fragment, written in no specific language:

```
float a; integer i; a := a + i;
```

The variable "i" is of integer type. It is converted to the float type before it is added to the data value. This is an *implicit type conversion*. If, on the other hand, the conversion needs to be specified by the program, for example,

```
a := a + float(i)
```

then it is an *explicit type conversion*.

Type *equivalence* is the strictest form of type compatibility; two types are equivalent if they are compatible without using implicit or explicit conversion. Type equivalence is usually characterized in terms of *name type equivalence*—two variables have the same type if they are declared in the same declaration or declarations that use the same type name—or *structure type equivalence*—two variables have the same type if they have identical structures. There are variations of these approaches and most languages use different combinations of them[28]. Therefore, a programmer skilled in one language can very well code inadvertent type errors when using a different language.

It is desirable for a program to be type safe because the application of operations to operands of an inappropriate type can produce unexpected results. In addition, the presence of type errors can reduce the effectiveness of static analysis for other problems. Searching for type errors is a valuable exercise because their presence often reveals design errors as well as coding errors. Many languages check for type errors—some at compile-time, others at run-time. Obviously, compile-time checking is more valuable because it can catch errors that are not executed by a particular set of test cases.

Making the most use of the type system of a language is useful in two ways. First, data conversions always bear the risk of changing the value. For example, a conversion from integer to float risks the loss of significant digits while the inverse conversion risks the loss of any fractional value. Conversion of an integer value from a type with a longer representation to a type with a shorter representation risks the loss of significant digits. This can produce particularly puzzling results if the value is used to index an array. Conversion of a floating-point value from a type with a longer representation to a type with a shorter representation risks the loss of precision. This can be particularly severe in computations where the number of calculations increases as a power of the problem size. (It should be noted that similar surprises can occur when an application is retargeted to a machine with different representations of numeric values.)

Second, a programmer can use the type system to increase the probability of catching design errors or coding blunders. For example, the following Ada fragment declares two distinct floating-point types:

```
type Celsius is new Float; type Fahrenheit is new Float;
```

The declaration makes it impossible to add a value of type Celsius to a value of type Fahrenheit without explicit conversion. Even explicit conversions also require additional numeric calculations that respect the relationship of the real-world units being converted. For example F = C (where F is Fahrenheit and C is Celcius) only works when C=-40, otherwise one needs F = convert_to_fahrenheit(C) which performs 9*C/5+32.

As another example, the following Pascal code:

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

```
type AltitudeInFeet = -1500.. 45000;
```

defines the operating range of a plane and lets the compiler decide on the appropriate underlying representation in contrast to a predefined type `integer` which can be represented in 16 bits (insufficient for the purpose) or 32 bits, depending on the target architecture.

6.2.4 Applicable language characteristics

This vulnerability is intended to be applicable to languages that support multiple types and allow conversions between types.

6.2.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

— Take advantage of any facility offered by the programming language to declare distinct types and use any mechanism provided by the language processor and related tools to check for or enforce type compatibility.

— Use available language and tools facilities to preclude or detect the occurrence of implicit type conversions, such as those in mixed type arithmetic. If it is not possible, use human review to assist in searching for implicit conversions.

— Avoid explicit type conversion of data values except when there is no alternative. Document such occurrences so that the justification is made available to maintainers.

— Use the most restricted data type that suffices to accomplish the job. For example, use an enumeration type to select from a limited set of choices (such as, a switch statement or the discriminant of a union type) rather than a more general type, such as integer. This will make it possible for tooling to check if all possible choices have been covered.

— Always respect the implied unit systems, when converting explicitly from one numeric type to another.

— Treat every compiler, tool, or run-time diagnostic concerning type compatibility as a serious issue. Do not resolve the problem by modifying the code to include an explicit conversion, without further analysis; instead examine the underlying design to determine if the type error is a symptom of a deeper problem.

— Never ignore instances of implicit type conversion; if the conversion is necessary, change it to an explicit conversion and document the rationale for use by maintainers.

— Analyse the problem to be solved to learn the magnitudes and/or the precisions of the quantities needed as auxiliary variables, partial results and final results.

— Create types that more accurately model the problem domain, with corresponding safe operations and conversions in lieu of using primitive types.

— Minimize use of predefined numeric types whose ranges and precisions are implementation defined. Instead, use types whose ranges and precision are guaranteed.

6.2.6 Implications for language design and evolution

In future language design and evolution activities, the following items should be considered.

— Language specifiers should standardize on a common, uniform terminology to describe their type systems so that programmers experienced in other languages can reliably learn the type system of a language that is new to them.

— Provide a mechanism for selecting data types with sufficient capability for the problem at hand.

— Provide a way for the computation to determine the limits of the data types actually selected.

Documents
Document Sharing Hub

© ISO/IEC 2019 – All rights reserved

Full document details: PD ISO/IEC TR 24772-1:2019

— Language implementers should consider providing compiler switches or other tools to provide the highest possible degree of checking for type errors.

6.3 Bit representations [STR]

6.3.1 Description of application vulnerability

Interfacing with hardware, other systems and protocols often requires access to one or more bits in a single computer word, or access to bit fields that cross computer words for the machine in question. Mistakes can be made as to what bits are to be accessed because of the *endianness* of the processor (see below) or because of miscalculations. Access to those specific bits can affect surrounding bits in ways that compromise their integrity. This can result in the wrong information being read from hardware, incorrect data or commands being given, or information being mangled, which can result in arbitrary effects on components attached to the system.

