ICS 35.240.01
CCS L00

# 团 体 标 准

T/CECC 41—2025

# 信息安全技术 软件供应链软件产品检测要素和方法

Information security technology—testing requirements and approaches for software supply chain products

2025-06-30 发布 2025-06-30 实施

中国电子商会 发布

# 目　次

# 前　　言

本文件按照GB/T 1.1—2020《标准化工作导则 第1部分：标准化文件的结构和起草规则》的规定起草。

请注意本文件的某些内容可能涉及专利。本文件的发布机构不承担识别这些专利的责任。

本文件由中国电子商会提出并归口。

本文件由厦门理工学院软件与信息安全测评实验室发起。

本文件起草单位：厦门理工学院、扬州大学信息工程学院、西北工业大学软件学院、厦门软件职业技术学院、湖南麒麟信安科技股份有限公司、金陵科技学院、海南软件职业技术学院、安徽科测信息技术有限公司、中科数测固源科技（安微）有限公司、安顺高新区检测中心有限公司、南京大学、电设信科（北京）技术有限公司、中国科学院软件研究所智能软件研究中心、北京中科微澜科技有限公司、北京理工大学软件评测中心、公安部第三研究所、中移互联网有限公司、上海同思廷软件技术有限公司、扬州数安技术有限公司、德中睿智互联网技术（上海）有限公司、江西省科技基础条件平台中心、广东南北科技有限公司、天津工业大学软件学院、宁波城市职业技术学院、内蒙古鹏捷工程管理咨询有限公司、物产中大数字安全科技（浙江）有限公司、苏州棱镜七彩信息科技有限公司、亿量（厦门）科技有限公司、云南省数字证书认证中心有限公司、牡丹江师范学院、厦门海洋职业技术学院、厦门众联世纪股份有限公司、软链安（厦门）信息安全有限公司、唐山职业技术学院、中国电子商会信息工程测试专业委员会。

本文件主要起草人：肖蕾、吴潇雪、朱顺痣、张晨、陈松政、曾岳、张国防、郑炜、许奇臻、盛浪、何铁科、孙小兵、杜侠、聂菁、刘芝影、牛永玺、吴敬征、杨牧天、王智钢、孙永清、柯树森、薛静锋、周震漪、刘熠、聂菁、王梅、庞敏、陈亚贤、菅志刚、刘志远、罗天悦、韩武光、王丽敏、何康鑫、余毅斌、石洛宜、万旭成、付康、王颉、朱云娜、王刚、朱凌军、周建文、方淼、赖宜亮、潘自鹏、谢小竹、史俊、黄鹏程、薄莉莉、蒋先发、刘晓更、张佳乐、陈荣赏、陈洪波、申煜湘、梁大功、林欣扬、邵妍洁、施炜利、施航海、黄浩东、李妙君、郭健、李艺豪、吴一凡、高振宇、童小刚、于海存、康健、张虎、马生武、杨庆师。

# 信息安全技术 软件供应链软件产品检测要素和方法

## 1 范围

本文件规定了软件供应链中的软件产品的检测要素和检测方法。

本文件适用于第三方检测机构对软件产品的基本信息、成分信息和安全信息等方面的测评工作。

## 2 规范性引用文件

下列文件中的内容通过文中的规范性引用而构成本文件必不可少的条款。其中，注日期的引用文件，仅该日期对应的版本适用于本文件；不注日期的引用文件，其最新版本（包括所有的修改单）适用于本文件。

GB/T 11457-2006 信息技术 软件工程术语

GB/T 15532-2008 计算机软件测试规范

GB/T 25000.51-2016 系统与软件工程 系统与软件质量要求和评价（SQuaRE）第51部分：就绪可用软件产品（RUSP）的质量要求和测试细则

GB/T 30279-2020 信息安全技术 网络安全漏洞分类分级指南

GB/T 36475-2018 软件产品分类

GB/T 43848-2024 网络安全技术 软件产品开源代码安全评价方法

ISO/IEC 30111:2019 信息技术-安全技术-漏洞处理流程

## 3 术语和定义

GB/T 11457-2006 和 GB/T 15532-2008 界定的以及下列术语和定义适用于本文件。

3.1

**软件供应链** software supply chain

指一个通过一级或多级软件设计、开发阶段编写软件，并通过软件交付渠道将软件从软件供应商送往软件用户的系统。

3.2

**软件物料清单** software bill of materials

记录构成软件的所有组件及其版本信息的数据结构。

3.3

**软件产品** software product

计算机软件、信息系统或设备中嵌入的软件或在提供计算机信息系统集成、应用等技术服务时提供的计算机软件,表现形式为一组计算机代码、规程以及可能的相关文档和数据。

[来源：GB/T 36475—2018,3.1,有修改]

