



International
Standard

ISO/IEC 24772-1

**Programming languages — Avoiding
vulnerabilities in programming
languages —**

**Part 1:
Language-independent catalogue of
vulnerabilities**

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

Partie 1: Catalogue de vulnérabilités indépendant du langage

**First edition
2024-10**

Reference number
ISO/IEC 24772-1:2024(en)

Full document details: ISO IEC 24772-1-2024

© ISO/IEC 2024



Documents
Document Sharing Hub



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2024

All rights reserved. Unless otherwise specified, or required in the context of its implementation, 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
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword	xv
Introduction	xvii
1 Scope	1
2 Normative references	1
3 Terms and definitions	1
3.1 Communication	1
3.2 Execution model	1
3.3 Properties	2
3.4 Safety and security	3
3.5 Vulnerabilities	3
3.6 Specific vulnerabilities	3
4 Using this document	4
4.1 Purpose of this document	4
4.2 Applying this document	5
4.3 Structure of this document	6
5 General vulnerability issues and primary avoidance mechanisms	7
5.1 General vulnerability issues	7
5.1.1 Predictable execution	7
5.1.2 Sources of unpredictability in language specification	8
5.1.3 Sources of unpredictability in language usage	9
5.2 Primary avoidance mechanisms	9
6 Programming language vulnerabilities	11
6.1 General	11
6.2 Type system [IHN]	12
6.2.1 Description of application vulnerability	12
6.2.2 Related coding guidelines	12
6.2.3 Mechanism of failure	12
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	14
6.3 Bit representations [STR]	14
6.3.1 Description of application vulnerability	14
6.3.2 Related coding guidelines	14
6.3.3 Mechanism of failure	15
6.3.4 Applicable language characteristics	15
6.3.5 Avoiding the vulnerability or mitigating its effects	15
6.3.6 Implications for language design and evolution	16
6.4 Floating-point arithmetic [PLF]	16
6.4.1 Description of application vulnerability	16
6.4.2 Related coding guidelines	16
6.4.3 Mechanism of failure	16
6.4.4 Applicable language characteristics	17
6.4.5 Avoiding the vulnerability or mitigating its effects	17
6.4.6 Implications for language design and evolution	18
6.5 Enumerator issues [CCB]	18
6.5.1 Description of application vulnerability	18
6.5.2 Related coding guidelines	19
6.5.3 Mechanism of failure	19
6.5.4 Applicable language Characteristics	19
6.5.5 Avoiding the vulnerability or mitigating its effects	20
6.5.6 Implications for language design and evolution	20
6.6 Conversion errors [FLC]	20
6.6.1 Description of application vulnerability	20

6.6.2	Related coding guidelines	20
6.6.3	Mechanism of failure	21
6.6.4	Applicable language characteristics.....	21
6.6.5	Avoiding the vulnerability or mitigating its effects.....	21
6.6.6	Implications for language design and evolution	22
6.7	String termination [CJM]	22
6.7.1	Description of application vulnerability	22
6.7.2	Related coding guidelines	22
6.7.3	Mechanism of failure	22
6.7.4	Applicable language characteristics.....	22
6.7.5	Avoiding the vulnerability or mitigating its effects.....	23
6.7.6	Implications for language design and evolution	23
6.8	Buffer boundary violation (buffer overflow) [HCB]	23
6.8.1	Description of application vulnerability	23
6.8.2	Related coding guidelines	23
6.8.3	Mechanism of failure	24
6.8.4	Applicable language characteristics.....	24
6.8.5	Avoiding the vulnerability or mitigating its effects.....	24
6.8.6	Implications for language design and evolution	25
6.9	Unchecked array indexing [XYZ]	25
6.9.1	Description of application vulnerability	25
6.9.2	Related coding guidelines	25
6.9.3	Mechanism of failure	25
6.9.4	Applicable language characteristics.....	26
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].....	27
6.10.1	Description of application vulnerability	27
6.10.2	Related coding guidelines	27
6.10.3	Mechanism of failure	27
6.10.4	Applicable language characteristics.....	27
6.10.5	Avoiding the vulnerability or mitigating its effects.....	28
6.10.6	Implications for language design and evolution	28
6.11	Pointer type conversions [HFC]	28
6.11.1	Description of application vulnerability	28
6.11.2	Related coding guidelines	28
6.11.3	Mechanism of failure	29
6.11.4	Applicable language characteristics.....	29
6.11.5	Avoiding the vulnerability or mitigating its effects.....	29
6.11.6	Implications for language design and evolution	29
6.12	Pointer arithmetic [RVG]	29
6.12.1	Description of application vulnerability	29
6.12.2	Related coding guidelines	29
6.12.3	Mechanism of failure	29
6.12.4	Applicable language characteristics.....	30
6.12.5	Avoiding the vulnerability or mitigating its effects.....	30
6.12.6	Implications for language design and evolution	30
6.13	Null pointer dereference [XYH]	30
6.13.1	Description of application vulnerability	30
6.13.2	Related coding guidelines	30
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	31
6.14	Dangling reference to heap [XYK]	31
6.14.1	Description of application vulnerability	31
6.14.2	Related coding guidelines	31
6.14.3	Mechanism of failure	31
6.14.4	Applicable language characteristics.....	32

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]	33
6.15.1	Description of application vulnerability	33
6.15.2	Related coding guidelines	33
6.15.3	Mechanism of failure	33
6.15.4	Applicable language characteristics	34
6.15.5	Avoiding the vulnerability or mitigating its effects	34
6.15.6	Implications for language design and evolution	34
6.16	Using shift operations for multiplication and division [PIK]	34
6.16.1	Description of application vulnerability	34
6.16.2	Related coding guidelines	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	35
6.16.6	Implications for language design and evolution	35
6.17	Choice of clear names [NAI]	35
6.17.1	Description of application vulnerability	35
6.17.2	Related coding guidelines	36
6.17.3	Mechanism of Failure	36
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	37
6.18	Dead store [WXQ]	37
6.18.1	Description of application vulnerability	37
6.18.2	Related coding guidelines	37
6.18.3	Mechanism of failure	37
6.18.4	Applicable language characteristics	37
6.18.5	Avoiding the vulnerability or mitigating its effects	38
6.18.6	Implications for language design and evolution	38
6.19	Unused variable [YZS]	38
6.19.1	Description of application vulnerability	38
6.19.2	Related coding guidelines	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	39
6.20	Identifier name reuse [YOW]	39
6.20.1	Description of application vulnerability	39
6.20.2	Related coding guidelines	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]	41
6.21.1	Description of application vulnerability	41
6.21.2	Related coding guidelines	41
6.21.3	Mechanism of Failure	41
6.21.4	Applicable language characteristics	41
6.21.5	Avoiding the Vulnerability or Mitigating its Effects	42
6.21.6	Implications for language design and evolution	42
6.22	Missing initialization of variables [LAV]	42
6.22.1	Description of application vulnerability	42
6.22.2	Related coding guidelines	42
6.22.3	Mechanism of failure	43
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	Related coding guidelines	44
6.23.3	Mechanism of failure	45
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	Related coding guidelines	46
6.24.3	Mechanism of failure	46
6.24.4	Applicable language characteristics	47
6.24.5	Avoiding the vulnerability or mitigating its effects	47
6.24.6	Implications for language design and evolution	47
6.25	Likely incorrect expression [KOA]	47
6.25.1	Description of application vulnerability	47
6.25.2	Related coding guidelines	47
6.25.3	Mechanism of failure	48
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]	49
6.26.1	Description of application vulnerability	49
6.26.2	Related coding guidelines	49
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 lack of static analysis [CLL]	51
6.27.1	Description of application vulnerability	51
6.27.2	Related coding guidelines	51
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	52
6.28	Non-demarcation of control flow [EOJ]	52
6.28.1	Description of application vulnerability	52
6.28.2	Related coding guidelines	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 variable abuse [TEX]	53
6.29.1	Description of application vulnerability	53
6.29.2	Related coding guidelines	53
6.29.3	Mechanism of failure	54
6.29.4	Applicable language characteristics	54
6.29.5	Avoiding the vulnerability or mitigating its effects	54
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	Related coding guidelines	55
6.30.3	Mechanism of failure	55
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]	56
6.31.1	Description of application vulnerability	56
6.31.2	Related coding guidelines	56
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.....	57
6.32	Passing parameters and return values [CSJ].....	57
6.32.1	Description of application vulnerability.....	57
6.32.2	Related coding guidelines.....	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.....	59
6.33	Dangling references to stack frames [DCM].....	59
6.33.1	Description of application vulnerability.....	59
6.33.2	Related coding guidelines.....	59
6.33.3	Mechanism of failure.....	59
6.33.4	Applicable language characteristics.....	60
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].....	61
6.34.1	Description of application vulnerability.....	61
6.34.2	Related coding guidelines.....	61
6.34.3	Mechanism of failure.....	61
6.34.4	Applicable language characteristics.....	61
6.34.5	Avoiding the vulnerability or mitigating its effects.....	62
6.34.6	Implications for language design and evolution.....	62
6.35	Recursion [GDL].....	62
6.35.1	Description of application vulnerability.....	62
6.35.2	Related coding guidelines.....	62
6.35.3	Mechanism of failure.....	62
6.35.4	Applicable language characteristics.....	63
6.35.5	Avoiding the vulnerability or mitigating its effects.....	63
6.35.6	Implications for language design and evolution.....	63
6.36	Ignored error status and unhandled exceptions [OYB].....	63
6.36.1	Description of application vulnerability.....	63
6.36.2	Related coding guidelines.....	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	Related coding guidelines.....	65
6.37.3	Mechanism of failure.....	66
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.....	67
6.38	Deep vs. shallow copying [YAN].....	67
6.38.1	Description of application vulnerability.....	67
6.38.2	Related coding guidelines.....	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.....	68
6.38.6	Implications for language design and evolution.....	68
6.39	Memory leaks and heap fragmentation [XYL].....	68
6.39.1	Description of application vulnerability.....	68
6.39.2	Related coding guidelines.....	68
6.39.3	Mechanism of failure.....	68
6.39.4	Applicable language characteristics.....	69
6.39.5	Avoiding the vulnerability or mitigating its effects.....	69
6.39.6	Implications for language design and evolution.....	69