6.3.2 Cross reference

JSF AV Rules[31] 147, 154 and 155

MISRA C[35]: 1.1, 6.1, 6.2, and 10.1

MISRA C++[36]: 5-0-21, 5-2-4 to 5-2-9, and 9-5-1

CERT C guidelines[38]: EXP38-C, INT00-C, INT07-C, INT12-C, INT13-C, and INT14-C

Ada Quality and Style Guide[1]: 7.6.1 through 7.6.9, and 7.3.1

Hogaboom, Richard, *A Generic API Bit Manipulation in C*[17]

6.3.3 Mechanism of failure

Computer languages frequently provide a variety of sizes for integer variables. Languages may support short, integer, long, and even big integers. Interfacing with protocols, device drivers, embedded systems, low level graphics or other external constructs may require each bit or set of bits to have a particular meaning. Those bit sets may or may not coincide with the sizes supported by a particular language implementation. When they do not, it is common practice to pack all of the bits into one word. Masking and shifting of the word using powers of two to pick out individual bits or using sums of powers of 2 to pick out subsets of bits (for example, using $28 = 2^2 + 2^3 + 2^4$ to create the mask 11100 and then shifting 2 bits) provides a way of extracting those bits. Knowledge of the underlying bit storage is usually not necessary to accomplish simple extractions such as these. Problems can arise when programmers mix their techniques (e.g. arithmetic and logical operations) to reference the bits or output the bit, since storage ordering of the bits may not be what the programmer expects.

Packing of bits in an integer is not inherently problematic. However, an understanding of the intricacies of bit level programming is necessary. Some computers or other devices store the bits left-to-right while others store them right-to-left. The kind of storage can cause problems when interfacing with external devices that expect the bits in the opposite order. One problem arises when assumptions are made when interfacing with external constructs and the ordering of the bits or words are not the same as the receiving entity. Programmers can inadvertently use the sign bit in a bit field and then may not be aware that an arithmetic shift (sign extension) is being performed when right shifting causing the sign bit to be extended into other fields. Alternatively, a left shift can cause the sign bit to be one. Bit manipulations can also be problematic when the manipulations are done on binary encoded records that span multiple words. The storage and ordering of the bits need to be considered when doing bit-wise operations across multiple words as bytes may be stored in big-endian or little-endian format.

6.3.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that allow bit manipulations.

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

6.3.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

— Explicitly document any reliance on bit ordering such as explicit bit patterns, shifts, or bit numbers.

— Understand the way bit ordering is done on the host system and on the systems with which the bit manipulations will be interfaced.

— Where the language supports it, use bit fields in preference to binary, octal, or hex representations.

— Avoid bit operations on signed operands.

— Localize and document the code associated with explicit manipulation of bits and bit fields.

— Use static analysis tools that identify and report reliance on bit ordering or bit representation.

6.3.6 Implications for language design and evolution

In future language design and evolution activities, for languages that are commonly used for bit manipulations, consider creating an *API* (Application Programming Interface) for bit manipulations that is independent of word size and machine instruction set should be defined and standardized.

6.4 Floating-point arithmetic [PLF]

6.4.1 Description of application vulnerability

Most real numbers cannot be represented exactly in a computer. To represent real numbers, most computers use IEC 60559[30]. If it is not followed, then the bit representation for a floating-point number can vary from compiler to compiler and on different platforms. However, relying on a particular representation can cause problems when a different compiler is used or the code is reused on another platform. Regardless of the representation, many real numbers can only be approximated since representing the real number using a binary representation may well require an endlessly repeating string of bits or more binary digits than are available for representation. Therefore, it should be assumed that a floating-point number is only an approximation, even though it can be an extremely good one. Floating-point representation of a real number or a conversion to floating-point can cause surprising results and unexpected consequences to those unaccustomed to the idiosyncrasies of floating-point arithmetic.

Many algorithms that use floating point can have anomalous behaviour when used with certain values. The most common results are erroneous results or algorithms that never terminate for certain segments of the numeric domain, or for isolated values. Those without training or experience in numerical analysis may not be aware of which algorithms, or, for a particular algorithm, of which domain values should be the focus of attention.

In some hardware, precision for intermediate floating-point calculations may be different than that suggested by the data type, causing different rounding results when moving to standard precision modes.

6.4.2 Cross reference

JSF AV Rules[31]: 146, 147, 184, 197, and 202

MISRA C[35]: 1.1 and 14.1

MISRA C++[36]: 0-4-3, 3-9-3, and 6-2-2

CERT C guidelines[38]: FLP00-C, FP01-C, FLP02-C and FLP30-C

Ada Quality and Style Guide[1]: 5.5.6 and 7.2.1 through 7.2.8

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

6.4.3 Mechanism of failure

Floating-point numbers are generally only an approximation of the actual value. Expressed in base 10 world, the value of 1/3 is 0,333333… The same type of situation occurs in the binary world, but the numbers that can be represented with a limited number of digits in base 10, such as 1/10=0,1 become endlessly repeating sequences in the binary world. So 1/10 represented as a binary number is:

```
0.0001100110011001100110011001100110011001100110011…
```

Which is `0*1/2 + 0*1/4 + 0*1/8 + 1*1/16 + 1*1/32 + 0*1/64…` and no matter how many digits are used, the representation will still only be an approximation of 1/10. Therefore, when adding 1/10 ten times, the final result may or may not be exactly 1.

Accumulating floating point values through the repeated addition of values, particularly relatively small values, can provide unexpected results. Using an accumulated value to terminate a loop can result in an unexpected number of iterations. Rounding and truncation can cause tests of floating-point numbers against other values to yield unexpected results. Another cause of floating point errors is reliance on comparisons of floating point values or the comparison of a floating point value with zero. Tests of equality or inequality can vary due to rounding or truncation errors, which can propagate far from the operation of origin. Even comparisons of constants can fail when a different rounding mode was employed by the compiler and by the application. Differences in magnitudes of floating-point numbers can result in no change of a very large floating-point number when a relatively small number is added to or subtracted from it.

