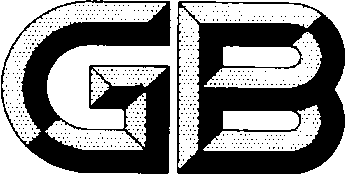
ICS 点击此处添加ICS号

点击此处添加中国标准文献分类号



中华人民共和国国家标准

GB/T XXXXX—XXXX

|  |
| --- |
|  |

信息安全技术 代码安全审计规范

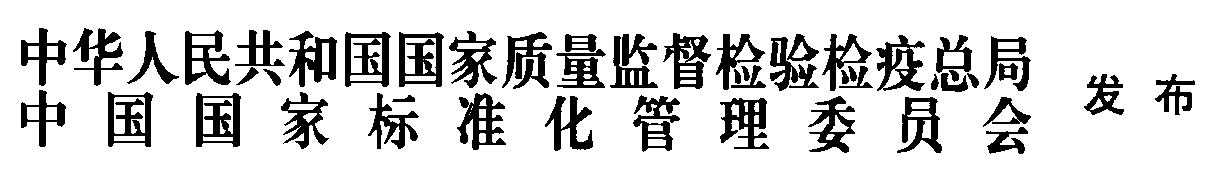
Information security technology—Code security audit specification

点击此处添加与国际标准一致性程度的标识

|  |
| --- |
|  |
|  |

XXXX - XX - XX发布

XXXX - XX - XX实施



目  次

[前言 IV](#_Toc493859664)

[1　范围 1](#_Toc493859665)

[2　规范性引用文件 1](#_Toc493859666)

[3　术语和定义 1](#_Toc493859667)

[4　源代码安全审计概述 2](#_Toc493859676)

[4.1　审计目的 2](#_Toc493859677)

[4.2　审计时机 2](#_Toc493859678)

[4.3　审计人员 2](#_Toc493859679)

[4.4　审计方法 2](#_Toc493859680)

[5　源代码安全审计过程 2](#_Toc493859681)

[5.1　总体流程 3](#_Toc493859682)

[5.2　审计准备 3](#_Toc493859683)

[5.3　审计实施 3](#_Toc493859684)

[5.4　审计报告 4](#_Toc493859685)

[5.5　改进跟踪 4](#_Toc493859686)

[6　源代码安全通用弱点审计列表 4](#_Toc493859687)

[6.1　概述 4](#_Toc493859688)

[6.2　环境与封装 4](#_Toc493859689)

[6.3　API调用 5](#_Toc493859690)

[6.4　指针安全 6](#_Toc493859691)

[6.5　初始化与清理环节 7](#_Toc493859692)

[6.6　错误处理 8](#_Toc493859693)

[6.7　时间和状态 8](#_Toc493859694)

[6.8　通用安全特性 11](#_Toc493859695)

[6.9　数据处理 13](#_Toc493859696)

[6.10　代码质量 17](#_Toc493859697)

[附录A（资料性附录）　源代码安全审计报告 20](#_Toc493859698)

[附录B（资料性附录）　代码示例 21](#_Toc493859699)

[参考文献 57](#_Toc493859700)

前  言

本标准的附录A，附录B为资料性附录。

本标准按照GB/T 1.1-2009规则起草。

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。

本标准由全国信息安全标准化技术委员会（SAC/TC 260）提出并归口。

本标准起草单位：信息安全共性技术国家工程研究中心、中国科学院信息工程研究所、国家保密科技测评中心、北京信息安全测评中心、中国信息安全测评中心、中国电子技术标准化研究院、公安部第三研究所。

本标准的主要起草人：

信息安全技术 代码安全审计规范

1. 范围

本标准规定了源代码安全的审计过程及方法，描述了软件源代码安全弱点的典型审计指标。本标准的审计活动主要对象是源代码，主要针对于源代码层面的安全问题，不包含需求分析、设计、测试、部署配置、运维等方面的安全问题。本标准审计的源代码对象不针对特定语言。

本标准适用于执行源代码安全审计的审计人员，被审计的开发人员，对于自动化审计工具的开发具有一定指导意义。

1. 规范性引用文件

下列文件对于本文件的应用是必不可少的。凡是注日期的引用文件，仅所注日期的版本适用于本文件。凡是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 15272 程序设计语言 C

GB/T 25069 信息安全技术 术语

GB/T 5271.8-2001 信息技术 词汇 第8部分：安全

GB/T 25056-2010 信息安全技术 证书认证系统密码及其相关安全技术规范

GB/T 28452-2012 信息安全技术 应用软件系统通用安全技术要求

GB/T 35273-2017《信息安全技术 个人信息安全规范》

1. 术语和定义

GB/T 15272、GB/T 25069、GB/T 5271.8-2001和GB/T 25056-2010界定的以及下列术语和定义适用于本文件。

3. 1

代码安全审计 Code Security Audit

一种以发现代码安全缺陷和违反代码安全规范为目标的源代码安全性分析。

3.2

隐蔽通道 Covert Channel

为系统中不受[安全策略](https://baike.baidu.com/item/%E5%AE%89%E5%85%A8%E7%AD%96%E7%95%A5)控制的信息泄露路径。按信息传递的方式区分，隐藏通道分为隐蔽存储通道和隐蔽定时通道。