6.40	Templates and generics [SYM]	70
6.40.1	Description of application vulnerability	70
6.40.2	Related coding guidelines	70
6.40.3	Mechanism of failure	70
6.40.4	Applicable language characteristics	71
6.40.5	Avoiding the vulnerability or mitigating its effects	71
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	Related coding guidelines	71
6.41.3	Mechanism of failure	72
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	Related coding guidelines	73
6.42.3	Mechanism of failure	74
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	Related coding guidelines	75
6.43.3	Mechanism of failure	75
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	Related coding guidelines	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	77
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	Related coding guidelines	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	78
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	Related coding guidelines	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	79
6.46.6	Implications for language design and evolution	79
6.47	Inter-language calling [DJS]	79
6.47.1	Description of application vulnerability	79
6.47.2	Related coding guidelines	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	81
6.48	Dynamically-linked code and self-modifying code [NYY]	81
6.48.1	Description of application vulnerability	81
6.48.2	Related coding guidelines	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.....	82
6.48.6	Implications for language design and evolution	82
6.49	Library signature [NSQ].....	82
6.49.1	Description of application vulnerability.....	82
6.49.2	Related coding guidelines	82
6.49.3	Mechanism of failure	82
6.49.4	Applicable language characteristics.....	83
6.49.5	Avoiding the vulnerability or mitigating its effects.....	83
6.49.6	Implications for language design and evolution	83
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	Related coding guidelines	83
6.50.4	Applicable language characteristics.....	83
6.50.5	Avoiding the vulnerability or mitigating its effects.....	84
6.50.6	Implications for language design and evolution	84
6.51	Pre-processor directives [NMP]	84
6.51.1	Description of application vulnerability.....	84
6.51.2	Related coding guidelines	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	Related coding guidelines	86
6.52.3	Mechanism of Failure.....	86
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	Related coding guidelines	87
6.53.3	Mechanism of Failure.....	87
6.53.4	Applicable language characteristics.....	87
6.53.5	Avoiding the vulnerability or mitigating its effect.....	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	Related coding guidelines	88
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	89
6.55	Unspecified behaviour [BQF]	89
6.55.1	Description of application vulnerability.....	89
6.55.2	Related coding guidelines	89
6.55.3	Mechanism of failure	89
6.55.4	Applicable language characteristics.....	90
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	Related coding guidelines	90
6.56.3	Mechanism of failure	91
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	Related coding guidelines	92
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	Related coding guidelines	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	Related coding guidelines	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	Related coding guidelines	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	Related coding guidelines	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	Related coding guidelines	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	Related coding guidelines	101
6.63.3	Mechanism of failure	101
6.63.4	Applicable language characteristics	102
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	Related coding guidelines	103
6.64.3	Mechanism of failure	103
6.64.4	Applicable language characteristics	103
6.64.5	Avoiding the vulnerability or mitigating its effects	104
6.64.6	Implications for language design and evolution	104
6.65	Modifying constants [UJO]	104
6.65.1	Description of application vulnerability	104

6.65.2	Related coding guidelines	104
6.65.3	Mechanism of failure	104
6.65.4	Applicable language characteristics	105
6.65.5	Avoiding the vulnerability or mitigating its effects	105
6.65.6	Implications for language design and evolution	105
7	Application vulnerabilities	105
7.1	General	105
7.2	Unrestricted file upload [CBF]	105
7.2.1	Description of application vulnerability	105
7.2.2	Related coding guidelines	105
7.2.3	Mechanism of failure	106
7.2.4	Avoiding the vulnerability or mitigating its effects	106
7.3	Download of code without integrity check [DLB]	106
7.3.1	Description of application vulnerability	106
7.3.2	Related coding guidelines	107
7.3.3	Mechanism of failure	107
7.3.4	Avoiding the vulnerability or mitigating its effects	107
7.4	Executing or loading untrusted code [XYS]	107
7.4.1	Description of application vulnerability	107
7.4.2	Related coding guidelines	107
7.4.3	Mechanism of failure	107
7.4.4	Avoiding the vulnerability or mitigating its effects	108
7.5	Inclusion of functionality from untrusted control sphere [DHU]	108
7.5.1	Description of application vulnerability	108
7.5.2	Related coding guidelines	108
7.5.3	Mechanism of failure	108
7.5.4	Avoiding the vulnerability or mitigating its effects	108
7.6	Use of unchecked data from an uncontrolled or tainted source [EFS]	109
7.6.1	Description of application vulnerability	109
7.6.2	Related coding guidelines	109
7.6.3	Mechanism of failure	109
7.6.4	Avoiding the vulnerability or mitigating its effects	109
7.7	Cross-site scripting [XYT]	110
7.7.1	Description of application vulnerability	110
7.7.2	Related coding guidelines	110
7.7.3	Mechanism of failure	110
7.7.4	Avoiding the vulnerability or mitigating its effects	111
7.8	URL redirection to untrusted site ("open redirect") [PYQ]	112
7.8.1	Description of application vulnerability	112
7.8.2	Related coding guidelines	112
7.8.3	Mechanism of failure	112
7.8.4	Avoiding the vulnerability or mitigating its effects	112
7.9	Injection [RST]	113
7.9.1	Description of application vulnerability	113
7.9.2	Related coding guidelines	113
7.9.3	Mechanism of failure	114
7.9.4	Avoiding the vulnerability or mitigating its effects	115
7.10	Unquoted search path or element [XZQ]	115
7.10.1	Description of application vulnerability	115
7.10.2	Related coding guidelines	115
7.10.3	Mechanism of failure	116
7.10.4	Avoiding the vulnerability or mitigating its effects	116
7.11	Path traversal [EWR]	116
7.11.1	Description of application vulnerability	116
7.11.2	Related coding guidelines	116
7.11.3	Mechanism of failure	117
7.11.4	Avoiding the vulnerability or mitigating its effects	118
7.12	Resource names [HTS]	118