3.4

**身份认证** identity authentication

指验证用户或系统实体的身份，以确保其确实是其声称的身份的过程，确保只有合法用户才能访问系统资源。

3.5

**开源代码** open source code

公众可以获取源代码的计算机代码。

[来源： GB/T 43848—2024,3.2,有修改]

## 4 缩略语

API ：应用程序编程接口(Application Programming Interface)

DAST：动态应用安全测试（Dynamic Application Security Testing）

IAST：交互式应用安全测试（Interactive Application Security Testing）

SAST：静态应用安全测试（Static Application Security Testing）

SBOM：软件物料清单（Software Bill of Materials）

SDK：软件开发工具包（Software Development Kit）

## 5 检测要素

### 5.1 软件产品信息

在软件供应链中，软件产品信息涵盖了软件从开发到分发整个过程中所涉及的各种信息和相关联的源代码和组件，如表 1 所示。

**表 1 软件产品信息要素**

| 分类 | 检测要素名称 | 说明 |
|---|---|---|
| 软件基本信息 | 软件名称 | 软件供应商发布的软件名称 |
| | 软件版本 | 软件供应商发布的软件版本 |
| | 完整性信息 | 数字签名等 |
| 软件开发信息 | 软件工程文档 | 需求分析、设计、编码、测试等阶段详细信息 |
| | 开发工具和框架 | 如集成开发环境（IDE）、编译器、测试框架等 |
| | 引入组件数量 | 开源代码、自研、第三方组件数量 |
| | 版本控制信息 | 软件的版本历史、变更记录和分支信息等 |

| | | |
|---|---|---|
| 软件发布信息 | 供方 | 直接供应商 |
| | 源开发商 | 开发者 |
| | 分发渠道 | 如官方网站、应用商店、第三方平台等 |
| | 授权信息 | 有效期限、许可证、序列号等 |
| | 部署环境要求 | 如操作系统、数据库、中间件等 |
| | 更新历史和补丁信息 | 有助于修复软件潜在的安全问题 |

## 5.2 软件物料清单

软件物料清单要素如表 2 所示。

**表 2 软件物料清单要素**

| 字段名 | 字段描述 | 字段类型 | 字段必要性 |
|---|---|---|---|
| formatName | 清单格式名称 | string | 必选项 |
| formatVersion | 格式版本 | string | 必选项 |
| documentName | 文档名称 | string | 必选项 |
| listID | 清单标识 | string | 必选项 |
| timestamp | 时间戳 | string | 必选项 |
| authors | 创建者 | string | 必选项 |

软件成分的要素如表 3 所示。

**表 3 软件成分的清单要素**

| 字段名 | | 字段描述 | 字段类型 | 字段必要性 |
|---|---|---|---|---|
| componentID | | 组件标识 | string | 必选项 |
| componentName | | 组件名称 | string | 必选项 |
| componentVersion | | 组件版本 | string | 必选项 |
| componentDescription | | 组件描述 | string | 可选项 |
| supplier | | 供应商 | string | 可选项 |
| acquisitionChannel | | 获取途径 | enum (of string) | 可选项 |
| progLanguage | | 组件语言 | array of string | 可选项 |
| licenseName | | 许可证名称 | array of string | 必选项 |
| copyrightText | | 版权文本 | array of string | 必选项 |
| downloadUrl | | 下载链接 | string | 可选项 |
| homepageUrl | | 主页链接 | string | 可选项 |
| intergrity | hashAlg | 杂凑算法 | string | 必选项 |
| | messageDigest | 消息摘要 | string | 必选项 |