3.3

SQL注入 SQL Injection

利用现有应用程序，将(恶意)的SQL命令注入到后台数据库引擎执行的能力。

3.4

跨站脚本攻击 Cross Site Script

恶意攻击者向Web页面里面插入恶意HTML代码，当用户浏览该页面时，嵌入到Web里面的HTML代码会被执行，从而达到恶意用户的特殊目的。

3.5

缓冲区溢出 Buffer Overflow

程序向一个已分配内存的固定长度缓冲区写数据，使得数据超出了缓冲区的边界。

3.6

线程安全 Thread Safe

多线程访问时采用了加锁机制，当一个线程访问该类的某个数据时进行保护，其他线程不能进行访问直到该线程读取完其他线程才可使用。

3.7

死锁 Deadlock

集合中的每一个进程都在等待只能由本集合中的其他进程才能引发的事件。

3.8

敏感信息 Sensitive Information

由权威机构确定的必须受保护的信息，该信息的泄露、修改、破坏或丢失会对人或事产生可预知的损害。

3.9

口令 Password

用于身份鉴别的秘密的字、短语、数或字符序列。

3.10

错误

系统运行中出现的非预期问题，可能导致系统崩溃或者暂定运行。

3.11

特殊元素 Special Elements

字节、字符或字的序列, 用于特定表达式或语言中分隔数据的不同部分。如CRLF（回车/换行）被用于MIME消息中作为多个头部之间的分隔符，是特殊元素。

3.12

异常 Exception

导致程序中断运行的一种指令流。如果不对异常进行正确的处理，则可能导致程序的中断执行。

1. 代码安全审计概述
   1. 审计目的

代码安全审计的目的在于提高软件系统的安全性，降低风险。具体而言，通过审计发现源代码层面的安全弱点，但不包含软件分析、设计、测试、应用部署等层面的安全弱点。需说明，鉴于安全漏洞形成的综合性和复杂性，源代码安全审计的弱点主要针对源代码层面风险和质量问题，以及规定形成漏洞的各种脆弱性因素。

最终通过审计给出审计报告，列出源代码中针对审计列表的符合性/违规性条目，提出对源代码修订措施和建议。

* 1. 审计时机

审计分为内部审计或外部审计。内部审计通常由公司内部的软件质量保证人员开展，审计的意义倾向于发现及预防安全问题的发生。而外部审计则是由公司外部独立的第三方开展。外部审计需要较多的准备工作和面谈时间，不宜安排很频繁。审计可安排在代码编写完成之后系统集成测试之前。由于资质认证、政策要求等进行的外部审计应提前通知并预留足够时间给开发团队；不应有非提前通知的审计。

内部审计应根据需要安排审计活动尽早发现潜在问题**，**通过多次审计并结合其他软件质量保证手段，来确保软件源代码安全符合要求。内部审计可安排在软件开发生命周期内的不同阶段。

无论是内部审计还是外部审计，历史审计结果可以做为本次审计考虑的因素之一。若审计出的安全问题较多，则后续应根据实际修复问题情况适当增加审计次数。此外在源代码开发过程中，如果有频繁改变源代码开发计划或调整里程碑等异常情况发生时，也应增强审计。

* 1. 审计人员

审计人员的主要工作职能在于收集和呈现信息，具有被审计源代码类型的基础知识，客观地呈现源代码中的问题，不应隐瞒；对源代码等相关内容保守秘密，不对外泄露相关内容。

* 1. 审计方法

审计最常用的方法是将源代码安全弱点形成审计检查列表，对照源代码逐一进行检查。检查列表应根据被审计的具体对象及应用场景进行调整。

考虑到审计内容的复杂性，审计方法建议以工具审计和人工审计相结合，多种手段综合运用为原则。  
首先，采用专业源代码审查工具对代码进行审计，给出审计结果报告，并对审计问题与标准相关审计项逐一用人工核对。由于审计工具的局限性，不可避免存在误报和漏报的问题。针对可能的误报问题，要求通过人工对比审计核查。此外，针对漏报的问题，要求采用多个工具交叉审计。人工审计是对工具审计的必要和有益的补充。人工审计主要针对工具审计漏洞报的问题展开。在人工审计具体实施中，可借助工具对源代码模块、数据流、控制流等逻辑结构进行分析提取，最后逐条比对分析。

在审计实施过程中，要落实多阶段审计工作。如按时间进度或代码编写里程碑划分，按周、月、季度开展审计；按函数、功能模块实施单元审计；按人员分工交叉审计等。

对于审计出来的弱点，可根据弱点的可利用性、造成的影响、弥补代价等因素进行分级排序。

1. 代码安全审计过程
   1. 总体流程

审计过程包括四个阶段：审计准备、审计实施、审计报告、改进跟踪。审计准备阶段，审计员会对待审计的源代码基本情况进行摸底调研，并准备检查清单；审计实施阶段，代码审查会被执行，相关资料会被检查；报告阶段包括审计结果的总结陈述等活动，如有必要进行相关问题的澄清和相关资料说明；改进部分工作由源代码开发团队进行，主要对审计出的问题进行修复。对于安全缺陷代码修改后，会再次进行审计。

* 1. 审计准备
     1. 明确审计目的

审计的应用场景可能包括源代码的采购/外包、软件产品的认证测试、公司源代码安全性自查等，在此类场景下对源代码安全性是否符合规范进行评估。

* + 1. 背景调研

初步了解源代码的应用场景、目标客户、开发内容，开发者遵循的标准、流程，查看源代码最近的状态报告。

* + 1. 熟悉代码

通过阅读代码，了解程序的代码结构、主要的功能模块，以及涉及到的编程语言。

* + 1. 初步制定检查列表

结合前面了解到的信息，根据审计人员身份（外部审计或内部审计）、审计目的（采购外包/认证测试/公司自查）等信息，给出源代码安全的审计要点，初步制定源代码安全的检查列表。检查列表包括审计员计划检查的项以及准备提问的问题列表。

* 1. 审计实施
     1. 审计入场实施

入场实施环节中，审计员和项目成员（关键代码开发人员等）均应参与。审计员介绍审计的主要目标，要面谈的对象和检查的资料。项目人员介绍项目进展、项目关键成员、项目背景、项目的主要功能以及项目的当前状态。

* + 1. 信息收集

信息收集环节通过面谈等方式获得源代码以及相应需求分析文档、设计文档、测试文档等资料。通过文档资料了解源代码的业务逻辑等信息。

* + 1. 代码安全弱点检测

代码安全弱点检测环节是根据制定的源代码安全的检查项，逐一检查是否存在列表项中的安全弱点。

* + 1. 特殊情形审查

在源代码有软件外包/采用开源软件/合作开发情形下，也应对开源软件或外包部分进行了代码安全性检测。

* 1. 审计报告

审计实施结束后，组织现场审计结束会，将初始审计结果提供给被审计项目成员组，提供被审计项目组澄清误解的机会，并允许项目成员提供其他需要补充的信息（如审计员未考虑到的）。项目成员可以立即了解被发现的安全问题并制定修正计划。