7.12.1	Description of application vulnerability	118
7.12.2	Related coding guidelines	119
7.12.3	Mechanism of Failure	119
7.12.4	Avoiding the vulnerability or mitigating its effects	119
7.13	Resource exhaustion [XZP]	119
7.13.1	Description of application vulnerability	119
7.13.2	Related coding guidelines	120
7.13.3	Mechanism of failure	120
7.13.4	Avoiding the vulnerability or mitigating its effects	120
7.14	Authentication logic error [XZO]	121
7.14.1	Description of application vulnerability	121
7.14.2	Related coding guidelines	121
7.14.3	Mechanism of failure	121
7.14.4	Avoiding the vulnerability or mitigating its effects	122
7.15	Improper restriction of excessive authentication attempts [WPL]	122
7.15.1	Description of application vulnerability	122
7.15.2	Related coding guidelines	122
7.15.3	Mechanism of failure	122
7.15.4	Avoiding the vulnerability or mitigating its effects	123
7.16	Hard-coded credentials [XYP]	123
7.16.1	Description of application vulnerability	123
7.16.2	Related coding guidelines	123
7.16.3	Mechanism of failure	123
7.16.4	Avoiding the vulnerability or mitigating its effects	124
7.17	Insufficiently protected credentials [XYM]	124
7.17.1	Description of application vulnerability	124
7.17.2	Related coding guidelines	124
7.17.3	Mechanism of failure	124
7.17.4	Avoiding the vulnerability or mitigating its effects	124
7.18	Missing or inconsistent access control [XZN]	125
7.18.1	Description of application vulnerability	125
7.18.2	Related coding guidelines	125
7.18.3	Mechanism of failure	125
7.18.4	Avoiding the vulnerability or mitigating its effects	125
7.19	Incorrect authorization [BJE]	125
7.19.1	Description of application vulnerability	125
7.19.2	Related coding guidelines	125
7.19.3	Mechanism of failure	125
7.19.4	Avoiding the vulnerability or mitigating its effects	126
7.20	Adherence to least privilege [XYN]	126
7.20.1	Description of application vulnerability	126
7.20.2	Related coding guidelines	126
7.20.3	Mechanism of failure	126
7.20.4	Avoiding the vulnerability or mitigating its effects	126
7.21	Privilege sandbox issues [XYO]	127
7.21.1	Description of application vulnerability	127
7.21.2	Related coding guidelines	127
7.21.3	Mechanism of failure	127
7.21.4	Avoiding the vulnerability or mitigating its effects	128
7.22	Missing required cryptographic step [XZS]	128
7.22.1	Description of application vulnerability	128
7.22.2	Related coding guidelines	128
7.22.3	Mechanism of failure	128
7.22.4	Avoiding the vulnerability or mitigating its effects	128
7.23	Improperly verified signature [XZR]	129
7.23.1	Description of application vulnerability	129
7.23.2	Related coding guidelines	129
7.23.3	Mechanism of failure	129
7.23.4	Avoiding the vulnerability or mitigating its effects	129

7.24	Use of a one-way hash without a salt [MVX]	129
7.24.1	Description of application vulnerability	129
7.24.2	Related coding guidelines	129
7.24.3	Mechanism of failure	129
7.24.4	Avoiding the vulnerability or mitigating its effects	129
7.25	Inadequately secure communication of shared resources [CGY]	130
7.25.1	Description of application vulnerability	130
7.25.2	Related coding guidelines	130
7.25.3	Mechanism of failure	130
7.25.4	Avoiding the vulnerability or mitigating its effect	131
7.26	Memory locking [XZX]	131
7.26.1	Description of application vulnerability	131
7.26.2	Related coding guidelines	131
7.26.3	Mechanism of failure	131
7.26.4	Avoiding the vulnerability or mitigating its effects	132
7.27	Sensitive information not cleared before use [XZK]	132
7.27.1	Description of application vulnerability	132
7.27.2	Related coding guidelines	132
7.27.3	Mechanism of failure	132
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	Related coding guidelines	133
7.28.3	Mechanism of failure	133
7.28.4	Avoiding the vulnerability or mitigating its effect	133
7.29	Discrepancy information leak [XZL]	133
7.29.1	Description of application vulnerability	133
7.29.2	Related coding guidelines	134
7.29.3	Mechanism of failure	134
7.29.4	Avoiding the vulnerability or mitigating its effects	134
7.30	Unspecified functionality [BVQ]	134
7.30.1	Description of application vulnerability	134
7.30.2	Related coding guidelines	134
7.30.3	Mechanism of failure	135
7.30.4	Avoiding the vulnerability or mitigating its effects	135
7.31	Fault tolerance and failure strategies [REU]	135
7.31.1	Description of application vulnerability	135
7.31.2	Related coding guidelines	136
7.31.3	Mechanism of failure	136
7.31.4	Avoiding the vulnerability or mitigating its effects	137
7.32	Distinguished values in data types [KLK]	137
7.32.1	Description of application vulnerability	137
7.32.2	Related coding guidelines	137
7.32.3	Mechanism of failure	138
7.32.4	Avoiding the vulnerability or mitigating its effects	138
7.33	Clock issues [CCI]	139
7.33.1	Description of application vulnerability	139
7.33.2	Related coding guidelines	139
7.33.3	Mechanism of failure	139
7.33.4	Avoiding the vulnerability or mitigating its effect	141
7.34	Time drift and jitter [CDJ]	141
7.34.1	Description of application vulnerability	141
7.34.2	Related coding guidelines	142
7.34.3	Mechanism of failure	142
7.34.4	Avoiding the vulnerability or mitigating its effect	142

Annex A (informative) Vulnerability taxonomy and list	143
--	------------

Annex B (informative) Selected principles for language designers	150
---	------------

Bibliography	152
---------------------	------------

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 or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and <https://patents.iec.ch>. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

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

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity 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. In the IEC, see www.iec.ch/understanding-standards.

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*.

This first edition of ISO/IEC 24772-1 cancels and replaces ISO/IEC TR 24772-1:2019, which has been technically revised.

The main changes are as follows:

- the document now describes avoidance mechanisms rather than providing specific guidance, in order to clarify that it is the responsibility of the implementation team to create design and coding standards, and that some of the avoidance mechanisms stated only apply to specific scenarios; "guidance" has been removed from the title accordingly;
- new terms have been added in 3.7 to the terms and definitions clause to address specific vulnerabilities;
- [Clause 4](#) has been expanded to explain how this document is used with programming language standards, safety standards, and security standards;
- [Clause 5](#) has been amended to provide general vulnerability issues and primary avoidance mechanisms;
- the titles of some [Clause 6](#) vulnerabilities have been renamed to better capture the actual vulnerability;
- the clause "Fault tolerance and failure strategies" was moved from [6.37](#) to [7.31](#) to reflect that the vulnerability is more about the system design of fault tolerance and failure recovery strategies than being language-oriented;
- a new language vulnerability "Modifying constants [UJO]" was added in [6.65](#);
- [Clause 7](#) was reorganized to gather similar application vulnerabilities together;
- new application vulnerabilities were added to expose issues with time management in real-time systems, in normal systems and in networked systems;

- a new [Annex B](#) has been added to collate material from the subclauses in [Clause 6](#) entitled “Avoiding the vulnerability or mitigating its effect” in a single place.

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

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 and www.iec.ch/national-committees.