### 5.3 软件产品合规信息

#### 5.3.1 合法性

软件供应链中的每个环节都应确保所开发的软件、使用的工具和技术符合当地的法律法规要求。

#### 5.3.2 合规性

软件供应链中的每个环节都应确保所开发的软件、使用的工具和技术符合对所使用地区的地方标准、国家标准和国际标准的合规性检测要求。

#### 5.3.3 隐私保护

确保在软件开发和供应链管理中，严格遵守关于个人隐私和数据保护的法律法规，确保用户数据的安全和隐私。

#### 5.3.4 开源许可协议

如果软件供应链中使用了开源组件，应确保这些组件符合开源社区的规范和标准，避免使用存在安全隐患或不合规的组件。

### 5.4 软件源代码编码规范

#### 5.4.1 概述

软件编码代码规范应包括可读性、可维护性和可靠性的检测要素。

#### 5.4.2 可读性

可读性应具有以下要求：

a）命名规则：一致性、可读性和避免冲突；

b）代码风格：一致性和清晰性；

c）注释规则：注释内容、注释风格和文档注释。

#### 5.4.3 可维护性

可维护性应具有以下要求：

a）逻辑单元：单一职责和模块化；

b）模块规则：模块边界和模块复用；

c）函数规则：函数设计和函数长度。

#### 5.4.4 可靠性

可靠性应具有以下要求：

a）语法检查：静态检查、编译和测试；

b）异常处理：通用异常处理、清理资源和异常链；

c）错误处理：详细错误信息、用户友好和日志记录。

## 5.5 软件产品代码测试要求

### 5.5.1 概述

软件源代码测试安全质量要求参考了 GB/T 43848-2024 的 6.3 开源代码安全质量评价流程。软件二进制代码测试安全质量要求参考了 ISO/IEC 30111：2019 的安全质量评价标准。

### 5.5.2 源代码漏洞率

源代码漏洞率的测试要求如下：

a）检测软件产品形成源代码漏洞检测报告，检查各源代码模块是否存在已知漏洞；

b）检查源代码中每千行的已知漏洞数量；

c）计算有漏洞的源代码中平均漏洞个数。

### 5.5.3 源代码漏洞严重性

源代码漏洞严重性的测试要求如下：

a）将各已知漏洞的被利用性参数进行赋值，根据赋值结果，按照 GB/T 30279—2020 的附录 A 计算得出漏洞被利用性分级；

b）将已知漏洞的影响程度参数进行赋值，根据赋值结果，按照 GB/T 30279—2020 的附录 B 计算得到影响程度分级；

c）根据被利用性和影响程度分级的结果，按照GB/T 30279—2020的附录D计算得到安全漏洞分级结果；

d）检查软件产品包含的源代码是否存在中危及以上漏洞；

e）统计软件产品包含的源代码中危及以上漏洞占比。

### 5.5.4 源代码漏洞修复率

源代码漏洞修复率的评价流程如下：

a）检查软件产品包含的开源代码漏洞修复记录；

b）检查漏洞出现正式编号的时间；

c）检查修复记录中危及以上漏洞的平均修复时间是否超过 3 个月；

d）统计中危及以上漏洞的平均修复时间在 3 个月内的占比。

### 5.5.5 开源组件版本更新情况

开源代码版本更新情况的评价流程如下：

a）检测软件产品形成开源代码清单列表，检查各开源代码模块当前使用的版本发布时间；

b）检查各开源代码模块在开源社区中最新版本发布时间；

c）检查各开源代码模块是否存在停止维护情况，如 4 年内未更新；

d）统计开源代码为较新稳定版本的占比。

### 5.5.6 二进制代码漏洞严重性

二进制代码漏洞严重性的测试要求如下：

a）对每个已知漏洞的被利用性进行评估，使用漏洞管理工具进行漏洞被利用性分析，根据该结果，按照 ISO/IEC 30111：2019 中的漏洞分类，进行漏洞被利用性分级；

b）对已知漏洞的影响程度进行评估，包括对机密性、完整性和可用性的影响，根据该结果，按照 ISO/IEC 30111：2019 中的漏洞影响评估方法，计算影响程度分级；

c）根据被利用性和影响程度分级的结果，按照 ISO/IEC 30111：2019，计算得到二进制漏洞的安全等级；

d）检查二进制代码中是否存在高危及以上的漏洞（如远程代码执行、堆溢出等）；

e）统计软件产品二进制代码中高危及以上漏洞的占比。

#### 5.5.7 二进制代码漏洞修复率

二进制代码漏洞修复率的评价流程如下：

a）检查二进制代码漏洞的修复记录，确认漏洞是否已被修复，修复记录是否可追溯；

b）检查漏洞修复的编号、修复时间；

c）检查高危及以上漏洞的修复时间是否超过 6 个月；

d）统计高危及以上漏洞修复时间在 6 个月内的占比。

#### 5.5.8 二进制代码版本更新情况

二进制代码版本更新情况的评价流程如下：

a）检测软件产品形成二进制代码清单，检查各二进制模块当前使用的版本发布信息；

b）检查各二进制模块是否为最新版本，或是否存在已知的安全更新；

c）检查二进制代码模块是否为较新稳定版本，如版本发布时间不超过 4 年；

d）统计二进制代码使用的较新稳定版本的占比。

## 6 测试流程

### 6.1 概述

软件供应链中软件产品的测试过程基本遵循 GB/T 15532-2008 的测试计划、测试设计、测试执行和测试总结四个过程，由于软件供应链动态更新的特点，因此还需要进行定期审查和总结。