Manipulating bits in floating-point numbers is also very implementation dependent if the implementation is not ISO/IEC/IEEE 60559[30] compliant or in the interpretation of NAN's. Typically, special representations are specified for positive and negative zero; infinity and subnormal numbers very close to zero. Relying on a particular bit representation is inherently problematic, especially when a new compiler is introduced or the code is reused on another platform. The uncertainties arising from floating-point can be divided into uncertainty about the actual bit representation of a given value (such as, big-endian or little-endian) and the uncertainty arising from the rounding of arithmetic operations (for example, the accumulation of errors when imprecise floating-point values are used as loop indices).

Note that most floating-point implementations are binary. Decimal floating-point numbers are available on some hardware and have been standardized in ISO/IEC/IEEE 60559[30] but be aware what precision guarantees your programming language makes. In general, fixed point arithmetic can be a better solution to common problems involving decimal fractions (such as financial calculations).

Implementations (libraries) for different precisions are often implemented in the highest precision. This can yield different results in algorithms such as exponentiation than if the programmer had performed the calculation directly.

Floating-point systems have more than one rounding mode. Round to the nearest even number is the default for almost all implementations. Repeatedly rounding iterative calculations towards zero or away from zero can result in a loss of precision, and can cause unexpected outcome.

Floating-point Min and Max can return an arbitrary sign when both parameters are zero (and of different sign). Tests that use the sign of a number rather than its relationship to zero can return unexpected results.

6.4.4 Applicable language characteristics

This vulnerability description is intended to be applicable to all languages with floating-point operations, since floating-point variables can be subject to rounding or truncation errors.

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

6.4.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

— Unless the program's use of floating-point is trivial, obtain the assistance of an expert in numerical analysis and in the hardware properties of your system to check the stability and accuracy of the algorithm employed.

— Do not use a floating-point expression in a Boolean test for equality unless it can be shown that the logic implemented by the equality test cannot be affected by prior rounding errors. Instead, use coding that determines the difference between the two values to determine whether the difference is acceptably small enough so that two values can be considered equal. Note that if the two values are very large, the "small enough" difference can be a very large number.

— Verify that the underlying implementation is compliant with ISO/IEC/IEEE 60559[30] or that it includes subnormal numbers (fixed point numbers that are close to zero). Be aware that implementations that do not have this capability can underflow to zero in unexpected situations.

— Be aware that infinities, NAN and subnormal numbers are possible and give special consideration to tests that check for those conditions before using them in floating point calculations.

— Use library functions with known numerical characteristics. Avoid the use of a floating-point variable as a loop counter. If it is necessary to use a floating-point value for loop control, use inequality to determine the loop control (that is, <, <=, > or >=).

— Understand the floating-point format used to represent the floating-point numbers. This will provide some understanding of the underlying idiosyncrasies of floating-point arithmetic.

— Avoid manipulating the bit representation of a floating-point number. Prefer built-in language operators and functions that are designed to extract the mantissa, exponent or sign.

— Do not use floating-point for exact values such as monetary amounts. Use floating-point only when necessary such as for fundamentally inexact values such as measurements or values of diverse magnitudes. Consider the use of fixed point arithmetic/libraries or decimal floating point when appropriate.

— Use known precision modes to implement algorithms.

— Avoid changing the rounding mode from RNE (round nearest even).

— Avoid reliance on the sign of the floating-point `Min` and `Max` operations when both numbers are zero.

— When adding (or subtracting) sequences of numbers, sort and add (or subtract) them from smallest to largest in absolute value, or use a suitable compensated summation algorithm to avoid loss of precision.

6.4.6 Implications for language design and evolution

In future language design and evolution activities, the following items should be considered.

— Languages that do not already adhere to or only adhere to a subset of ISO/IEC/IEEE 60559[30] should consider adhering completely to the standard. Examples of standardization that should be considered.

— Languages should consider providing a means to generate diagnostics for code that attempts to test equality of two floating point values.

— Languages should consider standardizing their data type to ISO/IEC 10967-1 and ISO/IEC 10967-2.

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

6.5 Enumerator issues [CCB]

6.5.1 Description of application vulnerability

Enumerations are a finite list of named entities that contain a fixed mapping from a set of names to a set of integral values (called the representation) and an order between the members of the set. In some languages, there are no other operations available except order, equality, first, last, previous, and next; in others, the full underlying representation operators are available, such as integer "+" and "-" and bit-wise operations.

Most languages that provide enumeration types also provide mechanisms to set non-default representations. If these mechanisms do not enforce whole-type operations and check for conflicts then some members of the set may not be properly specified or can have the wrong mappings. If the value-setting mechanisms are positional only, then there is a risk that improper counts or changes in relative order will result in an incorrect mapping.

For arrays indexed by enumerations with non-default representations, there is a risk of structures with holes, and if those indexes can be manipulated numerically, there is a risk of out-of-bound accesses of these arrays.

Most of these errors can be readily detected by static analysis tools with appropriate coding standards, restrictions and annotations. Similarly mismatches in enumeration value specification can be detected statically. Without such rules, errors in the use of enumeration types are computationally hard to detect statically as well as being difficult to detect by human review.

6.5.2 Cross reference

MISRA C[35]: 8.12, 9.2, and 9.3

MISRA C++[36]: 8-5-3

CERT C guidelines[38]: INT09-C

Holzmann[18] rule 6

Ada Quality and Style Guide[1]: 3.4.2

6.5.3 Mechanism of failure

As a program is developed and maintained the list of items in an enumeration often changes in three basic ways: new elements are added to the list; order between the members of the set often changes; and representation (the map of values of the items) change. Expressions that depend on the full set or specific relationships between elements of the set can create value errors that can result in wrong results or in unbounded behaviours if used as array indices.

Improperly mapped representations can result in some enumeration values being unreachable, or can create *holes* in the representation where values that cannot be defined are propagated.

If arrays are indexed by enumerations containing non-default representations, some implementations may leave space for values that are unreachable using the enumeration, with a possibility of unnecessarily large memory allocations or a way to pass information undetected (hidden channel).