根据现场审计结束会上收集到的信息对初始结果进行调整后，审计员准备书面报告。书面报告应突出重大问题，对问题和优点都有所陈述；审计报告包括审计的总体描述，审计结论等部分。审计结论给出每条审计条款的符合/不符合的陈述。审计报告的内容示例见附录A；如果是外部审计，则审计报告的报告对象是组织本次审计的管理人员；如果是内部审计，则按照流程要求进行。

* 1. 改进跟踪

在审计人员完成审计报告后，项目管理组织源代码开发人员采取措施来解决报告中识别的问题。对于审计发现的问题进行修改；对于未修改的应提供理由。对源代码的有效变更进行记录存档。在某些情况下，可通过跟进审计来确认问题是否解决。

1. 代码安全通用弱点审计列表
   1. 概述

通用弱点审计列表是针对常见的桌面应用软件的源代码（以下简称代码）弱点列举的参考列表，特殊应用(如WEB应用、移动应用）的弱点有部分涉及，但不是本章节描述重点。审计时可根据被审计的具体对象及应用场景对本列表进行调整制定。考虑到语言的多样性，以典型的结构化语言（C）和面向对象语言（Java）为目标进行描述。

本章将从11个方面给出审查代码安全的具体条款。其中，环境与封装8条审计条款；API调用4条审计条款；指针安全7条审计条款；初始化与清理环节8条审计条款；错误处理1条审计条款；时间和状态12条审计条款；通用安全特性5类子项，39条审计条款；数据处理33条审计条款；代码质量17条审计条款；总计129条审计条款。

* 1. 环境与封装
     1. 对错误会话暴露数据元素

审计指标：代码应避免对错误会话暴露数据元素。

审计人员应检查代码在应用的不同会话之间是否会发生信息泄露，尤其是在多线程环境下。代码的不规范/规范用法示例见附录B.2.1。

* + 1. 遗留调试代码

审计指标：代码中不应遗留调试代码。

审计人员应检查部署到应用环境的源代码是否含有调试或测试功能的代码，该部分代码是否会造成意外的后门入口。

* + 1. 数据信任边界的违背

审计指标：代码应避免将可信和不可信数据组合在同一结构体中，违背信任边界。

审计人员应检查源代码是否将来自可信源和非可信源的数据混合在同一数据结构体或同一结构化的消息体中，模糊了二者的边界。

* + 1. 类的错误比较

审计指标：在类进行比较时，不应只使用名称比较。

审计人员应检查代码中当多个类具有相同的名字时，是否仅通过名字比较类。只通过类名称进行比较可能会导致使用错误的类。如Java语言中应该同时使用 getClass() 方法和运算符“==”来比较类。代码的不规范/规范用法示例见附录B.2.2。

* + 1. 私有数组类型数据域的不安全操作

审计指标：公开方法中不应返回私有的数组类型的数据域，或将公共的数据赋值给私有数组类型的数据域。

1. 审计人员应检查源代码声明为公开的方法时，是否返回私有数组的引用，如果结果为肯定，数组的数据会有遭到破坏的安全风险。
2. 审计人员应检查源代码是否将公共数据分配给私有数组，如果结果为肯定，则等于给了该数组公众访问权限，数组的数据会有遭到破坏的安全风险。
   * 1. 包含敏感数据类的安全

审计指标：包含敏感数据的类不应可复制和可序列化。

审计人员应检查源代码中包含敏感数据的类的相关行为是否安全，具体要求包括但不限于：

* + - * 1. 应检查源代码中包含敏感数据的类是否可复制，如Java语言中的实现了Clonenable接口使类可复制，包含敏感数据的类不应被复制。
        2. 应检查源代码中包含敏感数据的类是否可实现了序列化接口，使类可序列化，包含敏感数据的类不应可序列化。
    1. 暴露危险的方法或函数

审计指标：不应暴露危险的方法或函数。

审计人员应检查源代码中的应用程序编程接口（API）或与外部交互的类似接口是否暴露了危险方法或函数，他们本应受到适当的限制。危险方法或函数暴露的形式有如下几种：

1. 该方法/函数原本设计为非外部用户使用；
2. 该方法/函数原本设计为部分用户访问，如限定来自基于Internet访问的某网站。

代码的不规范/规范用法示例见附录B.2.3。

* + 1. 存储不可序列化的对象到磁盘

审计指标：不应将不可序列化的对象存储到磁盘。

审计人员应检查（如J2EE框架中）源代码是否试图将不可序列化的对象写到磁盘中，将不可序列化的对象存储到磁盘上可能导致序列化失败和应用程序崩溃。代码的不规范/规范用法示例见附录B.2.4。

* 1. API调用
     1. 参数指定错误

审计指标：函数功能调用应正确指定参数。

审计人员应检查函数/方法调用时参数指定是否正确，是否存在如下情况：

a) 不正确的变量或引用

b) 不正确数量的参数

c) 参数顺序不正确

d) 参数类型不正确

e) 错误的值

以上检查项的任一结果为肯定，则提示存在安全风险。代码的不规范/规范用法示例见附录B.3.1。

* + 1. 堆内存释放问题

审计指标：应避免在释放堆内存前清理不恰当而导致敏感信息暴露。

审计人员应检查代码在释放堆内存前是否采用合适的方式进行信息的清理。如C语言中是否使用realloc()调整存储敏感信息的缓冲区大小，如存在该操作，将存在可能暴露敏感信息的风险。 realloc()函数不是从内存中删除，而通常是将旧内存块的内容复制到一个新的、更大的内存块，这可能暴露给攻击者使用“memory dump”或其他方法来进行读取敏感数据的“堆检查”攻击。代码的不规范/规范用法示例见附录B.3.2。

* + 1. 忽略函数返回值

审计指标：应正确检查函数的返回值，避免忽略函数的返回值。

审计人员应检查代码是否对方法或函数调用的返回值作了检查，如未检查函数的返回值，则提示代码存在无法检测非预期状态的风险。

* + 1. 端口多重绑定

审计指标：不应对同一端口进行多重绑定。

审计人员应检查代码是否有多个套接字绑定到相同端口，从而导致该端口上的服务有被盗或被欺骗的风险。代码的不规范/规范用法示例见附录B.3.3。

* 1. 指针安全
     1. 不兼容的指针类型

审计指标：不应使用不兼容类型的指针来访问变量。

审计人员应检查代码是否使用不兼容类型的指针来访问变量。通过不可兼容类型的指针修改变量可能会导致不可预测的结果。代码的不规范/规范用法示例见附录B.4.1。

* + 1. 利用指针减法确定内存大小

审计指标：应避免使用指针的减法来确定内存大小。

审计人员应检查代码是否采用一个指针减去另一个指针的方式来确定内存大小。如果两个指针不在相同内存块中，那么使用指针的减法来确定内存大小的计算可能会不正确。代码的不规范/规范用法示例见附录B.4.2。