### 6.2 测试计划

测试计划主要对软件产品的测试过程进行策划。计划应确定测试目标、范围、依据、环境和工具，应分析与评估测试风险，并制定应对措施。测试计划应明确地规定测试环境的要求，包括硬件配置、软件版本、网络配置等，以确保测试结果的可靠性和可重复性；重点明确测试阶段和各阶段的人员角色、任务、时间和工作成果，形成测试进度计划表，如表 4 所示。

表 4 测试进度计划表

| 测评阶段 | 阶段任务 | 责任人/配合人 | 起止时间 | 输出文档 |
|---|---|---|---|---|
| 测试计划 | | | | |
| 测试设计 | | | | |
| 测试执行 | | | | |
| 定期审查 | | | | |
| 测试总结 | | | | |

### 6.3 测试设计

测试设计应根据测试目标，结合被测软件产品的业务和技术特点，明确测试环境和工具，确定测试需求、测试方法、测试内容、测试准入条件和测试准出条件。测试方法应采用信息收集、自动化工具扫描和人工分析相结合的方法。测试内容应包括第 5 章提出检测要素。

### 6.4 测试执行

测试执行包括信息收集、自动化工具扫描和人工分析。

应根据测试用例明确的操作步骤，采用信息收集、自动化工具扫描和人工分析相结合的方法执行测试，记录测试执行过程及测试结果。

### 6.5 定期审查

由于软件供应链具有动态更新的特点，对于软件产品信息、软件物料清单和软件合规信息的检测要素进行应定期审查，记录测试执行过程及测试结果，并出具审查报告。

高风险软件系统，每季度应至少审查一次；低风险软件产品，应半年或一年审查一次。

### 6.6 测试总结

测试总结应包括以下任务：

a）核查测试环境、工具、内容、方法和结果是否正确；

b）确认测试目标和测试需求是否得到满足；

c）总结测试内容、方法和结果，出具测试报告。

## 7 软件产品检测方法

### 7.1 软件产品信息收集

#### 7.1.1 利用内部系统

利用内部的软件管理系统，如配置管理系统、版本控制系统等，获取软件产品的版本信

息、依赖关系、构建脚本等；利用企业资源规划系统，了解软件产品的采购、库存、分发等供应链环节的信息。

#### 7.1.2 与供应商合作

与供应商建立紧密的合作关系，确保能够获取到软件产品的最新信息，包括更新、补丁、安全公告等；通过供应商提供的 API 或 SDK，集成到企业内部的系统中，实现软件产品信息的自动化获取。

### 7.2 软件成分分析

#### 7.2.1 明确分析目标

在进行软件成分分析之前，首先需要明确分析的目标。这可能包括识别软件中的开源组件、检测潜在的漏洞和威胁、确保许可证合规性等。明确目标有助于确定分析的范围和深度。

#### 7.2.2 选择成分分析工具

选择 SCA 工具时，需要考虑以下因素：

a）数据库支持：确保成分分析工具支持全面数据库，以便能够准确地识别和分析软件中的组件；

b）语言支持：根据软件使用的编程语言选择支持相应语言的成分分析工具；

c）报告生成能力：选择能够生成全面、详细的报告的成分分析工具，以便后续分析和处理；

d）集成能力：考虑成分分析工具是否能够与现有的开发流程、CI/CD 管道等集成，以提高分析效率。

#### 7.2.3 扫描软件

依据成分数据库，使用选定的成分分析工具扫描软件。扫描过程将识别软件中的开源组件、依赖项以及它们之间的关系。扫描完成后，查看扫描结果中列出的开源组件，确保它们都是已知且安全的。

### 7.3 软件产品许可合规检测

#### 7.3.1 许可合规信息收集

收集并仔细阅读软件使用许可协议，了解软件的版权归属、使用权限、限制和处罚等方面的规定。对企业内部使用的软件资源进行全面盘点，包括软件类型、数量、版本、授权方式等信息。