When enumerators are set and initialized explicitly, and the language permits incomplete initializers, then changes to the order of enumerators or the addition or deletion of enumerators can result in the wrong values being assigned or default values being assigned improperly. Subsequent indexing can result in invalid accesses and possibly unbounded behaviours.

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

6.5.4 Applicable language characteristics

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

— Languages that permit incomplete mappings between enumerator specification and value assignment, or that provide a positional-only mapping require additional static analysis tools and annotations to help identify the complete mapping of every literal to its value.

— Languages that provide a trivial mapping to a type such as integer require additional static analysis tools to prevent mixed type errors. They also cannot prevent invalid values from being placed into variables of such enumerator types. For example:

```
enum Directions {back, forward, stop};

enum Directions a = forward, b = stop, c = a + b;
```

In this example, c can have a value not defined by the enumeration, and any further use as that enumeration will lead to erroneous results.

— Some languages provide no enumeration capability, leaving it to the programmer to define named constants to represent the values and ranges.

6.5.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

— Use static analysis tools that will detect inappropriate use of enumerators, such as using them as integers or bit maps, and that detect enumeration definition expressions that are incomplete or incorrect. For languages with a complete enumeration abstraction this is the compiler.

— In code that performs different computations depending on the value of an enumeration, ensure that each possible enumeration value is covered, or provide a default that raises an error or exception.

— Use an enumerated type to select from a limited set of choices and use tools that statically detect omissions of possible values in an enumeration.

6.5.6 Implications for language design and evolution

In future language design and evolution activities, the following items should be considered.

— Languages that currently permit arithmetic and logical operations on enumeration types can provide a mechanism to ban such operations program-wide.

— Languages that provide automatic defaults or that do not enforce static matching between enumerator definitions and initialization expressions can provide a mechanism to enforce such matching.

6.6 Conversion errors [FLC]

6.6.1 Description of application vulnerability

Certain contexts in various languages may require exact matches with respect to types.

```
aVar := anExpression value1 + value2 foo(arg1, arg2, arg3, … , argN)
```

Type conversion seeks to follow these exact match rules while allowing programmers some flexibility in using values such as: structurally equivalent types in a name-equivalent language, types whose value ranges can be distinct but intersect (for example, subranges), and distinct types with sensible/meaningful corresponding values (for example, integers and floats).

Conversions can lead to a loss of data, if the target representation is not capable of representing the original value. For example, converting from an integer type to a smaller integer type can result in

© ISO/IEC 2019 – All rights reserved

AVA Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

truncation if the original value cannot be represented in the smaller size and converting a floating point to an integer can result in a loss of precision or an out-of-range value. Converting from a character type to a smaller character type can result in the misrepresentation of the character.

Type-conversion errors can lead to erroneous data being generated, algorithms that fail to terminate, array bounds-errors, or arbitrary program execution.

See also 6.44 Polymorphic variables [BKK] for up-casting errors.

6.6.2 Cross reference

CWE[8]: 192. Integer Coercion Error

MISRA C[35]: 7.2, 10.1, 10.3, 10.4, 10.6-10.8, and 11.1-11.8

MISRA C++[36]: 2-13-3, 5-0-3, 5-0-4, 5-0-5, 5-0-6, 5-0-7, 5-0-8, 5-0-9, 5-0-10, 5-2-5, 5-2-9, and 5-3-2

CERT C guidelines[38]: FLP34-C, INT02-C, INT08-C, INT31-C, and INT35-C

6.6.3 Mechanism of failure

Conversion errors result in data integrity issues, which can result in a number of safety and security vulnerabilities.

When the conversion results in no change in representation but a change in value for the new type, this may result in a value that is not expressible in the new type, or that has a dramatically different order or meaning. One such situation is the change of sign between the origin and destination (negative -> positive or positive -> negative), which changes the relative order of members of the two types and can result in memory access failures if the values are used in address calculations. Numeric type conversions can be less obvious because some languages will silently convert between numeric types.

Vulnerabilities typically occur when appropriate range checking is not performed, and unanticipated values are encountered. Ariane 5[2][33] launcher failure occurred due to an improperly handled conversion error resulting in the processor being shut down.

Conversion errors can also result in security issues. An attacker can input a particular numeric value to exploit a flaw in the program logic. The resulting erroneous value can then be used as an array index, a loop iterator, a length, a size, state data, or in some other security-critical manner. For example, a truncated integer value can be used to allocate memory, while the actual length is used to copy information to the newly allocated memory, resulting in a buffer overflow[30].

Numeric type-conversion errors can lead to undefined states of execution resulting in infinite loops or crashes. In some cases, integer type-conversion errors can lead to exploitable buffer overflow conditions, resulting in the execution of arbitrary code. Integer type-conversion errors result in an incorrect value being stored for the variable in question.

Explicit conversions between entities of different unit systems without the application of the correct conversion factors can lead to incorrect computations. For example, the first Martian lander failed due to an improper conversion from meters to feet resulting in the loss of the lander.

6.6.4 Applicable language characteristics

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

— Languages that perform implicit type conversion (coercion).

— *Languages that permit conversions between subtypes of a polymorphic type.* See 6.44 upcasts and downcasts.

— Weakly typed languages that do not strictly enforce type rules.

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

— Languages that support logical, arithmetic, or circular shifts on integer values.

— Languages that do not generate exceptions on problematic conversions.

6.6.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

— If range checking is not provided by the language, use explicit range checks, type checks or value checks to validate the correctness of all values originating from a source that is not trusted. However, it is difficult to guarantee that multiple input variables cannot be manipulated to cause an error to occur in some operation somewhere in a program[30].

— Alternatively, use explicit range checks to protect each operation. Because of the large number of integer operations that are susceptible to these problems and the number of checks required to prevent or detect exceptional conditions, this approach can be prohibitively labour intensive and expensive to implement.

— Choose a language that generates exceptions on erroneous data conversions.