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 catalogues software programming language vulnerabilities to be avoided in the development of systems where assured behaviour is required for security, safety, mission critical or business critical software. In general, this is applicable to the software developed, reviewed, or maintained for any application.

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 in the ISO/IEC 24772 series, such as ISO/IEC 24772-2 for Ada and ISO/IEC 24772-3 for C describe how the language-independent analysis of this document apply to the specific programming language addressed by that particular document.

This document is intended to catalogue avoidance mechanisms 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. These mechanisms can also be used by developers to select source code evaluation tools that can discover and eliminate some constructs that can lead to vulnerabilities in their software or to select a programming language that avoids anticipated problems.

The intended audience for this document consists of parties 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 are required by their organization to avoid language and design 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 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 must ensure properties of confidentiality, integrity, and availability;
- mission-critical applications that must avoid loss or damage to property or finance;
- business-critical applications where correct operation is essential to the successful operation of the business;
- scientific, modeling and simulation applications that require high confidence in the results of possibly complex, expensive and extended calculation.

This document can be relevant to other developers as well. 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. All developers can use this document to ensure that common vulnerabilities are removed or at least minimized from all applications.

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. 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 consider design issues rather than coding issues.

This document is inherently incomplete, as 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.

Programming languages — Avoiding vulnerabilities in programming languages —

Part 1: Language-independent catalogue of vulnerabilities

1 Scope

This document enumerates approaches and techniques to avoid software programming language vulnerabilities in the development of systems where assured behaviour is required for security, safety, mission-critical and business-critical software. In general, the description of the vulnerabilities and description of avoidance mechanisms are applicable to the software developed, reviewed, or maintained for any application.

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

2 Normative references

There are no normative references in this document.

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO and IEC terminology databases and in this clause apply.

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

- ISO Online browsing platform: available at <https://www.iso.org/obp>
- IEC Electropedia: available at <https://www.electropedia.org/>

3.1 Communication

3.1.1 protocol

set of rules and supporting structures for the interaction of concurrent entities, such as tightly embedded interactions of threads or loosely coupled arrangements such as message communication spanning computer systems and networks

3.1.2 stateless protocol

communication or cooperation between threads where no state is preserved in the *protocol* (3.1.1) itself, such as the HTTP protocol or direct access to a shared resource

3.2 Execution model

3.2.1 thread

sequential stream of execution such as a single thread in a process or a process in an operating system

3.2.2**thread activation**

creation and setup of a *thread* (3.2.1) up to the point where the thread begins execution

3.2.3**activated thread**

thread (3.2.1) that is created and then begins execution as a result of the *thread activation* (3.2.1)

3.2.4**activating thread**

thread (3.2.1) that exists first and makes the library calls or contains the language syntax that causes another thread to be activated

3.2.5**static thread activation**

creation and initiation of a *thread* (3.2.1) at program initiation, by an operating system or runtime kernel, or by another thread as part of a declarative part of the thread before it begins execution

3.2.6**dynamic thread activation**

creation and initiation of a *thread* (3.2.1) by another thread (including the main program) as an executable, repeatable command, statement or subprogram call

3.2.7**thread abort**

request to stop and shut down a *thread* (3.2.1) immediately, whether that request comes from an operating system, another thread via the operating system, or a request via shared data or communicating channel to have the thread cease execution

3.2.8**termination directing thread**

thread (3.2.1), including an operating system thread, that requests the termination of one or more threads

3.2.9**thread termination**

completion and orderly shutdown of a *thread* (3.2.1), where the thread is permitted to make data objects consistent, release any acquired resources, and notify any dependent threads that it is terminating

3.2.10**terminated thread**

thread (3.2.1) that has been halted from any further execution

3.2.11**master thread**

thread (3.2.1) that initiates other threads and that eventually waits for one or all *terminated threads* (3.2.10) 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, which is permitted to execute independently, or which can interact in programmed ways with other processes, and which can share resources such as memory, processor and filing system with other processes

3.3 Properties**3.3.1****predictable execution**

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

3.4 Safety and security

3.4.1

safety hazard

potential source of material or environmental damage, physical injury, or damage to the health of people

3.4.2

safety-critical

type of software or application where failure can cause very serious consequences such as human injury or death

3.4.3

salt

randomized value that is additional input to a cryptographic algorithm

3.5 Vulnerabilities

3.5.1

application vulnerability

security vulnerability ([3.5.3](#)) or *safety hazard* ([3.4.1](#)) or defect

3.5.2

language vulnerability

property or feature of a programming language that through its presence or absence can contribute to, or that is strongly correlated with, application vulnerabilities in programs written in that language

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

3.6 Specific vulnerabilities

3.6.1

failure

malfunction of the system or component which has as subcategories: *omission failure* ([3.6.2](#)), *commission failure* ([3.6.3](#)), *timing failure* ([3.6.4](#)) and *value failure* ([3.6.5](#))

3.6.2

omission failure

service that is requested but never rendered

3.6.3

commission failure

service that initiates unexpected actions

3.6.4

timing failure

service that is not rendered before an imposed deadline

3.6.5

value failure

service that delivers incorrect or tainted results

3.6.6

dangling reference

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

3.6.7

unspecified functionality

code that can be executed, but whose behaviour does not contribute to the requirements of the application

4 Using this document

4.1 Purpose of this document

This document describes programming language vulnerabilities and application design vulnerabilities, as well as mechanisms to avoid them. Programming language vulnerabilities can be design or programming mistakes, problematic language features, or the absence of a language feature.

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. 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 (see IEC 61508-1 for electrical system safety process requirements and IEC 61508-3 for software safety processes). 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.

This document can be used by the following:

- Programmers familiar with the vulnerabilities of a specific language can reference this document for more generic descriptions and their manifestations in less familiar languages.
- Tool vendors can select from this document vulnerabilities to be addressed by their tools.
- Individual organizations planning to write their own coding standards to reduce the number of vulnerabilities in their software products can use this document to assist in the identification of vulnerabilities to be addressed in their coding standards and the selection of coding guidelines to be enforced.
- Organizations or individuals selecting a language for use in a project and considering the vulnerabilities inherent in various candidate languages.
- Scientists, engineers, economists, statisticians, or others who write computer programs can read this document to become more familiar with the issues that can adversely affect their work.
- Educators can use the document as a reference for dangerous vulnerabilities in programming and for guidance to avoid or mitigate them.

There are several ways to avoid a vulnerability:

- Coding guidelines can steer programmers away from constructs found to be problematic.
- Static analysis tools can be used to detect anomalous situations such as usage of a tool that refuses to pass a harmful construct. For instance, this includes a compiler that provides error messages or warnings if a construct is problematic.
- A programming language can be chosen that avoids or mitigates a class of vulnerabilities.
- Specific runtime checks can be written to detect situations that can lead to problematic behaviour.
- Automated analysis tools can be used to enforce coding standards.
- Verification and validation methods such as focused human review of code can be undertaken.

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 are often language specific. In addition, separate language-specific documents have been developed or are under development for particular languages, such as Ada, C, Python, and Fortran that describe the vulnerabilities and their mitigations in a manner specific to each language. For example, ISO/IEC TR 24772-2 describes programming language vulnerabilities for the Ada programming language. The language-dependent documents should be read in conjunction with this language-independent document, as its advice is usually applicable but not replicated in the language-dependent documents.

Throughout this document, avoidance mechanisms are specified to each vulnerability listed to prevent the vulnerabilities from occurring. Readers should be aware, however, that suggested avoidance mechanisms can be contradictory to each other as they provide alternatives to choose from according to project requirements.

As new vulnerabilities are always being discovered, new descriptions can be necessary in future editions to identify the new vulnerabilities. For that reason, a scheme of unique, random identifiers was chosen as permanent identification as opposed to subclause numbering which can change between editions. Each description has been assigned an arbitrarily generated, unique three-letter code. Tool vendors can use the three-letter codes as a succinct way to “profile” the selection of vulnerabilities considered by their tools.