#### 7.3.2 审查与分析

审查与分析应具以下要求：

a）协议条款：仔细阅读协议条款，关注版权归属、使用权、限制和处罚等约定，确保企业具备签署协议的授权，避免产生法律纠纷；

b）检查软件：是否按照许可协议的规定进行使用，识别是否存在违反许可条款的软件安装或使用情况；

c）风险评估：对软件的使用情况进行风险评估，识别潜在的法律风险。根据风险评估结果，制定相应的风险应对措施。

### 7.4 软件源代码编码规范检测

#### 7.4.1 可读性检测

可读性检测应具以下要求：

a）命名规则检测：使用集成代码风格检查工具检查变量、函数、类、文件和目录的命名一致性。确保命名符合项目统一命名规范，避免缩写和模糊不清的命名；命名具备描述性，能够准确反映其功能以及是否有命名冲突，如标准库函数、全局变量或关键字等；

b）代码风格检测：使用格式化工具自动格式化代码，结合手动检查确保一致的缩进、空格、行长度等。确保代码遵循统一的代码风格，包括缩进、空格、行长度、括号等格式要求；

c）注释规则检测：使用注释检查工具生成注释文档，确保函数、类和模块具备清晰、准确的注释。检查代码中是否有足够的注释，特别是关键和复杂的代码段。确保注释内容清晰、简洁，且与代码同步更新，避免过时的注释。

#### 7.4.2 代码可维护性检测

代码可维护性检测应具以下要求：

a）逻辑单元检测：静态分析工具和复杂度检查工具用于分析函数、类的长度与复杂度；

b）模块规则检测：依赖分析工具检测模块间的耦合度与接口定义。确保模块间有清晰的边界和接口，避免直接访问内部数据；

c）函数设计检测：使用代码复杂度检测工具帮助分析函数设计。检查函数的输入输出是否明确，避免使用全局变量。

#### 7.4.3 代码可靠性检测

代码可靠性检测应具以下要求：

a）语法检测：集成静态代码分析工具到 CI/CD 管道中，确保代码在提交之前没有语法错误；

b）异常处理检测：异常检测工具检测是否有异常未处理的情况。检查代码是否对所有可能的异常进行处理，避免因未处理的异常导致崩溃；

c）错误处理检测：使用静态分析工具和错误日志检查工具来检测错误处理机制的有效性。检查代码中的错误信息是否详细，能够提供异常类型、描述、代码和相关上下文。

#### 7.4.4 代码修复与优化

基于检测工具提供的报告，分析和修复代码中的问题，确保修复后的代码符合编码规范。对于一些简单的风格或格式问题，可采用自动化工具进行代码修复。对于更复杂的设计问题，应手动优化并确保代码逻辑的健壮性。持续跟踪和分析检测结果，以便及时发现并修复潜在问题，提升代码质量。

#### 7.4.5 持续改进

定期回顾编码规范检测结果，结合项目需求和团队反馈进行调整。根据实际问题和技术演进调整编码规范，确保规范始终符合项目需求。根据行业变化和项目要求不断优化和更新编码规范，确保团队始终能够遵循最新的规范。

### 7.5 软件产品安全质量测试

#### 7.5.1 概述

软件产品安全质量测试主要包括：静态源代码漏洞测试、动态软件产品漏洞测试、交互式软件产品漏洞测试和模糊测试四种。

#### 7.5.2 静态源代码漏洞测试

静态源代码漏洞测试应具以下要求：

a）工具选择：根据软件源代码的编程语言、框架和需求，选择合适的 SAST 工具；

b）配置和集成：将 SAST 工具集成到开发环境中，如 IDE、CI/CD 管道中，配置扫描规则和策略，以适应项目的具体需求；

c）代码扫描：定期或在代码变更时触发 SAST 工具对代码库进行扫描，统计源代码漏洞率，并对各个漏洞的严重性进行计算、分级。生成详细的质量报告、漏洞报告；

d）代码修复：开发团队根据 SAST 工具提供的报告，分析和修复代码中的质量问题和安全漏洞。必要时，可以更新扫描规则以减少误报和漏报并统计源代码漏洞修复率；

e）持续改进：根据扫描结果和漏洞修复情况，不断优化和改进代码安全策略，提升整体代码质量和安全性。

#### 7.5.3 动态软件产品漏洞测试

动态软件产品漏洞测试应具以下要求：

a）工具选择：根据项目需求选择合适的源代码 DAST 工具；

b）环境配置：配置测试环境，确保应用程序在受控环境中运行，并确保测试不会影响生产环境；

c）扫描设置：配置 DAST 工具的扫描策略和规则，包括目标 URL、扫描深度、漏洞类型等，确保覆盖所有需要测试的部分。对于二进制文件 DAST 工具的扫描策略和规则应包括目标二进制文件、扫描深度、漏洞类型等。确保覆盖所有需要测试的二进制代码部分，测试过程中没有遗漏潜在漏洞；