* + 1. 将固定地址赋值给指针

审计指标：不应把固定地址赋值给指针。

审计人员应检查代码是否将一个NULL或0以外的固定地址赋值给指针。将固定地址赋值给指针会降低代码的可移植性，并为攻击者进行注入代码等攻击提供便利。

* + 1. 试图访问非结构体类型的域

审计指标：不应把指向非结构体类型指针强制转换为指向结构类型的指针并访问其字段。

审计人员应检查代码是否将指向非结构体类型的指针，强制转换为指向结构类型的指针并访问其字段，如果结果为肯定，则可能存在内存访问错误或数据损坏的风险。

* + 1. 释放一个不在缓冲区起始位置的指针

审计指标：不应释放不在缓冲区起始位置的指针。

审计人员应检查代码是否释放不在缓冲区起始位置的指针。如果是，则存在应用程序崩溃等风险。

* + 1. 指针偏移越界

审计指标：不应使用越界的指针偏移。

审计人员应检查代码在使用指针时是否存在偏移越界的现象，因指标偏移越界可能会造成访问缓冲区溢出风险。

* + 1. 无效指针使用

审计指标：应避免无效指针的使用。

审计人员应检查代码是否存在使用无效指针的现象，因使用无效指针，可能会产生非预期行为。

* 1. 初始化与清理环节
     1. 资源与变量不安全初始化

审计指标：应避免不安全的资源或变量的初始化。

审计人员应检查源代码是否对资源或变量进行了安全的初始化，具体要求包括但不限于：

a) 应检查源代码是否对关键变量进行初始化,未初始化关键变量易导致系统按非预期值执行，如结果为否定，则提示存在安全风险；

b) 应检查源代码是否采用了不安全或安全性较差的缺省值来初始化内部变量。缺省值通常和产品一起发布，容易被熟悉产品的潜在攻击者获取而带来系统安全风险，如结果为肯定，则提示存在安全风险；

c) 应检查源代码中关键的内部变量或资源是否采用了可信边界外的外部输入值进行初始化，如结果为肯定，则提示存在安全风险。

* + 1. 引用计数的更新不恰当

审计指标：应避免引用计数的更新不恰当。

审计人员应检查源代码中管理资源的引用计数是否正确更新，引用计数更新不正确，可能会导致资源在使用阶段就被过早释放，或虽已使用完毕但得不到释放的安全风险。

* + 1. 过期的文件描述符

审计指标：不应使用过期的文件描述符。

审计人员应检查源代码是否在文件描述符关闭后再次使用或访问它。特定文件或设备的文件描述符被释放后，可被重用，但可能引用了其他的文件或设备。

* + 1. 初始化失败后未退出

审计指标：初始化失败后应退出程序。

审计人员应检查源代码在初始化失败后能否安全退出。

* + 1. 异常状态的处理

审计指标：发生异常时，应恢复对象到先前的状态。

审计人员应检查源代码中当异常发生时对象(尤其是关系到系统安全性的对象)能否维持一致的状态。

* + 1. 启用后台线程前主线程未完成类的初始化

审计指标：不应在初始化类时启用后台线程。

审计人员应检查源代码是否确保主线程完成类的初始化后，再启用后台线程，否则提示系统可能存在类初始化死锁的风险。

* + 1. 发布未完成初始化的对象

审计指标：不应发布部分初始化的对象。

审计人员应检查源代码在多线程环境中，是否有初始化尚未完成的对象被公开，使得其在本线程初始化完成前就可被其他线程引用。

* + 1. 资源不安全清理

审计指标：应避免不安全的资源清理。

审计人员应检查源代码中资源清理部分的相关功能，检查项包括但不限于：

1. 审计人员应检查源代码中当特定错误（如不可恢复逻辑错误）发生时，程序是否在以不可预测的状态继续执行，而导致数据有被损坏的风险，在程序终止时应执行正确的清理动作。代码的不规范/规范用法示例见附录B.5.1。
2. 审计人员应检查源代码在使用资源后是否恰当地执行临时文件或辅助资源的清理，避免清理环节不完整。
   1. 错误处理

审计指标：应恰当进行错误处理。

审计人员应检查源代码中错误处理是否安全，具体要求包括但不限于：

1. 应检查是否对错误进行检查并处理检测到的错误（代码的不规范/规范用法示例见附录B.6.1）；
2. 应检查是否采用标准化的、一致的错误处理机制来处理代码中的错误；
3. 应检查错误发生时，是否提供正确的状态代码或返回值来标示发生的错误（代码的不规范/规范用法示例见附录B.6.2）；
4. 应检查是否确保线程池中正在执行的任务失败后给出提示；
5. 应检查是否对执行文件I/O的返回值进行检查；
6. 应检查是否对函数或操作返回值是否为预期值进行了检查；
7. 应检查是否返回定制的错误页面给用户来预防敏感信息的泄露。

如上检查项的任一结果为否定，则提示存在安全风险。

* 1. 时间和状态
     1. 关键状态数据外部可控

审计指标：应避免关键状态数据被外部控制。

审计人员应检查源代码中是否将与用户信息或软件自身安全密切相关的状态信息，存储在非授权实体都可以访问的地方，如结果为肯定，则系统可能有关键状态数据能被外部访问或篡改的安全风险。代码规范/不规范示例见附录 B.7.1。