4.2 Applying this document

This document is expected to be used in the creation of software that is safe, secure and trusted within the context of the system in which it is fielded. IEC 61508–3 defines safety-related software as software that is used to implement safety functions in a safety-related system. Notwithstanding that in some domains a distinction is made between safety-related software (that can lead to harm) and safety-critical software (that can be life threatening), this document uses the term safety-critical for all vulnerabilities that can result in safety hazards. Similar to the security-related systems defined in ISO/IEC 27001, this document uses the term security-critical systems in the description of all vulnerabilities that can result in security hazards.

This document is expected to be used in conjunction with some of the following documents, depending upon the planned application of the software:

- IEC 61508-1 and IEC 61508-3 on functional safety;
- ISO/IEC 27001 and ISO/IEC 27002 on security, and application-related standards produced by ISO/IEC/JTC 1/SC 27;
- national safety or security standards;
- sector-specific standards such as MISRA C for automotive sector;^[3]
- corporate or organizational standards and directives.

In particular, this document provides answers for questions raised in the construction of:

- safety-critical applications;
- security-critical applications;
- mission-critical/ business-critical applications;
- scientific, modelling and simulation applications that have social impact.

Organizations can use this document for system or application development following the relevant standards in their safety, security or application domains, in order to:

- determine the criticality of the system, including safety levels, security and privacy;
- analyse failure modes of the system, including omission failures, commission failures, value and timing failures;

- identify and analyse external events and how they can affect the system; or
- identify and analyse attack surfaces of the system.

To use this document effectively, organizations are expected to:

- identify the programming language(s) to be used in programming the applications in the system;
- identify and analyse weaknesses in the product or system, including systems, subsystems, modules, and individual components;
- identify and analyse sources of programming errors;
- determine acceptable programming paradigms and practices to avoid vulnerabilities using the documentation provided in [5.2](#), [Clause 6](#) and [Clause 7](#);
- map the identified acceptable programming practices into organizational coding standards;
- select and deploy tooling and processes to enforce coding rules or practices;
- implement controls (in keeping with the requirements of the safety, security and privacy needs of the system) that enforce these practices and procedures to ensure that the vulnerabilities do not affect the safety and security of the system under development.

In choosing avoidance and mitigation mechanisms, organizations should consult the language-dependent documents of the ISO/IEC 24772 series applicable to their chosen programming language(s), such as ISO/IEC TR 24772-2 for Ada (ISO/IEC 8652) and ISO/IEC TR 24772-3 for C (ISO/IEC 9899).

Tool vendors that follow this document provide tools that diagnose the vulnerabilities described in this document.

Programmers and software designers that follow this document adopt the architectural and coding guidelines of their organization and choose appropriate mitigation techniques when a vulnerability is not avoidable.

4.3 Structure of this document

The rest of the document is organized as follows:

[Clause 5](#) explains how many of the vulnerabilities common to programming languages occur. The issues discussed are not vulnerabilities but are language characteristics that can lead to mistakes and vulnerabilities that can be exploited. [Table 1](#) provides a summary list of the top 20 approaches to avoid the most common vulnerabilities with references to the applicable more detailed descriptions provided in [Clauses 6](#) and [7](#). For many that cannot invest the resources to research all of the vulnerabilities documented in [Clauses 6](#) and [7](#), implementing the documented mechanisms in [Table 1](#) 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, the tool environment, and the operating system that is being used.

This document will rarely be used in isolation, as every program is written in one or more programming languages. Therefore, this document is supported by a set of standards or technical reports, i.e. ISO/IEC TR 24772-2 (for Ada), ISO/IEC TR 24772-3 (for C), that can provide additional specific documentation on the application of this document to the specific language in question.

[Clause 7](#) provides descriptions of selected vulnerabilities, generally unrelated to programming language features, which have been found and exploited in a number of applications. These vulnerabilities result from

design decisions made by coders in the absence of suitable language library routines or other mechanisms but have known mitigation techniques. 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](#) generally do not include the use of programming language-specific features or choices but consist of alternate design choices or programming techniques.

[Annex A](#) is a categorization of the vulnerabilities of this document by general topic areas.

[Annex B](#) summarizes information for language designers cited in the subclauses of [Clause 6](#) entitled “Implications for language design and evolution”.

Throughout this document, the font `courier` is used for tokens typically present in programming languages, such as `false` and `true`, but also for representative program samples from actual programming languages.

5 General vulnerability issues and primary avoidance mechanisms

5.1 General vulnerability issues

5.1.1 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 the usage of programming languages in ways that render the execution of the code less predictable, or the usage of design paradigms that weaken the application and make it susceptible to attack.

Achieving predictable execution is complicated by that fact that software is often 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, the 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, it is crucial that programmers take additional care to ensure predictable execution despite the new challenges.

Software vulnerabilities are characteristics of software that permit software to execute in ways that are unexpected. Programmers introduce vulnerabilities into software by using language features that are inherently unpredictable in the various circumstances outlined above or by using features in a manner that reduces predictability. Although, complete predictability is an ideal (particularly because new vulnerabilities are often discovered through experience), programmers 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.

Here is an example to 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 sets and enforces 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.

The primary purpose of this document is to survey common programming language vulnerabilities; which is done in [Clause 6](#). Each description explains how an application vulnerability can result and provides various mitigations and avoidance mechanisms that can prevent the vulnerability from appearing in a program.

[Clause 7](#) documents vulnerabilities that do not directly result from language vulnerabilities. For example, it is possible that a programmer stores a password in plain text (see [7.17](#) "Insufficiently protected stored credentials [XYM]") because the programming language does 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.

5.1.2 Sources of unpredictability in language specification

5.1.2.1 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 often require compatibility with older versions of the language to be maintained, even to the extent of retaining inherently vulnerable features. Sometimes the full implications and the interactions of new or complex features are not completely known, especially when used in combination with other features.

5.1.2.2 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.

5.1.2.3 Unspecified behaviour

The language specification incompletely specifies the behaviour of some features, leaving the language implementer to choose from a finite set of choices, but the choice is not always apparent to the programmer. In such cases, different compilers or the same compiler with different options processing the code selected can lead to different results, with possible harmful results.

5.1.2.4 Implementation-defined behaviour

In some cases, the results of execution depend upon characteristics of the compiler that was used, the processor upon which the software is executed, or the other systems with which the software has interfaces. In principle, it is possible to 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 leads to problems when a different processor or compiler is used — sometimes even if different compiler options are used.

5.1.2.5 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 in C-based languages, typing "=" for assignment when one really intended "==" for comparison).

5.1.2.6 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.

5.1.3 Sources of unpredictability in language usage

5.1.3.1 Porting and interoperation

The behaviour of a program can change when it is recompiled using a different compiler, recompiled using different compilation options, executed with different libraries, executed on a different platform, or even interfaced with different systems. 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.

5.1.3.2 Compiler selection and usage

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

5.2 Primary 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](#) documents the most effective mitigations, together with references to which vulnerabilities they apply.