d）执行扫描：启动 DAST 工具，对应用程序进行全面扫描。工具会模拟各种攻击方式，尝试发现潜在的安全漏洞；

e）分析报告：扫描完成后，DAST 工具会生成详细的漏洞报告。开发团队需要分析报告，确认漏洞的真实性和严重性，同时还要统计源代码漏洞率；

f）漏洞修复：根据报告中的漏洞信息，开发团队进行修复和优化。必要时，可以结合 SAST 等其他测试方法，确保全面的安全性保障。最后统计软件产品漏洞修复率；

g）重复测试：修复漏洞后，重新执行漏洞测试工具扫描，验证修复效果，并确保没有引入新的安全问题。

### 7.5.4 交互式软件产品漏洞测试

交互式软件产品漏洞测试应具以下要求：

a）工具选择：根据项目需求选择合适的 IAST 工具；

b）环境配置：配置 IAST 工具，使其能够监控和分析应用程序的运行情况。通常需要将 IAST 代理或探针集成到应用程序服务器或容器中；

c）运行和测试：启动应用程序并执行常规的功能测试、性能测试或安全测试。IAST 工具将自动监控应用程序的运行状态，并在发现漏洞时生成报告；

d）分析报告：IAST 工具会生成详细的漏洞报告，包含软件的漏洞率，漏洞严重性，漏洞的具体位置、触发条件和修复建议。开发团队需要对报告进行分析和验证；

e）漏洞修复：根据 IAST 工具提供的漏洞信息，开发团队进行修复和优化。IAST 工具通常能够在修复后实时验证修复效果。开发团队最后需要统计软件产品漏洞修复率；

f）持续集成和监控：将 IAST 工具集成到持续集成 CI/CD 管道中，确保每次代码变更都经过安全性测试，保持持续监控。

### 7.5.5 模糊测试

模糊测试测试应具以下要求：

a）工具选择：根据测试需求选择合适的模糊测试工具；

b）目标分析：确定要测试的目标软件，了解其输入接口和数据格式，以便配置模糊测试工具生成有效的测试数据；

c）测试配置：配置模糊测试工具，包括输入数据生成策略、测试用例数量、监控参数等，确保测试能够充分覆盖目标软件的各个方面；

d）执行测试：启动模糊测试工具，对目标软件进行大规模的随机输入测试。工具会自动生成并输入测试数据，监控软件的运行状态，记录异常情况；

e）分析结果：在测试过程中，模糊测试工具会生成详细的日志和报告，记录所有导致软件崩溃或异常行为的输入数据。开发团队需要分析这些结果，确定漏洞的根本原因，软件产品漏洞率以及漏洞严重性；

f）漏洞修复：根据分析结果，开发团队修复发现的漏洞和缺陷。必要时，可以重新运行模糊测试，验证修复效果。修复完成后，需要对漏洞修复率进行统计。

# 附 录 A

（规范性）

# 代码示例

## A.1 概述

本附录给出了软件供应链源代码检测内容中，源代码不规范用法示例和规范用法示例。

## A.2 检测内容

### A.2.1 代码质量

#### A.2.1.1 命名规则

示例：

```
# 不推荐的命名：使用缩写且不一致
def calc(x):
    return x * 2
# 推荐的命名：遵循一致的命名规范，避免使用缩写
def calculate_area(radius):
    return radius * radius * 3.14
```

不推荐的例子中，calc 是一个缩写，不明确其具体功能。推荐的例子中，calculate_area 使用了完整的单词，清晰描述了函数的功能，且遵循了项目的命名规范。

#### A.2.1.2 注释规则

示例：

```
# 不推荐的注释：缺乏对复杂代码段的解释
def calculate_factorial(n):
    if n == 0:
        return 1
    else:
        return n * calculate_factorial(n-1)
# 推荐的注释：详细解释代码段的功能和逻辑
def calculate_factorial(n):
    """
    计算整数 n 的阶乘
    参数:
    n (int): 要计算阶乘的整数
```

```
    返回:
    int: n 的阶乘
    示例:
    >>> calculate_factorial(5)
    120
    """
    if n == 0:
        return 1
    else:
        return n * calculate_factorial(n-1)
```

不推荐的例子中，没有注释解释代码的功能和逻辑。推荐的例子中，注释详细解释了函数的功能、参数、返回值以及使用示例，便于理解和维护。

A.2.1.3 代码风格

示例：

```
# 不推荐的代码风格：不一致的缩进和空格使用
def func1(x,y):
    return x+y
def func2(x, y ):
    return x - y
# 推荐的代码风格：一致的缩进和空格使用，使用代码格式化工具自动格式化代码
def func1(x, y):
    return x + y
def func2(x, y):
    return x - y
```