* + 1. 隐蔽通道

审计指标：应避免隐蔽通道。

审计人员应检查源代码中是否有隐蔽通道如隐蔽时间通道，如结果为肯定，则提示系统存在信息暴露的安全风险。代码不规范示例见附录 B.7.2。

隐蔽时间通道通过随着时间推移调节某些方面的系统行为来传递信息，接收到信息的程序能够监测系统行为并推断出受保护的信息，从而造成信息暴露。

* + 1. 未加限制的外部可访问锁

审计指标：应对外部可访问锁加以限制，不允许被预期范围之外的实体影响。

审计人员应检查源代码中的锁是否可被预期范围之外的实体控制或影响，如结果为肯定，则系统存在易受到拒绝服务攻击的安全风险。

* + 1. 会话过期机制缺失

审计指标：应制定会话过期机制。

审计人员应检查源代码中会话过程是否存在会话过期机制，如结果为否定，则提示源代码存在保护机制被绕过的风险。代码规范/不规范示例见附录 B.7.3。

* + 1. 将资源暴露给错误范围

审计指标：不应将资源暴露给错误的范围。

审计人员应检查源代码是否将文件和目录等资源暴露给错误的范围，如果存在，则提示源代码存在信息暴露等风险。例如，一个程序本来意图只是向特定的用户提供某些私密文件，防止攻击者访问这些私密文件。如果文件权限是不安全的，那么特定用户以外的其他用户就有可能访问这些文件。

* + 1. 未经控制的递归

审计指标：应避免未经控制的递归。

审计人员应检查源代码是否避免未经控制的递归，未控制递归发生的数量可造成预防资源消耗过多的安全风险。

* + 1. 无限循环

审计指标：执行迭代或循环应恰当地限制循环执行的次数，应避免无限循环。

审计人员应检查源代码中软件执行迭代或循环，是否充分限制循环执行的次数，以避免无限循环的发生或避免攻击者占用过多的资源。

* + 1. 信号错误

审计指标：应正确使用信号处理函数。

审计人员应检查源代码中信号处理函数的使用，具体要求包括但不限于：

1. 应检查源代码中如果信号处理函数不可中断，那么信号处理函数在执行时是否已屏蔽该信号，如果结果为否定，则提示存在安全风险；
2. 应检查源代码中信号处理函数代码序列是否包含非异步安全代码序列，如果结果为肯定，则提示存在安全风险。代码规范/不规范示例见附录 B.7.4。
   * 1. 共享资源的并发安全问题

审计指标：应对共享资源使用正确的并发处理机制。

审计人员应检查源代码中共享资源的使用及并发处理的过程，具体要求包括但不限于：

1. 应检查源代码在多线程环境中对共享数据的访问是否为同步访问，如果结果为否定，则提示存在安全风险；
2. 应检查源代码中线程间共享的对象是否声明正确的存储持续期，如果结果为否定，则提示存在安全风险；代码规范/不规范示例见附录 B.7.5；
3. 应检查源代码中是否在并发上下文中使用不可重入的函数，如果结果为肯定，则提示存在安全风险。
4. 应检查源代码中是否避免了检查时间与使用时间资源冲突。代码不规范示例见附录 B.7.6；
5. 应检查源代码中多个线程中等待彼此释放锁的可执行片段是否避免了死锁情况发生，如果结果为否定，则提示存在安全风险；
6. 审计人员应检查代码对共享资源执行敏感操作时是否检查加锁状态，如果结果为否定，则提示存在安全风险；代码规范/不规范示例见附录B.7.7。
7. 审计人员应检查代码是否将敏感数据存储在没有被锁定或被错误锁定的内存中，（将敏感数据存储于加锁不恰当的内存区域，可能会导致该内存通过虚拟内存管理器被写入到在磁盘上的交换文件中，从而使得数据更容易被外部获取），如果结果为肯定，则提示存在安全风险；
8. 应检查源代码中是否存在互斥体被锁定时删除它们的情况，如果结果为肯定，则提示存在安全风险；
9. 应检查源代码中是否存在关键资源多重加锁，如果结果为肯定，则提示存在安全风险；
10. 应检查源代码中是否存在关键资源多重解锁，如果结果为肯定，则提示存在安全风险；
11. 应检查源代码中是否存在对未加锁的资源进行解锁，如果结果为肯定，则提示存在安全风险；
12. 应检查源代码中在异常发生时是否释放已经持有的锁，如果结果为否定，则提示存在安全风险；
    * 1. 不安全的临时文件

审计指标：应正确使用临时文件

审计人员应检查源代码中临时文件是否安全，预防因临时文件造成的敏感信息泄露，具体要求包括：

1. 应检查源代码中是否创建或使用不安全的临时文件，如果结果为肯定，则提示存在安全风险；
2. 应检查源代码中临时文件是否在程序终止前移除，如果结果为否定，则提示存在安全风险。
3. 应检查源代码中是否在具有不安全权限的目录中创建临时文件，如果结果为肯定，则提示存在安全风险。
   * 1. 符号名称未映射到正确对象