Table 1 — Primary avoidance mechanisms for software developers

Number	Avoidance mechanism	Applicable vulnerabilities	
1	Validate input, not make assumptions about the values of parameters and check parameters for valid ranges and values in the calling and/or called functions before performing any operations.	6.6 [FLC] 7.18 [XZN]	7.13 [XZP] 7.28 [CCM]
2	When functions return error values, check the error return values before processing any other returned data.	6.36 [OYB] 6.60 [CGT]	
3	Enable compiler static analysis checking and resolve compiler warnings.	6.8 [HBC] 6.14 [XYK] 6.16 [PIK] 6.18 [WXQ] 6.22 [LAV] 6.26 [XYQ] 6.29 [TEX] 6.34 [QTR] 6.38 [YAN] 6.47 [DJS] 6.56 [EWF] 6.60 [CGT] 6.62 [CGS]	6.10 [XYW] 6.15 [FIF] 6.17 [NIA] 6.19 [YZS] 6.25 [KOA] 6.27 [CLL] 6.30 [XZH] 6.36 [OYB] 6.39 [XYL] 6.54 [BRS] 6.57 [FAB] 6.61 [CGX] 7.28 [CCM]
4	Run a static analysis tool to detect anomalies not caught by the compiler.	6.3 [STR] 6.7 [CJM] 6.10 [XYW] 6.15 [FIF] 6.17 [NIA] 6.19 [YZS] 6.25 [KOA] 6.27 [CLL] 6.30 [XZH] 6.36 [OYB] 6.39 [XYL] 6.54 [BRS] 6.57 [FAB] 6.61 [CGX] 7.28 [CCM]	6.6 [FLC] 6.8 [HBC] 6.14 [XYK] 6.16 [PIK] 6.18 [WXQ] 6.22 [LAV] 6.26 [XYQ] 6.29 [TEX] 6.34 [QTR] 6.38 [YAN] 6.47 [DJS] 6.56 [EWF] 6.60 [CGT] 6.62 [CGS]
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 [FLC] 6.8 [HBC] 6.16 [PIK]	
6	Allocate and free resources, such as memory, threads or locks, at the same level of abstraction.	6.14 [XYK]	
7	Avoid constructs that have unspecified but bounded behaviour, and if the construct is needed, test for all possible behaviours.	6.24 [XYK] 6.56 [EWF]	
8	Make error detection, error reporting, error correction, and recovery an integral part of a system design.	6.36 [OYB]	
9	Use only those features of the programming language that enforce a logical structure on the program.	6.31 [EWD]	
10	Avoid using features of the language which are not specified to an exact behaviour or that are undefined, implementation-defined or deprecated.	6.55 [BQF] 6.57 [FAB] 6.59 [CGA]	6.56 [EWF] 6.58 [MEM]
11	Avoid using libraries without proper signatures.	6.34 [QTR]	
12	Prohibit the modification of loop control variables inside the loop body.	6.29 [TEX]	
13	Prohibit assignments within Boolean expressions, even if allowed by the language.	6.25 [KOA]	

Table 1 (continued)

Number	Avoidance mechanism	Applicable vulnerabilities
14	Prohibit dependence on side effects of a term in the expression itself.	6.31 [EWD] 6.24 [SAM]
15	Use names that are clear and visually unambiguous and be consistent in choosing names.	6.17 [NIA]
16	Use careful programming practice when programming border cases.	6.6 [FLC] 6.29 [TEX] 6.30 [XZH]
17	Beware of short-circuiting behaviour when expressions with side effects are used on the right side of a short-circuited Boolean expression, since a left-hand expression evaluating to <code>false</code> , dictates that the right-hand expression, including function calls with side effects, will not be evaluated.	6.24 [SAM] 6.25 [KOA]
18	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 [CLL]
19	Avoid using floating-point arithmetic when integers would suffice, especially for counters associated with program flow, such as loop control variables.	6.4 [PLF]
20	Sanitize, erase, or encrypt data that will be visible to others (for example, freed memory, transmitted data).	7.11 [EWR] 7.12 [HTS]

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 the other parts of the ISO/IEC 24772 series. In each language-specific part of the ISO/IEC 24772 series, such as ISO/IEC TR 24772-2 (Ada), the behaviour of the programming language is assumed to be as specified by the language standard cited in the respective part of the ISO/IEC 24772 series. 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 vulnerability descriptions in this document are written in a language-independent manner except when specific languages are used in examples. Language-specific vulnerability descriptions and avoidance mechanisms are found in the respective language-specific parts of the ISO/IEC 24772 series (e.g. ISO/IEC TR 24772-2 for the Ada programming language), which mirror the structure of this document.

Where applicable, cross-references to existing coding guidelines or rules are provided in the subclauses entitled "Related coding guidelines".

In general, this clause will use the terminology that is most natural to the description of each individual vulnerability. Hence, 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 Related coding guidelines

JSF AV Rules^[34]: 148 and 183

MISRA C^[39]: 4.6, 10.1, 10.3, and 10.4

MISRA C++^[40]: 3-9-2, 5-0-3 to 5-0-14

CERT C Secure Coding Standard^[41]: DCL07-C, DCL11-C, DCL35-C, EXP05-C and EXP32-C

Ada Quality and Style Guide^[1]: 3.43.4

6.2.3 Mechanism of failure

The type of a data object informs the compiler how values are represented, and which operations are available. 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 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.

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 considered 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.

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 will focus 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 type. 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 is required by the programming language 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, such as the C bounds-checking interface (see ISO/IEC TR 15942). Therefore, a programmer skilled in one language can very well code inadvertent type errors when using a different language.

Programs should be type-safe because the application of operations to operands of an inappropriate type often 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.

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 (ISO/IEC 8652) fragment declares two distinct floating-point types:

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

The declarations make it impossible by the language rules to add a value of type Celsius to a value of type Fahrenheit without explicit conversion. Of course, explicit conversions require additional numeric calculations that respect the relationship of the real-world units being converted. For example, $F = CC$ (where F is Fahrenheit and CC is Celsius) only works in the special case when $CC = -40$, otherwise it is necessary to have:

```
F = Convert_to_Fahrenheit(CC)
```

where the function `Convert_To_Fahrenheit` performs $9 \times C / 5 + 32$.