不推荐的例子中，函数定义的缩进和空格使用不一致。推荐的例子中，函数定义遵循一致的缩进和空格使用规范，提高了代码的一致性和可读性。

A.2.1.4 逻辑单元

示例：

```
# 不推荐的代码：一个函数负责多个逻辑功能
def process_data(data):
    # 数据清洗
    cleaned_data = [x.strip() for x in data if x]
    # 数据分析
```

```
    analysis_result = sum([int(x) for x in cleaned_data])
    return analysis_result
# 推荐的代码：将不同逻辑分离到独立的函数中
def clean_data(data):
    return [x.strip() for x in data if x]
def analyze_data(cleaned_data):
    return sum([int(x) for x in cleaned_data])
def process_data(data):
    cleaned_data = clean_data(data)
    return analyze_data(cleaned_data)
```

不推荐的例子中，一个函数同时处理数据清洗和数据分析，职责不单一。推荐的例子中，将数据清洗和数据分析逻辑分离到独立的函数中，符合单一职责原则，提高了代码的可维护性。

A.2.1.5 模块规则

示例：

```
# 不推荐的代码：模块间直接访问内部数据
class ModuleA:
    def __init__(self):
        self.data = "internal data"


class ModuleB:
    def access_data(self, module_a):
        return module_a.data
# 推荐的代码：使用接口或抽象类定义模块间的交互
class ModuleA:
    def __init__(self):
        self._data = "internal data"
    def get_data(self):
        return self._data
class ModuleB:
    def access_data(self, module_a):
        return module_a.get_data()
```

不推荐的例子中，ModuleB 直接访问了 ModuleA 的内部数据。推荐的例子中，通过

ModuleA 提供的接口 get_data 进行数据访问，增强了模块的可替换性和安全性。

A.2.1.6 函数规则

示例：

```
# 不推荐的代码：使用全局变量
global_var = 10
def add_to_global_var(x):
    global global_var
    global_var += x
    return global_var
# 推荐的代码：避免使用全局变量，使用明确的输入和输出
def add(x, y):
    return x + y
```

不推荐的例子中，函数使用全局变量，增加了代码的复杂性和潜在的错误。推荐的例子中，函数通过明确的输入参数和返回值进行操作，避免了全局变量的使用，提高了代码的可维护性。

A.2.1.7 函数安全

示例：

```
# 不推荐的代码：没有输入验证
def divide(a, b):
    return a / b
# 推荐的代码：添加输入验证
def divide(a, b):
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
        raise ValueError("Inputs must be numbers")
    if b == 0:
        raise ValueError("Division by zero is not allowed")
    return a / b
```

不推荐的例子中，没有对输入进行验证，可能导致运行时错误。推荐的例子中，通过验证输入类型和除零情况，提高了函数的安全性和可靠性。

A.2.1.8 语言错误

示例：

```
# 不推荐的代码：没有进行静态检查
def add(x, y)
```

```
    return x + y
# 推荐的代码：使用静态检查工具检查语法错误
def add(x, y):
    return x + y
```

不推荐的例子中，缺少对语法错误的检查，代码中有明显的语法错误。推荐的例子中，使用静态代码分析工具检查语法错误，确保代码质量。

### A.2.2 错误与异常

#### A.2.2.1 必要错误异常

示例：

```
# 不推荐的代码：未处理可能的异常
def read_file(file_path):
    with open(file_path, 'r') as file:
        return file.read()
# 推荐的代码：捕获并处理必要的异常
def read_file(file_path):
    try:
        with open(file_path, 'r') as file:
            return file.read()
    except FileNotFoundError:
        print(f"Error: {file_path} not found")
    except IOError as e:
        print(f"IO error occurred: {e}")
```

不推荐的例子中，未处理可能的文件读取异常，可能导致程序崩溃。推荐的例子中，通过捕获并处理必要的异常，确保程序的稳定性。

#### A.2.2.2 自定义异常捕获

示例：

```
# 不推荐的代码：使用通用异常
def process_data(data):
    if not data:
        raise Exception("Invalid data")
# 推荐的代码：使用自定义异常
class InvalidDataError(Exception):
    pass
```

```
def process_data(data):
    if not data:
        raise InvalidDataError("Data cannot be empty")
```

不推荐的例子中，使用通用异常 Exception，无法明确具体错误。推荐的例子中，通过自定义异常 InvalidDataError，提高了错误处理的清晰度和代码的可维护性。

#### A.2.2.3 错误信息

```
# 不推荐的代码：提供过多的内部信息
def connect_to_database(connection_string):
    try:
        # 数据库连接逻辑
        pass
    except Exception as e:
        print(f"Error: {e}")
# 推荐的代码：提供有用且安全的错误信息
def connect_to_database(connection_string):
    try:
        # 数据库连接逻辑
        pass
    except Exception:
        print("Error: Failed to connect to the database. Please check your connection settings.")
```