— Design objects and program flow such that multiple or complex explicit type conversions are unnecessary. Understand any explicit type conversion needed to reduce the plausibility of error in use.

— Use static analysis tools to identify whether or not unacceptable conversions will occur, to the extent possible.

— Avoid the use of *plausible but wrong* default values when a calculation cannot be completed correctly. Either generate an error or produce a value that is out of range and is certain to be detected. Take care that any error processing does not lead to a denial-of-service vulnerability.

— Always respect the implied unit systems, when converting explicitly from one numeric type to another.

6.6.6 Implications for language design and evolution

In future language design and evolution activities, the following items should be considered.

— Languages should provide mechanisms to prevent programming errors due to conversions.

— Languages should consider making all type-conversions explicit or at least generating warnings for implicit conversions where loss of data may occur.

6.7 String termination [CJM]

6.7.1 Description of application vulnerability

Some programming languages use a termination character to indicate the end of a string. Relying on the occurrence of the string termination character without verification can lead to either exploitation or unexpected behaviour.

6.7.2 Cross reference

CWE[8]:

170. Improper Null Termination

CERT C guidelines[38]: STR03-C, STR31-C, STR32-C, and STR36-C

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

6.7.3 Mechanism of failure

String termination errors occur when the termination character is solely relied on to stop processing on the string and the termination character is not present. Continued processing on the string can cause an error or potentially be exploited as a buffer overflow. This can occur as a result of a programmer making an assumption that a string that is passed as input or generated by a library contains a string termination character when it does not.

Programmers can forget to allocate space for the string termination character and expect to be able to store an *n*-length character string in an array that is *n* characters long. Doing so can work in some instances depending on what is stored after the array in memory, but it can fail or be exploited at some point.

6.7.4 Applicable language characteristics

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

— Languages that use a termination character to indicate the end of a string.

— Languages that do not do bounds checking when accessing a string or array.

6.7.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

— Do not rely solely on the string termination character.

— Use library calls that do not rely on string termination characters such as `strncpy` instead of `strcpy` in the standard C library.

— Use static analysis tools that detect errors in string termination.

6.7.6 Implications for language design and evolution

In future language design and evolution activities, the following items should be considered.

— Eliminating library calls that make assumptions about string termination characters.

— Checking bounds when an array or string is accessed, C Bounds Checking Library[28].

— Specifying a string construct that does not need a string termination character.

6.8 Buffer boundary violation (buffer overflow) [HCB]

6.8.1 Description of application vulnerability

A buffer boundary violation arises when, due to unchecked array indexing or unchecked array copying, storage outside the buffer is accessed. Usually boundary violations describe the situation where such storage is then written. Depending on where the buffer is located, logically unrelated portions of the stack or the heap can be modified maliciously or unintentionally. Usually, buffer boundary violations are accesses to contiguous memory beyond either end of the buffer data, accessing before the beginning or beyond the end of the buffer data is equally possible, dangerous and maliciously exploitable.

6.8.2 Cross reference

CWE[8]:

120. Buffer copy without Checking Size of Input (‘Classic Buffer Overflow’)

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

122. Heap-based Buffer Overflow

124. Boundary Beginning Violation ('Buffer Underwrite')

129. Unchecked Array Indexing

131. Incorrect Calculation of Buffer Size

787. Out-of-bounds Write

805. Buffer Access with Incorrect Length Value

JSF AV Rule: 15 and 25

MISRA C[35]: 21.1

MISRA C++[36]: 5-0-15 to 5-0-18

CERT C guidelines[38]: ARR30-C, ARR32-C, ARR33-C, ARR38-C, MEM35-C and STR31-C

6.8.3 Mechanism of failure

The program statements that cause buffer boundary violations are often difficult to find.

There are several kinds of failures (in all cases an exception may be raised if the accessed location is outside of some permitted range of the run-time environment).

— A read access will return a value that has no relationship to the intended value, such as, the value of another variable or uninitialized storage.

— An out-of-bounds read access is used to obtain information that is intended to be confidential.

— A write access will not result in the intended value being updated and may result in the value of an unrelated object (that happens to exist at the given storage location) being modified, including the possibility of changes in external devices resulting from the memory location being hardware-mapped.

— When an array has been allocated storage on the stack, an out-of-bounds write access can modify internal runtime housekeeping information (for example, a function's return address) which can change a program's control flow.

— An inadvertent or malicious overwrite of function pointers that is in memory, causing them to point to an unexpected location or the attacker's code. Even in applications that do not explicitly use function pointers, the run-time will usually store pointers to functions in memory. For example, object methods in object-oriented languages are generally implemented using function pointers in a data structure or structures that are kept in memory. The consequence of a buffer boundary violation can be targeted to cause arbitrary code execution; this vulnerability can be used to subvert any security service.

6.8.4 Applicable language characteristics

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

— Languages that do not detect and prevent an array being accessed outside of its declared bounds (either by means of an index or by pointer[1)]).

— Languages that do not automatically allocate storage when accessing an array element for which storage has not already been allocated.

1) Using the physical memory address to access the memory location.

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

— Languages that provide bounds checking but permit the check to be suppressed.

— Languages that allow a copy or move operation without an automatic length check ensuring that source and target locations are of at least the same size. The destination target can be larger than the source being copied.

6.8.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

— Use of implementation-provided functionality to automatically check array element accesses and prevent out-of-bounds accesses.

— Use of static analysis to verify that all array accesses are within the permitted bounds. Such analysis may require that source code contain certain kinds of information, such as, that the bounds of all declared arrays be explicitly specified, or that pre- and post-conditions be specified.

— Perform sanity checks on all calculated expressions used as an array index or for pointer arithmetic.