As another example, the following Pascal code

```
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 will be represented in 16 bits, 32 bits or 64 bits depending on the target architecture. In this case, 16 bit integers are insufficient.

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

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

- 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, such as the C ec-checking interface, ISO/IEC TR 24731;

- use available language and tool capabilities to preclude or detect the occurrence of implicit type conversions, such as those in mixed type arithmetic. If this is not possible, human review can assist in searching for implicit conversions;
- avoid explicit type conversion of data values except when there is no alternative. Documenting such occurrences makes the justification available to maintainers;
- use the most restricted data type that suffices to accomplish the job; for example, using 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, enables tooling to check if all possible choices have been covered;
- 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 and avoid resolution of the issue 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;
- identify all instances of implicit type conversion, and for each case, if the conversion is necessary, change it to an explicit conversion and document the rationale for the 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 the use of predefined numeric types whose ranges and precisions are implementation-defined, instead using types whose ranges and precision are guaranteed.

6.2.6 Implications for language design and evolution

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

- standardizing 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;
- providing a mechanism for selecting data types with sufficient capability for the problem at hand;
- providing a way for the computation to determine the limits of the data types actually selected;
- providing compiler options or other mechanisms 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 can cross computer words for the machine in question. Mistakes can be made as to what bits are accessed because of the "endianness" of the processor (whether the highest order bit is called bit 0 or bit n) 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 Related coding guidelines

JSF AV Rules^[34] 147, 154 and 155

MISRA C^[39]: 1.1, 6.1, 6.2, and 10.1

MISRA C++^[40]: 5-0-21, 5-2-4 to 5-2-9, and 9-5-1

CERT C++ Secure Coding Standard^[6]: EXP38-C, INT00-C, INT07-C, INT12-C, INT13-C, and INT14-C

See also Hogaboom.^[12]

6.3.3 Mechanism of failure

Computer languages frequently provide a variety of sizes for integer variables, such as `short`, `integer`, `long`, and even `big integers`. Interfacing with protocols, device drivers, embedded systems, low-level graphics or other external constructs often require each bit or set of bits to have a particular meaning. Those bit sets can but do not always coincide with the sizes supported by a particular language implementation. When they do not, it is common practice to pack all 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 $28 = 2^4 + 2^3 + 2^2$ to create the mask `11100`) 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 need not be what the programmer expects.

For the C programming language (ISO/IEC 9899), Hogaboom^[12] discusses generic bit manipulation in C. The C++ programming language, ISO/IEC 14882, also shares many of C's characteristics but also provides higher level constructs that help the programmer avoid associated vulnerabilities.

Packing of bits in an integer is not inherently problematic, however, an understanding of the intricacies of bit-level programming is crucial to correct programming of the algorithm. Some computers or other devices number the bits smallest-to-largest while others number them largest-to-smallest.

NOTE Some programmers think of this as left-to-right and right-to-left. Common terminology discusses shifting bits left-to-right or right-to-left where the sign bit (if present) is considered to be the left-most bit.

Storage organization can cause problems when interfacing with external devices that number the bits in opposite order. One problem arises when incorrect assumptions are made when interfacing with external data sources or sinks and the ordering of the bits or words are not the same on both sides. Programmers can inadvertently use the sign bit in a bit field but 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. Knowledge of the storage and ordering of the bits is essential when doing bit-wise operations across multiple words, as bytes can 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.

6.3.5 Avoiding the vulnerability or mitigating its effects

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

- 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 supported by the language, use bit fields in preference to binary, octal, or hexadecimal representations;
- avoid bit operations on signed operands;
- localize and document code associated with explicit manipulation of bits and bit fields;

- use static analysis tools that identify and report reliance upon 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, consideration should be given to creating a standardized application programming interface (API) for bit manipulations that is independent of word size and machine instruction set.

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 ISO/IEC 60559. If ISO/IEC 60559 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 often requires an endlessly repeating string of bits or more binary digits than are available for representation. A floating-point number is only an approximation, albeit 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 are often not aware of the algorithms, or the domain values for a particular algorithm that require attention.

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

6.4.2 Related coding guidelines

JSF AV Rules^[34]: 146, 147, 184, 197, and 202

MISRA C^[39]: 1.1 and 14.1

MISRA C++^[40]: 0-4-3, 3-9-3, and 6-2-2

CERT C Secure Coding Standard^[41]: FLP00-C, FP01-C, FLP02-C and FLP30-C

Ada Quality and Style Guide^[1]:

- 5.5 subsection “Accuracy of Operations with Real Numbers”
- 7.2 subsection “Accuracy Model”

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 $0.333333\dots$. 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:

0.00011001100110011001100110011001100110011001100110011...

which is $0 \cdot 1/2 + 0 \cdot 1/4 + 0 \cdot 1/8 + 1 \cdot 1/16 + 1 \cdot 1/32 + 0 \cdot 1/64 \dots$ and no matter how many digits are used, the representation will still only be an approximation of $1/10$. Therefore, when adding $1/10$ 10 times, it is possible that the final result is not 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 upon 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 60559 compliant or in the interpretation of `NAN`'s. Typically, special representations are specified for positive zero and negative zero; infinity and subnormal numbers are specified 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).

Most floating-point implementations are binary. Decimal floating-point numbers are available on some hardware and that capability has been standardized in ISO/IEC 60559 but one should aware what precision guarantees the implementation programming language makes. In general, fixed-point arithmetic is often 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. The other rounding modes "Round toward zero" and "Round away from zero" can result in a more significant loss of precision and can cause unexpected outcome.

Some floating-point functions can return an arbitrary sign when the result is exactly zero. Tests that use the sign of a number rather than its relationship to zero can return unexpected results.

See also Goldberg.^[9]

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.

6.4.5 Avoiding the vulnerability or mitigating its effects

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

- 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 the target system to check the stability and accuracy of the algorithm employed;
- avoid the use of floating-point expressions 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. 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 60559 or that it includes subnormal numbers (fixed point numbers that are close to zero); and 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, but 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 to provide some understanding of the underlying idiosyncrasies of floating-point arithmetic;
- avoid manipulating the bit representation of a floating-point number; instead prefer built-in language operators and functions that are designed to extract the mantissa, exponent, or sign;
- avoid the use of floating-point for exact values such as monetary amount, and instead 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);
- prohibit reliance on the sign of the floating-point `Min` and `Max` operations when both numbers are zero;
- when adding (or subtracting) sequences of floating-point 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, language designers should consider the following items:

- if a language does not already adhere to or only adheres to a subset of ISO/IEC 60559, it should adhere completely to ISO/IEC 60559;
- providing a means to generate diagnostics for code that attempts to test equality of two floating-point values;
- standardizing their data type to ISO/IEC 10967-1:2012 and ISO/IEC 10967-2:2001.

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 it is possible that some members of the set are not properly specified or 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 Related coding guidelines

MISRA C^[39]: 8.12, 9.2, and 9.3

MISRA C++^[40]: 8-5-3

CERT C Secure Coding Standard^[41]: INT09-C

Ada Quality and Style Guide^[1]: 3.4 subsection "Enumeration Types"

See also Holzmann^[13] rule 6.

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; the relationship between the members of the set can change; representation (the map of values of the items) change; and 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 having holes in the representation where values that cannot be defined are propagated.

If arrays are indexed by enumerations containing non-default representations, some implementations can 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.

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

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

- 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 enforced by 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. For languages with a complete enumeration abstraction, this is enforced by the compiler.

6.5.6 Implications for language design and evolution

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

- for languages that currently permit arithmetic and logical operations on enumeration types, a mechanism should be provided to ban such operations program-wide;
- for languages that provide automatic defaults or that do not enforce static matching between enumerator definitions and initialization expressions, a mechanism should be provided to enforce such matching.

6.6 Conversion errors [FLC]

6.6.1 Description of application vulnerability

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

```
aVar:= anExpression
or
value1 + value2
or
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 are 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 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 Related coding guidelines

CWE^[7]: 192. Integer Coercion Error

MISRA C^[39]: 7.2, 10.1, 10.3, 10.4, 10.6-10.8, and 11.1-11.8

MISRA C++^[40]: 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 Secure Coding Standard ^[41]: 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 can 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. An Ariane 5^{[2][39]} launcher failure occurred due to an improperly handled conversion error resulting in the processor being shut down and the destruction of the spacecraft.

Conversion errors can also result in security issues, such as when an attacker inputs 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, when a truncated integer value is used to allocate memory, while the actual length is used to copy information to the newly allocated memory, this results in a buffer overflow, as specified in ISO/IEC 60559.

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 metres 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](#) "Polymorphic variables [BKK]";
- weakly typed languages that do not strictly enforce typing rules;
- 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

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

- 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.

NOTE 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; see Jones.^[33]

- use explicit range checks to protect each operation, but pay attention to the large number of integer operations that are susceptible to these problems and the number of checks required to prevent or detect exceptional conditions, potentially resulting in prohibitively labour intensive implementation and expensive computation;
- 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;
- document any explicit type conversion made necessary by the algorithm 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; instead, 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;
- 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, language designers should consider the following items:

- providing mechanisms to prevent programming errors due to conversions;
- making all type-conversions explicit or at least generating warnings for implicit conversions where loss of data can 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 Related coding guidelines

CWE^[Z]: 170. Improper Null Termination

CERT C Secure Coding Standard^[41]: STR03-C, STR31-C, STR32-C, and STR36-C

6.7.3 Mechanism of failure

String termination errors occur when the termination character is solely relied upon 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 because, for a string that is passed as input or generated by a library, a programmer assumes that it contains a string termination character when it does not.

If the programmers forget to allocate space for the string termination character, they can 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 will almost always 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

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

- avoid relying 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, consider the following items:

- eliminate library calls that make assumptions about string termination characters;
- check bounds when an array or string is accessed, such as the C bounds checking interface (see ISO/IEC TR 15942);
- specify a string construct that does not require 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. Hence, access to the region before the beginning or beyond the end of the buffer data are equally possible, dangerous and maliciously exploitable.

6.8.2 Related coding guidelines

CWE^[7]:

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

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 ^[34]: Rule 15 and 25

MISRA C^[39]: 21.1

MISRA C++^[40]: 5-0-15 to 5-0-18

CERT C Secure Coding Standard^[41]: 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.