审计指标：符号名称应映射到正确对象。

审计人员应检查源代码中符号名称是否映射到正确对象，如果结果为否定，则提示存在安全风险。

* + 1. Web重定向后执行额外代码

审计指标：web应用不应在重定向后执行额外代码。

审计人员应检查web代码是否存在重定向后执行额外代码，如果结果为肯定，则提示存在安全风险。

* 1. 通用安全特性
     1. 权限、特权与访问控制问题

审计指标：应确保权限、特权的管理安全以及其他访问控制措施的安全。

审计人员应检查源代码中的权限、特权与访问控制功能相关部分，具体要求包括但不限于：

1. 应检查是否缺失认证机制，如果结果为肯定，则提示存在安全风险。
2. 应检查是否缺失授权机制，如果结果为肯定，则提示存在安全风险。
3. 应检查是否存在会话ID确定，如果结果为肯定，则提示存在安全风险。
4. 应检查是否违背最小特权原则，以高于功能所需的特权级别（如根用户或管理员用户）在执行一些操作，如果结果为肯定，则提示存在安全风险（代码的不规范/规范用法示例见附录B.8.1.1）；
5. 应检查放弃特权后，是否检查其放弃是否成功，如果结果为否定，则提示存在安全风险；
6. 应检查是否创建具有正确访问权限的文件，如果结果为否定，则提示存在安全风险（代码的不规范/规范用法示例见附录B.8.1.2）；
7. 应检查是否避免关键资源的不正确权限授予，如果结果为否定，则提示存在安全风险（代码的不规范/规范用法示例见附录B.8.1.3）；
8. 应检查是否确保正确的行为次序，避免在解析与净化处理之前进行授权，如果结果为否定，则提示存在安全风险（代码的不规范/规范用法示例见附录B.8.1.4）；
9. 应检查是否限制过多认证尝试，如果结果为否定，则提示存在安全风险。
10. 应检查是否避免使用单一因素认证机制，如果结果为否定，则提示存在安全风险（代码的不规范/规范用法示例见附录B.8.1.5）；
11. 应检查是否可避免攻击者使用欺骗、候选名、候选路径/通道、或捕获重放攻击等手段绕过身份认证，如果结果为否定，则提示存在安全风险。
12. 应检查是否避免不恰当地信任反向DNS，如果结果为否定，则提示存在安全风险（代码的不规范/规范用法示例见附录B.8.1.6）；
13. 对于客户端/服务器架构的产品，应检查是否避免仅在客户端而非服务器端执行认证，如果结果为否定，则提示存在安全风险。
14. 应检查是否避免过于严格的账户锁定机制（账户锁定保护机制过于严格且容易被触发，就允许攻击者通过锁定合法用户的账户来拒绝服务合法的系统用户），如果结果为否定，则提示存在安全风险；
15. 应检查是否在认证机制中使用口令哈希代替口令，如果结果为否定，则提示存在安全风险；
16. 应检查是否未对信道两端的操作者进行充分的身份认证，或未充分保证信道的完整性，从而允许中间人攻击发生，如果结果为肯定，则提示存在安全风险；
17. 应检查是否避免通信通道源的验证不当，确保请求来自预期源，如果结果为否定，则提示存在安全风险；
18. 应检查是否避免通过用户控制密钥绕过授权机制，如果结果为否定，则提示存在安全风险；
19. 应检查通信信道是否正确指定目的地来预防如下风险：攻击者在目的地伪装成受信任的服务器来窃取数据或引起拒绝服务。如果结果为否定，则提示存在安全风险（代码的不规范/规范用法示例见附录B.8.1.7）。
    * 1. 密码安全问题

审计指标：密码相关功能应符合国家密码相关标准的要求。

审计人员应检查源代码中密码功能是否符合国家密码相关标准的要求，具体要求包括但不限于：

1. 应检查是否避免在加密中重用密钥；
2. 应检查是否避免使用已过期的密钥；
3. 应检查是否避免对敏感数据的明文存储或明文传输；
4. 应检查是否避免使用已被攻破或存在风险的密码学算法；
5. 应检查是否选择被业界专家公认比较健壮的算法，未自行开发或定制私有加密算法并定期确认加密算法是否已经过期；
6. 应检查是否在密码块链接(CBC)加密模式中使用随机初始化向量（代码的不规范/规范用法示例见附录B.8.2.1）；
7. 应检查RSA算法是否结合最优非对称加密填充（OAEP）进行使用。

如上检查项的任一结果为否定，则提示存在安全风险。

* + 1. 随机数安全问题

审计指标：应确保产生安全的随机数。

审计人员应检查源代码是否安全产生随机数，具体审计要求包括但不限于：

1. 应检查是否采用能产生充分信息熵的算法或方案（代码的不规范/规范用法示例见附录B.8.3.1）；
2. 应检查是否避免随机数的空间太小；
3. 应检查是否避免伪随机数生成器（PRNG）每次都使用相同的种子、可预测的种子（如进程ID或系统时间的当前值）或空间太小的种子；
4. 应检查是否避免使用具有密码学弱点的伪随机数生成器（PRNG）用于加密场景；

如上检查项的任一结果为否定，则提示存在安全风险。

* + 1. 数据真实性验证问题

审计指标：应验证数据的起源或真实性，避免接收无效数据。

审计人员应检查源代码是否对数据的真实性进行验证，具体审计要求包括但不限于：

1. 应检查是否有数据源或通信源验证错误（代码的不规范/规范用法示例见附录B.8.4.1）；
2. 应检查是否未验证或不正确验证数据的密码学签名；
3. 应检查是否接受外部混杂在在可信数据中的不可信数据（代码的不规范/规范用法示例见附录B.8.4.2）；
4. 应检查是否缺失或进行不恰当完整性检查（代码的不规范/规范用法示例见附录B.8.4.3）；
5. 应检查安全相关的输入是否仅依赖于加密技术而未进行完整性检查；
6. 应检查是否依赖于文件名或外部提供文件的扩展名而非文件内容进行决策；
7. 应检查是否信任来自系统事件的信息（代码的不规范/规范用法示例见附录B.8.4.4）；
8. 应检查是否依赖未经验证和完整性检查的Cookie进行决策（代码的不规范/规范用法示例见附录B.8.4.5）。

如上检查项的任一结果为肯定，则提示存在安全风险。

* + 1. 个人信息保护

审计指标：应确保个人信息保护。

审计人员应检查源代码是否不恰当地阻止个人隐私信息，在未经明确授权或未经数据相关人的默认同意的情况被访问。不当处理私人信息，可能危及用户隐私，并且可能非法。

* 1. 数据处理
     1. 非预期数据类型处理不当

审计指标：应避免非预期数据类型处理不当。

审计人员应检查代码是否正确处理特定元素并非预期类型时的情形,如预期是数字(0-9)，但被提供字母(A-Z)，源代码是否正确处理。

* + 1. 数据/内存布局问题

审计指标：不应依赖数据/内存布局。

审计人员应检查代码逻辑是否依赖于对协议数据或内存在底层组织形式的无效假设。当平台或协议版本变动时，数据组织形式可能会发生变化从而带来非预期行为。代码规范/不规范示例见附录 B.9.1。