— Ascertain whether or not the compiler can insert bounds checks while still meeting the performance requirements of the program and direct the compiler to insert such checks where appropriate

Some guideline documents recommend only using variables having an unsigned data type when indexing an array, on the basis that an unsigned data type can never be negative. This recommendation simply converts an indexing underflow to an indexing overflow because the value of the variable will wrap to a large positive value rather than a negative one. Also, some languages support arrays whose lower bound is greater than zero, so an index can be positive and be less than the lower bound. Some languages support zero-sized arrays, so any reference to a location within such an array is invalid.

In the past, the implementation of array bound checking has sometimes incurred what has been considered to be a high runtime overhead (often because unnecessary checks were performed). It is now practical for translators to perform sophisticated analysis that significantly reduces the runtime overhead (because runtime checks are only made when it cannot be shown statically that no bound violations can occur).

6.8.6 Implications for language design and evolution

In future language design and evolution activities, the following items should be considered.

— Languages should provide safe copying of arrays as built-in operation.

— Languages should consider only providing array copy routines in libraries that perform checks on the parameters to ensure that no buffer overrun can occur.

— Languages should perform automatic bounds checking on accesses to array elements, unless the compiler can statically determine that the check is unnecessary. This capability may be optional for performance reasons.

— Languages that use pointer types should consider specifying a standardized feature for a pointer type that would enable array bounds checking.

6.9 Unchecked array indexing [XYZ]

6.9.1 Description of application vulnerability

Unchecked array indexing occurs when a value is used as an index into an array without checking that it falls within the acceptable index range.

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

6.9.2 Cross reference

CWE[8]:

129. Unchecked Array Indexing

676. Use of Potentially Dangerous Function

JSF AV Rules[31]: 164 and 15

MISRA C[35]: 21.1

MISRA C++[36]: 5-0-15 to 5-0-18

CERT C guidelines[38]: ARR30-C, ARR32-C, ARR33-C, and ARR38-C

Ada Quality and Style Guide[1]: 5.5.1, 5.5.2, 7.6.7, and 7.6.8

6.9.3 Mechanism of failure

A single fault can allow both an overflow and underflow of the array index. An index overflow exploit can use buffer overflow techniques, but this can often be exploited without having to provide "large inputs." Array index overflows can also trigger out-of-bounds read operations, or operations on the wrong objects; that is, buffer overflows are not always the result. Unchecked array indexing, depending on its instantiation, can be responsible for any number of related issues. Most prominent of these possible flaws is the buffer overflow condition, with consequences ranging from denial of service, and data corruption, to arbitrary code execution. The most common situation leading to unchecked array indexing is the use of loop index variables as buffer indexes. If the end condition for the loop is subject to a flaw, the index can grow or shrink unbounded, therefore causing a buffer overflow or underflow. Another common situation leading to this condition is the use of a function's return value, or the resulting value of a calculation directly as an index in to a buffer. Unchecked array indexing can result in the corruption of relevant memory and perhaps instructions, lead to the program halting, if the values are outside of the valid memory area. If the memory corrupted is data, rather than instructions, the system can continue to function with improper values. If the corrupted memory can be effectively controlled, it may be possible to execute arbitrary code, as with a standard buffer overflow.

Language implementations may or may not statically detect out of bound access and generate a compile-time diagnostic. At runtime, the implementation may or may not detect the out-of-bound access and provide a notification. The notification can be treatable by the program or it may not be. Accesses can violate the bounds of the entire array or violate the bounds of a particular index. It is possible that the former is checked and detected by the implementation while the latter is not. The information needed to detect the violation may or may not be available depending on the context of use. (For example, passing an array to a subroutine via a pointer can deprive the subroutine of information regarding the size of the array.)

Aside from bounds checking, some languages have ways of protecting against out-of-bounds accesses. Some languages automatically extend the bounds of an array to accommodate accesses that can otherwise have been beyond the bounds. However, this may or may not match the programmer's intent and can mask errors. Some languages provide for whole array operations that can obviate the need to access individual elements thus preventing unchecked array accesses.

6.9.4 Applicable language characteristics

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

— Languages that do not automatically bounds check array accesses.

— Languages that do not automatically extend the bounds of an array to accommodate array accesses.

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

6.9.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

— Include sanity checks to ensure the validity of any values used as index variables.

— Consider choosing a language that is not susceptible to these issues.

— When available, use whole array operations whenever possible.

— Do not suppress bounds checks if provided by the language.

6.9.6 Implications for language designers

In future language design and evolution, the following items should be considered.

— Provide compiler switches or other tools to check the size and bounds of arrays and their extents that are statically determinable.

— Providing whole array operations that can obviate the need to access individual elements.

— Languages should consider the capability to generate exceptions or automatically extend the bounds of an array to accommodate accesses that can otherwise have been beyond the bounds.

6.10 Unchecked array copying [XYW]

6.10.1 Description of application vulnerability

When the size and addresses of both the source and destination of an array or compound object are not checked before the copy operation begins, the results can be catastrophic to program integrity. The classic case of buffer overflow happens when some number of bytes (or other units of storage) are copied from one buffer to another and the amount being copied is greater than is allocated for the destination buffer. Data corruption can happen when the program, or the programmer, does not check for overlap between the source and target.

The first situation, overflow of a buffer in a sensitive region of a system, has been exploited as a classic attack vector to render systems inoperable or to take them over.

The second situation, that of overlap, can result in data corruption, which is likely to result in incorrect functioning of the system with potentially disastrous consequences to the containing system.

6.10.2 Cross reference

CWE[8]:

121. Stack-based Buffer Overflow

JSF AV Rule: 15

MISRA C[35]: 21.1

MISRA C++[36]: 5-0-15 to 5-0-18

CERT C guidelines[38]: ARR33-C and STR31-C

Ada Quality and Style Guide[1]: 7.6.7 and 7.6.8

6.10.3 Mechanism of failure

Many languages and some third-party libraries provide functions that efficiently copy the contents of one area of storage to another area of storage. Most of these libraries do not perform any checks

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

to ensure that the copied from/to storage area is large enough to accommodate the amount of data being copied.