不推荐的例子中，错误信息包含具体异常 e，可能泄露内部实现细节。推荐的例子中，错误信息简洁明了，避免泄露内部实现细节，同时提供了有用的指导信息。

# 附 录 B

（资料性）

## SPDX 格式的 SBOM 清单

SPDX 是一种广泛使用的 SBOM 格式，旨在提供软件组件的透明性和可追溯性。SPDX 格式通过标准化描述软件包、文件、许可证信息及其关系，帮助组织管理软件供应链的安全性和合规性。其核心特点和结构如下：

| 部分 | 字段/属性 | 描述 |
|---|---|---|
| **文档创建信息** | spdxVersion | SPDX 文档版本（如 "SPDX-2.2"）。 |
| | dataLicense | 文档使用的许可证（通常为 "CC0-1.0"）。 |
| | SPDXID | 文档的唯一标识符（如 "SPDXRef-DOCUMENT"）。 |
| | name | 文档名称。 |
| | creationInfo | 包含创建者（creators）和创建时间（created）。 |
| **包信息** | SPDXID | 包的唯一标识符（如 "SPDXRef-Package-1"）。 |
| | name | 包的名称。 |
| | version | 包的版本号。 |
| | downloadLocation | 包的下载位置（URL）。 |
| | checksums | 包的校验和（如 SHA1、SHA256）。 |
| | licenseConcluded | 包的实际使用许可证。 |
| | licenseDeclared | 包声明的许可证。 |
| **文件信息** | SPDXID | 文件的唯一标识符（如 "SPDXRef-File-1"）。 |
| | fileName | 文件的名称。 |
| | checksums | 文件的校验和（如 SHA1、SHA256）。 |
| | licenseConcluded | 文件的实际使用许可证。 |
| **许可证信息** | licenseID | 许可证的唯一标识符（引用 SPDX License List）。 |
| | extractedText | 自定义许可证的文本内容（若未使用标准许可证）。 |
| **关系信息** | spdxElementId | 关系中的主体元素（如 "SPDXRef-Package-1"）。 |
| | relationshipType | 关系类型（如 "CONTAINS"、"DEPENDS_ON"、"DERIVED_FROM"）。 |
| | relatedSpdxElement | 关系中的关联元素（如 "SPDXRef-File-1"）。 |
| **注释信息** | comment | 对文档、包、文件或关系的额外描述或注释。 |

以下是简化的 SPDX SBOM 示例（JSON 格式）：

```
{
  "spdxVersion": "SPDX-2.2",
  "dataLicense": "CC0-1.0",
  "SPDXID": "SPDXRef-DOCUMENT",
  "name": "Example-SBOM",
  "creationInfo": {
    "creators": ["Tool: SPDX-Tool-1.0", "Organization: ExampleOrg"],
    "created": "2023-10-01T00:00:00Z"
  },
  "packages": [
    {
      "SPDXID": "SPDXRef-Package-1",
      "name": "ExamplePackage",
      "version": "1.0.0",
      "downloadLocation": "https://example.com/package",
      "checksums": [
        {
          "algorithm": "SHA256",
          "checksumValue": "abc123..."
        }
      ],
      "licenseConcluded": "MIT",
      "licenseDeclared": "MIT"
    }
  ],
  "files": [
    {
      "SPDXID": "SPDXRef-File-1",
      "fileName": "example.c",
      "checksums": [
        {
          "algorithm": "SHA1",
          "checksumValue": "def456..."
        }
```

```
      ],
      "licenseConcluded": "MIT"
    }
  ],
  "relationships": [
    {
      "spdxElementId": "SPDXRef-Package-1",
      "relationshipType": "CONTAINS",
      "relatedSpdxElement": "SPDXRef-File-1"
    }
  ]
}
```

# 参考文献

[1] GB/T 20158-2006 信息技术 软件生存周期过程 配置管理
[2] GB/T 25069-2022 信息安全技术 术语
[3] GB/T 34943-2017 信息安全技术 C/C++语言源代码漏洞测试规范
[4] GB/T 34944-2017 信息安全技术 Java 语言源代码漏洞测试规范
[5] GB/T 34946-2020 信息安全技术 C#语言源代码漏洞测试规范
[6] GB/T 39412-2020 信息安全技术 代码安全审计规范
[7] GB/T 43698-2024 网络安全技术 软件供应链安全要求
[8] ISO/IEC 5962：2021 信息技术 SPDX 规范