* + 1. 绕过净化和验证

审计指标：应防止以大小写混合的方式绕过净化和验证。

审计人员应检查字符串在查找、替换、比较等操作时，是否存在因大小写问题而被绕过的情况。代码规范/不规范示例见附录 B.9.3.2。

* + 1. 在字符串验证前未进行过滤

审计指标：不应在过滤字符串之前对字符串进行验证。

审计人员应检查对字符串进行验证之前是否存在对该字符串进行过滤，来防止注入类攻击的发生。

* + 1. 条件比较不充分

审计指标：执行比较时不应部分比较或不充分的比较。

审计人员应检查比较条件是否充分，防止不充分比较造成逻辑绕过风险。

* + 1. 泛型和非泛型原始数据类型

审计指标：不应混用具有泛型和非泛型的原始数据类型。

审计人员应检查代码是否存在泛型和非泛型之间原始数据类型的混用现象,应避免泛型和非泛型原始数据类型的混用。代码规范/不规范示例见附录 B.9.2。

* + 1. 字节序使用问题

审计指标：应避免字节序使用不正确。

审计人员应检查代码在跨平台或网络通信处理输入时是否考虑到字节顺序，避免字节序使用不正确。

* + 1. 结构体长度

审计指标：不应将结构体的长度等同于其各个成员长度之和。

审计人员应检查代码是否将结构体的长度等同于其各成员长度之和，不应将结构体长度等同于各成员长度之和。结构对象可能存在无名的填充字符从而造成结构体长度与各个成员长度之和并不相等。代码不规范示例见附录 B.9.3。

* + 1. 数值越界回绕错误

审计指标：应避免越界回绕错误。

审计人员应检查代码是否存在数值赋值超出数值类型范围，应避免赋值越界。代码不规范示例见附录 B.9.4。

* + 1. 除零问题

审计指标：应避免除零错误。

审计人员应检查代码是否存在除零操作，应避免除零错误。代码规范/不规范示例见附录 B.9.5。

* + 1. 边界值检查缺失

审计指标：数值范围比较时，不应遗漏边界值检查。

审计人员应检查代码在进行数值范围比较时，是否遗漏了最小值、最大值等边界值检查。代码规范/不规范示例见附录 B.9.6。

* + 1. 敏感信息暴露

审计指标：应避免敏感信息暴露。

审计人员应检查源代码中是否有敏感信息暴露，重点审查暴露的途径包含但不限于：

1. 通过发送数据导致的信息暴露；
2. 通过数据查询导致的信息暴露；
3. 通过差异性（响应差异性、行为差异性、时间差异性）导致的信息暴露；
4. 通过错误消息导致的信息暴露；
5. 敏感数据的不恰当跨边界移除导致信息暴露；
6. 通过进程信息导致的信息暴露；
7. 通过调试信息导致的信息暴露；
8. 在释放前未清除导致信息暴露；
9. 通过输出流或日志将系统数据暴露到未授权控制的范围；
10. 通过缓存导致的信息暴露；
11. 通过日志文件导致的信息暴露；
12. 通过源代码导致的信息暴露，如测试代码、源代码、注释等；
13. 敏感信息使用GET请求传递导致信息暴露；
14. 备份文件导致信息暴露。

代码规范/不规范示例见附录 B.9.7。

* + 1. 信息丢失或遗漏

审计指标：应避免信息丢失或遗漏。

审计人员应检查代码是否未记录或不恰当记录安全相关信息,应避免信息丢失或遗漏。信息丢失或遗漏导致的常见形式：

1. 截断与安全有关的信息的显示、记录或处理，掩盖攻击的来源或属性；
2. 不记录或不显示信息（如日志），而该信息对确定攻击的来源或性质，或确定行动是否安全具有重要意义；
3. 记录安全相关信息时，使用候选名称而非正式规范的名称导致安全相关信息混淆。
   * 1. 数据结构控制域安全问题

审计指标：应避免对数据结构控制域的删除或意外增加。

审计人员应检查代码关于数据结构控制域的操作：

a)应检查代码是否存在对数据结构控制域的删除而导致系统安全风险。因控制数据被删除，可能会引起严重编程逻辑错误，应避免对数据结构控制域删除。代码不规范示例见附录 B.9.8。

b)应检查代码是否存在对数据结构控制域的意外增加而导致系统安全风险。应避免对数据结构控制域意外增加。代码不规范示例见附录 B.9.9。

* + 1. 内存缓冲区边界操作

审计指标：应避免内存缓冲区边界操作的限制不恰当。

审计人员应检查代码在内存缓冲区边界操作时是否存在越界现象，因内存缓冲区访问越界可能会造成缓冲区溢出漏洞。代码规范/不规范示例见附录 B.9.10。

* + 1. 忽略字符串结尾符

审计指标：应保证字符串的存储具有足够的空间容纳字符数据和结尾符。

审计人员应检查代码字符串的存储空间是否能容纳下结尾符，字符串不以结尾符结束会造成字符串越界访问。代码规范/不规范示例见附录 B.9.11。

* + 1. 对环境变量长度做出假设

审计指标：不应对环境变量的长度做出假设。

审计人员应检查代码在使用环境变量时是否对环境变量的长度做出特定值的假设，因环境变量可由用户进行设置修改，故对环境变量的长度做出假设可能会发生错误。代码规范/不规范示例见附录 B.9.12。

* + 1. 缓冲区复制造成溢出

审计指标：应避免未检查输入数据大小就进行缓冲区复制。

审计人员应检查代码在进行缓冲区复制时，是否存在未对输入数据大小进行检查的现象，因未检查输入数据大小，可能会造成缓冲区溢出。代码不规范示例见附录 B.9.13。

* + 1. 使用错误长度访问缓冲区

审计指标：应避免使用错误的长度值访问缓冲区。

审计人员应检查代码在访问缓冲区时使用长度值是否正确，因使用错误的长度值来访问缓冲区可能会造成缓冲区溢出风险。代码规范/不规范示例见附录 B.9.14。

* + 1. 路径遍历

审计指标：应避免路径遍历。

审计人员应检查代码是否将由外部输入构造的标识文件或目录的路径名，规范化后限制在受限目录，路径遍历会造成非授权访问资源的风险。代码不规范示例见附录 B.9.15、B.9.16。

路径遍历指未将路径名限制在受限目录。

* + 1. 文件访问解析为链接不恰当

审计指标：应避免在文件访问前对链接解析不恰当。

审计人员应检查代码基于文件名访问文件时，是否能恰当避免文件名被识别为链接或者快捷方式。链接或快捷方式可能会解析为意想不到的资源。

* + 1. 非可信环境下命令执行

审计指标：应避免执行的命令或加载的库文件来自非可信源或在非可信环境中执行。

审计人员应检查代码执行的命令或加载的库文件是否来自非可信源，或在非可信环境中执行。如结果为肯定，则应用程序有执行攻击者恶意命令的风险。代码不规范示例见附录 B.9.17。