When the source and target areas overlap, some libraries do not produce the expected outcome of copying the value of the source area into the target area, because they do not identify the situation and save into a temporary first to isolate the overlapped ranges.

The arguments to these library functions include the addresses of the contents of the two storage areas and the number of bytes (or some other measure) to copy. Passing the appropriate combination of incorrect start addresses or number of bytes to copy makes it possible to read or write outside of the storage allocated to the source/destination area. When passed incorrect parameters the library function performs one or more unchecked array index accesses, as described in 6.9.

6.10.4 Applicable language characteristics

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

— Languages that contain standard library functions for performing bulk copying of storage areas.

— The same range of languages having the characteristics listed in 6.9.

6.10.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

— Only use library functions that perform checks on the arguments to ensure no buffer overrun can occur (perhaps by writing a wrapper for the standard provided functions). Perform checks on the argument expressions prior to calling the Standard library function to ensure that no buffer overrun will occur.

— Use static analysis to verify that the appropriate library functions are only called with arguments that do not result in a buffer overrun or overlap. Such analysis may require that source code contain certain kinds of information, for example, that the bounds of all declared arrays be explicitly specified, or that pre- and post-conditions be specified as annotations or language constructs.

— Sanitize input data so that excessively large input data that can result in overflows is rejected.

— Do not suppress bounds checks if provided by the language.

6.10.6 Implications for language design and evolution

In future language design and evolution activities, the following items should be considered.

— Languages should consider only providing libraries that perform checks on the parameters to ensure that no buffer overrun can occur.

— Languages should consider providing full array assignment.

6.11 Pointer type conversions [HFC]

6.11.1 Description of application vulnerability

The code produced for access via a data or function pointer requires that the type of the pointer is appropriate for the data or function being accessed. Otherwise undefined behaviour can occur. Specifically, *access via a data pointer* is defined to be “fetch or store indirectly through that pointer” and *access via a function pointer* is defined to be “invocation indirectly through that pointer.” The detailed requirements for the meaning of *appropriate type* can vary among languages.

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

Even if the type of the pointer is appropriate for the access, erroneous pointer operations can still cause a fault.

6.11.2 Cross reference

CWE[8]:

136. Type Errors

188. Reliance on Data/Memory Layout

JSF AV Rules[31]: 182 and 183

MISRA C[35]: 11.1-11.8

MISRA C++[36]: 5-2-2 to 5-2-9

CERT C guidelines[38]: INT11-C and EXP36-A

Hatton[15] rule 13: Pointer casts

Ada Quality and Style Guide[1]: 7.6.7 and 7.6.8

6.11.3 Mechanism of failure

If a pointer's type is not appropriate for the data or function being accessed, data can be corrupted or privacy can be broken by inappropriate read or write operation using the indirection provided by the pointer value. With a suitable type definition, large portions of memory can be maliciously or accidentally modified or read. Such modification of data objects will generally lead to value faults of the application. Modification of code elements such as function pointers or internal data structures for the support of object-orientation can affect control flow. This can make the code susceptible to targeted attacks by causing invocation via a pointer-to-function that has been manipulated to point to an attacker's malicious code.

6.11.4 Applicable language characteristics

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

— Pointers (and/or references) can be converted to different pointer (and/or reference) types.

— Pointers to functions can be converted to pointers to data.

6.11.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

— Treat the compiler's pointer-conversion warnings as serious errors.

— Adopt programming guidelines (preferably augmented by static analysis) that restrict pointer conversions. For example, consider the rules itemized above from JSF AV[31], CERT C[38], Hatton[15], or MISRA C[35].

— Use other means of assurance such as proofs of correctness, analysis with tools, verification techniques, or other methods to check that pointer conversions do not lead to later undefined behaviour.

6.11.6 Implications for language design and evolution

In future language design and evolution activities, languages should consider creating a mode that provides a runtime check of the validity of all accessed objects before the object is read, written or executed.

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

6.12 Pointer arithmetic [RVG]

6.12.1 Description of application vulnerability

Using pointer arithmetic incorrectly can result in addressing arbitrary locations, which in turn can cause a program to behave in unexpected ways.

6.12.2 Cross reference

JSF AV[31] Rule: 215

MISRA C[35]: 18.1-18.4

MISRA C++[36]: 5-0-15 to 5-0-18

CERT C guidelines[38]: EXP08-C

6.12.3 Mechanism of failure

Pointer arithmetic used incorrectly can produce:

— Addressing arbitrary memory locations, including buffer underflow and overflow.

— Arbitrary code execution.

— Addressing memory outside the range of the program.

6.12.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that allow pointer arithmetic.

6.12.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

— Avoid using pointer arithmetic for accessing anything except composite types.

— Prefer indexing for accessing array elements rather than using pointer arithmetic.

— Limit pointer arithmetic calculations to the addition and subtraction of integers.

6.12.6 Implications for language design and evolution

[None]

6.13 Null pointer dereference [XYH]

6.13.1 Description of application vulnerability

A null-pointer dereference takes place when a pointer with a value of `NULL` is used as though it pointed to a valid memory location. This is a special case of accessing storage via an invalid pointer.

6.13.2 Cross reference

CWE[8]:

476. NULL Pointer Dereference

JSF AV Rule 174

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

CERT C guidelines[38]: EXP34-C

Ada Quality and Style Guide[1]: 5.4.5

6.13.3 Mechanism of failure

When a pointer with a value of `NULL` is used as though it pointed to a valid memory location, then a null-pointer dereference is said to take place. This can result in a segmentation fault, unhandled exception, or accessing unanticipated memory locations.

6.13.4 Applicable language characteristics

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

— Languages that permit the use of pointers and that do not check the validity of the location being accessed prior to the access.

— Languages that allow the use of a `NULL` pointer.