In all cases, an exception can be raised if the accessed location is outside of some permitted range of the run-time environment. Typical kinds of failures are:

- 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 can be used to obtain information that is intended to be confidential.
- A write access will not result in the intended value being updated and can 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 (e.g. a function's return address) which can change a program's control flow.
- An inadvertent or malicious overwrite of function pointers in memory can cause them to point to an unexpected location or an 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, by means of an index, by pointer, or by using the physical memory address to access memory locations;
- languages that do not automatically allocate storage when accessing an array element for which storage has not already been allocated;
- 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

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

- use any implementation-provided functionality to automatically check array element accesses and prevent out-of-bounds accesses;
- use static analysis to verify that all array accesses are within the permitted bounds. Such analysis often requires 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;
- perform sanity checks on all calculated expressions used as an array index or for pointer arithmetic;
- ascertain whether 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.

NOTE 1 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.

NOTE 2 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, language designers should consider the following items:

- providing safe copying of arrays as built-in operation;
- providing array copy routines in libraries that perform checks on the parameters to ensure that no buffer overrun can occur;
- performing automatic bounds checking on accesses to array elements, unless the compiler can statically determine that the check is unnecessary. It is possible that this capability is optional for performance reasons;
- where pointer types are provided, 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.

6.9.2 Related coding guidelines

CWE^[7]:

129. Unchecked Array Indexing

676. Use of Potentially Dangerous Function

JSF AV Rules^[34]: 164 and 15

MISRA C^[39]: 21.1

MISRA C++^[40]: 5-0-15 to 5-0-18

CERT C Secure Coding Standard ^[41]: ARR30-C, ARR32-C, ARR33-C, and ARR38-C

Ada Quality and Style Guide^[1]:

- 5.5 subsection “Array Attributes”
- 7.6 subsections “Input/Output on Access Types” and “Package Ada.Streams.Stream_IO”

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 into a buffer. Unchecked array indexing can result in the corruption of relevant memory and perhaps the corruption of instructions. If the memory corrupted contains data, the program can continue to function with improper values or stop due to some system error, e.g. an access outside the valid memory. If the memory corrupted contains instructions, then the access can result in arbitrary or malicious changes to the executing program. If the corrupted memory can be effectively controlled, then the execution of arbitrary code becomes possible, as with a standard buffer overflow.

Some language implementations statically detect out of bound access and generate a compile-time diagnostic. At runtime, an implementation that detects the out-of-bound access can provide a notification. Such a notification can be treatable by the program, or not. 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 can be available, or not, 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, if this does not match the programmer's intent, it can mask errors. Some languages provide for whole array operations that 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.

6.9.5 Avoiding the vulnerability or mitigating its effects

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

- 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;
- prohibit the suppression of language-provided bounds checks without first statically verifying that the code is free from out-of-bounds accesses.

6.9.6 Implications for language designers

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

- providing 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 obviate the need to access individual elements;

- providing the capability to generate exceptions or automatically extend the bounds of an array to accommodate accesses that could 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 also 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 Related coding guidelines

CWE^[7]: 121. Stack-based Buffer Overflow

JSF AV Rules^[34]: 15

MISRA C^[39]: 21.1

MISRA C++^[40]: 5-0-15 to 5-0-18

CERT C Secure Coding Standard^[41]: ARR33-C and STR31-C

Ada Quality and Style Guide^[1]:

7.6 subsection “Input/Output on Access Types”

7.6 subsection “Package Ada.Streams.Stream_IO”

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 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](#) “Unchecked array indexing [XYZ]”.

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](#) “Unchecked array indexing [XYZ]”.

6.10.5 Avoiding the vulnerability or mitigating its effects

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

- only use library functions that perform checks on the arguments to ensure no buffer overrun can occur and 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;

NOTE Such analysis can require the source code to contain certain kinds of information, for example, that the bounds of all declared arrays are explicitly specified, or that pre- and post-conditions are specified as annotations or language constructs.

- sanitize all input data so that excessively large input data that can result in overflows is rejected;
- prohibit the suppression of any bounds checks provided by the language.

6.10.6 Implications for language design and evolution

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

- providing libraries that perform checks on the parameters to ensure that no buffer overrun can occur;
- 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.

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

6.11.2 Related coding guidelines

CWE^[7]:

136. Type Errors

188. Reliance on Data/Memory Layout

JSF AV Rules^[34]: 182 and 183

MISRA C^[39]: 11.1-11.8

MISRA C++^[40]: 5-2-2 to 5-2-9

CERT C Secure Coding Standard ^[41]: INT11-C and EXP36-A

Ada Quality and Style Guide^[1]:

- 7.6 subsection “Input/Output on Access Types”
- 7.6 subsection “Package Ada.Streams.Stream_IO”

See also Hatton^[10] rule 13: Pointer casts.

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 read or modified. 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 or from pointers to data.

6.11.5 Avoiding the vulnerability or mitigating its effects

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

- treat all compiler pointer-conversion warnings as serious errors;
- adopt programming guidelines, preferably augmented by static analysis, that restrict pointer conversions, such as the rules itemized above from JSF AV,^[34] CERT,^[41] Hatton^[10] or MISRA C^[39];
- use other means of assurance such as proofs of correctness, analysis with tools, verification techniques, or other methods to verify 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, language designers 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.

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 Related coding guidelines

JSF AV^[34] Rule: 215

MISRA C^[39]: 18.1-18.4

MISRA C++^[40]: 5-0-15 to 5-0-18

CERT C Secure Coding Standard ^[41]: 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

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

- avoid using pointer arithmetic for accessing anything except composite types;
- prefer indexing for accessing array elements rather than using pointer arithmetic in languages that permit the dual modes of access;
- limit pointer arithmetic calculations to the addition and subtraction of integers.

6.12.6 Implications for language design and evolution

No implications apply.

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 Related coding guidelines

CWE^[7]:

476. NULL Pointer Dereference

JSF AV^[34]: Rule 174

CERT C Secure Coding Standard^[41]: EXP34-C

Ada Quality and Style Guide^[1]: 5.4 subsection “Dynamic Data”

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 itself;
- 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, language designers should 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

Memory designated by a dangling reference can be reused as soon as the referenced object has been deleted; therefore, any subsequent access through the dangling reference can affect an apparently arbitrary location of memory, corrupting data or code.

This description concerns dangling references to the heap. The description of dangling references to stack frames can be found in [6.33](#) “Dangling reference to stack frame [DCM]”. 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 it can have no adverse effects at all.

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

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

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.

6.14.2 Related coding guidelines

CWE[\[7\]](#):

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

416. Use After Free

MISRA C[\[39\]](#): 18.1-18.6

MISRA C++[\[40\]](#): 0-3-1, 7-5-1, 7-5-2, 7-5-3, and 18-4-1

CERT C Secure Coding Standard[\[41\]](#): MEM01-C, MEM30-C, and MEM31.C

Ada Quality and Style Guide[\[1\]](#):

- 5.4 subsection “Dynamic Data”
- 7.2 subsection “Storage Pool Mechanisms
- 7.6 subsection “Input/Output on Access Types”

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. A deallocation causes all remaining copies of the reference to become dangling.

The use of dangling references to previously freed memory can have a 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, 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 deallocation, 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, it is possible that various function pointers are 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.

6.14.5 Avoiding the vulnerability or mitigating its effects

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

- 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, such as if the language is object-oriented, ensure that object destructors delete each chunk of memory only once, and ensure that all pointers are set to `NULL` once the memory they point to have been freed;
- 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;
- allocate and free memory at the same level of abstraction, and ideally in the same code module.

6.14.6 Implications for language design and evolution

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

- providing implementations of the free function that can 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, providing an assertion mechanism for checking properties at run-time, with the option to inhibit assertion checking if efficiency is a concern;