6.13.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects by ensuring that prior to dereferencing a pointer, its value is not equal to NULL.

6.13.6 Implications for language design and evolution

In future language design and evolution activities, consider a language feature that would check a pointer value for NULL before performing an access.

6.14 Dangling reference to heap [XYK]

6.14.1 Description of application vulnerability

A dangling reference is a reference to an object whose lifetime has ended due to explicit deallocation or the stack frame in which the object resided has been freed due to exiting the dynamic scope. The memory for the object may be reused; therefore, any access through the dangling reference can affect an apparently arbitrary location of memory, corrupting data or code.

This description concerns the former case, dangling references to the heap. The description of dangling references to stack frames can be found in 6.33. In many languages, references are called pointers; the issues are identical.

A notable special case of using a dangling reference is calling a deallocator, for example, `free()`, twice on the same pointer value. Such a *double free* can corrupt internal data structures of the heap administration, leading to faulty application behaviour (such as infinite loops within the allocator, returning the same memory repeatedly as the result of distinct subsequent allocations, or deallocating memory legitimately allocated to another request since the first `free()` call, to name but a few), or no adverse effects at all.

Memory corruption through the use of a dangling reference is among the most difficult errors to locate.

With sufficient knowledge about the heap management scheme [often provided by the *OS* (Operating System) or run-time system], use of dangling references is an exploitable vulnerability, since the dangling reference provides a method with which to read and modify valid data in the designated memory locations after freed memory has been re-allocated by subsequent allocations.

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

6.14.2 Cross reference

CWE[8]:

415. Double Free (Note that Double Free (415) is a special case of Use After Free (416))

416. Use After Free

MISRA C[35]: 18.1-18.6

MISRA C++[36]: 0-3-1, 7-5-1, 7-5-2, 7-5-3, and 18-4-1

CERT C guidelines[38]: MEM01-C, MEM30-C, and MEM31.C

Ada Quality and Style Guide[1]: 5.4.5, 7.3.3, and 7.6.6

6.14.3 Mechanism of failure

The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists and retains its last-stored value throughout its lifetime. If an object is referred to outside of its lifetime, the behaviour is undefined. Explicit deallocation of heap-allocated storage ends the lifetime of the object residing at this memory location (as does leaving the dynamic scope of a declared variable). The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime. Such pointers are called dangling references.

The use of dangling references to previously freed memory can have any number of adverse consequences — ranging from the corruption of valid data to the execution of arbitrary code, depending on the instantiation and timing of the deallocation causing all remaining copies of the reference to become dangling, of the system's reuse of the freed memory, and of the subsequent usage of a dangling reference.

Like memory leaks and errors due to double de-allocation, the use of dangling references has two common and sometimes overlapping causes:

— An error condition or other exceptional circumstances that unexpectedly cause an object to become undefined.

— Developer confusion over which part of the program is responsible for freeing the memory.

If a pointer to previously freed memory is used, it is possible that the referenced memory has been reallocated. Therefore, assignment using the original pointer has the effect of changing the value of an unrelated variable. This induces unexpected behaviour in the affected program. If the newly allocated data happens to hold a class description, in an object-oriented language for example, various function pointers may be scattered within the heap data. If one of these function pointers is overwritten with an address of malicious code, execution of arbitrary code can be achieved.

6.14.4 Applicable language characteristics

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

— Languages that permit the use of pointers and that permit explicit deallocation by the developer or provide for alternative means to reallocate memory still pointed to by some pointer value.

— Languages that permit definitions of constructs that can be parameterized without enforcing the consistency of the use of parameter at compile time.

© ISO/IEC 2019 – All rights reserved

Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019

6.14.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

— Use an implementation that checks whether a pointer is used that designates a memory location that has already been freed.

— Use a coding style that does not permit deallocation.

— In complicated error conditions, be sure that clean-up routines respect the state of allocation properly. If the language is object-oriented, ensure that object destructors delete each chunk of memory only once. Ensuring that all pointers are set to `NULL` once the memory they point to have been freed can be an effective strategy. The utilization of multiple or complex data structures can lower the usefulness of this strategy.

— Use a static analysis tool that is capable of detecting some situations when a pointer is used after the storage it refers to is no longer a pointer to valid memory location.

— Memory should be allocated and freed at the same level of abstraction, and ideally in the same code module[2)].

6.14.6 Implications for language design and evolution

In future language design and evolution activities, the following items should be considered.

— Implementations of the free function may tolerate multiple frees on the same reference/pointer or frees of memory that was never allocated. Such an operation is called an idempotent operation.

— For properties that cannot be checked at compile time, language specifiers should provide an assertion mechanism for checking properties at run-time. It should be possible to inhibit assertion checking if efficiency is a concern.

— A storage allocation interface should be provided that will allow the called function to set the pointer used to NULL after the referenced storage is deallocated.

6.15 Arithmetic wrap-around error [FIF]

6.15.1 Description of application vulnerability

Wrap-around errors can occur whenever a value is incremented past the maximum or decremented past the minimum value representable in its type and, depending on whether:

— the type is signed or unsigned,

— the specification of the language semantics and/or implementation choices,

— the computation *wraps around* to an unexpected value.

This vulnerability is related to 6.16 [3)].

6.15.2 Cross reference

CWE[8]:

128. Wrap-around Error

2) Allocating and freeing memory in different modules and levels of abstraction burdens the programmer with tracking the lifetime of that block of memory. This can cause confusion regarding when and if a block of memory has been allocated or freed, leading to programming defects such as double-free vulnerabilities, accessing freed memory, or dereferencing NULL pointers or pointers that are not initialized

3) This description is derived from Wrap-Around Error [XYY], which appeared in ISO/IEC TR 24772:2008.

© ISO/IEC 2019 – All rights reserved
Documents
Document Sharing Hub
Full document details: PD ISO/IEC TR 24772-1:2019