* + 1. 循环条件输入导致拒绝服务

审计指标：应检查循环条件输入，避免过度循环导致拒绝服务。

审计人员应检查代码是否存在循环条件输入，因过度循环条件输入可能会导致拒绝服务风险。代码规范/不规范示例见附录 B.9.18。

* + 1. 外部控制文件名或路径检查

审计指标：应避免文件名或路径的外部可控制。

审计人员应检查文件系统操作中的文件名或路径是否被外部控制。若允许用户输入来控制或影响在文件系统操作中所使用的路径或者文件名，则攻击者可能会访问或修改系统文件或对于应用程序至关重要的其他非预期文件。

* + 1. 外部控制格式化字符串问题

审计指标：应避免外部控制的格式化字符串。

审计人员应检查代码函数接受格式化字符串作为参数，格式化字符串是否来自外部，如果是，则可能有注入类漏洞风险。代码规范/不规范示例见附录 B.9.19。

* + 1. 对方法或函数参数验证问题

审计指标：应对方法或函数的参数进行验证。

审计人员应检查代码是否存在对方法或函数的参数进行合法性或安全性校验。代码规范/不规范示例见附录 B.9.20。

* + 1. URL重定向

审计指标：不应开放不可信站点的URL重定向。

审计人员应检查代码是否存在URL重定向到不可信站点的现象。因重定向到不可信站点，可能会发生跨站伪造请求漏洞风险。

* + 1. 命令行注入

审计指标：应正确处理命令中的特殊元素。

审计人员应检查代码对利用外部输入来构造命令或部分命令时，是否对其中的特殊元素进行了处理，命令注入通常发生在以下情况中：

1. 数据从非可信源进入到应用程序中；
2. 数据是字符串的一部分，该字符串被应用系统当作命令来执行的；
3. 通过执行这个命令，应用程序为攻击者提供了攻击者不应拥有的权限或能力。
   * 1. SQL执行语句注入

审计指标：应正确处理SQL命令中的特殊元素。

审计人员应检查代码利用用户可控的输入数据构造SQL命令时，是否对外部输入数据中的特殊元素进行处理，如果未处理，那么这些数据有可能被解释为SQL命令而非普通用户的输入数据。攻击者可对此加以利用来修改查询逻辑，从而绕过安全检查或插入可以修改后端数据库的额外语句，可能包括执行操作系统命令。

* + 1. 跨站脚本

审计指标：应避免跨站脚本攻击。

审计人员应检查代码中当用户提供的数据放到在网页中，被送到浏览器进行显示前，是否进行了验证或过滤。

* + 1. 未恰当处理语法形式不完全符合预期规范的输入

审计指标：应恰当处理输入语法形式不完全符合预期规范的输入。

审计人员应检查代码中输入功能的部分：

* + - * 1. 审计人员应检查代码是否正确处理当参数被定义但相关的值缺失情形。
        2. 审计人员应检查当超过预期的的值被输入时，源代码是否正确处理。
        3. 审计人员应检查当输入参数未定义或不支持，源代码是否正确处理。代码不规范示例见附录 B.9.21。
        4. 审计人员应检查代码在函数调用是否对缺失参数的情况进行了处理。代码不规范示例见附录 B.9.22。
        5. 审计人员应检查代码是否处理超过预期数量的参数，应避免未对多余参数做处理。
        6. 审计人员应检查代码是否正确处理未定义的参数，应避免未对未定义参数做处理。
    1. 对输出日志中特殊字符处理问题

审计指标：应对输出日志中的特殊字符进行过滤和验证。

审计人员应检查代码是否对输出日志中的特殊字符做过滤和验证。因对特殊字符未做过滤，可能会造成信息泄露。代码规范/不规范示例见附录 B.9.23。

* + 1. 对HTTP头Web脚本特殊字符处理问题

审计指标：应对HTTP头的Web脚本语法中的特殊字符进行过滤和验证。

审计人员应检查代码是否对HTTP头中的Web脚本特殊字符进行过滤处理。因HTTP头中的Web脚本含有特殊字符，可能会导致浏览器执行恶意脚本。代码规范/不规范示例见附录 B.9.24。

* 1. 代码质量

由于有些代码的质量的好坏会影响软件的安全性，本规范将部分代码质量规范纳入标准范畴。

* + 1. 文件描述符穷尽问题

审计指标：不应导致文件描述符穷尽。

审计人员应检查代码是否存在导致文件描述符穷尽情形。具体检查项包括但不限于：

1. 是否对打开文件描述符未做关闭处理；
2. 是否到达关闭阶段之前，失去对文件描述符的所有引用；
3. 进程完成后是否未关闭文件描述符。
   * 1. 内存未释放

审计指标：应及时释放动态分配的内存。

审计人员应检查代码是否有动态分配的内存使用完毕后未释放导致内存泄漏的情形。内存泄漏可能会导致资源耗尽从而带来拒绝服务的安全风险。代码规范/不规范示例见附录 B.10.1。

* + 1. 对网络消息容量的控制

审计指标：应避免对网络消息容量的控制不充分。

审计人员应检查代码是否控制网络传输流量不超过被允许的值。如果源代码没有机制来跟踪流量传输，系统或应用程序会很容易被滥用于传输大流量(超过了请求值或客户端被允许的值),从而带来拒绝服务的安全风险。

* + 1. 算法复杂度问题

审计指标：应避免算法复杂度攻击。

审计人员应检查代码中算法是否在最坏情况下非常低效，复杂度高，会严重降低系统性能。如果是，则攻击者就可以利用精心编制的操作来触发最坏情况的发生，从而引发算法复杂度攻击。

* + 1. 早期放大问题

审计指标：应遵守正确的行为次序避免早期放大攻击数据。

审计人员应检查代码是否存在允许实体在授权或认证前执行合法但代价高的操作。代码规范/不规范示例见附录 B.10.2。

* + 1. 子进程访问父进程敏感资源问题

审计指标：在调用子进程之前应关闭敏感文件描述符，避免子进程使用这些描述符来执行未经授权的I/O操作。

审计人员应检查代码是否存在调用子进程之前有未关闭敏感文件描述符的情形。当一个新进程被创建或执行时，子进程继承任何打开的文件描述符，如不关闭则可能会造成未经授权的访问。代码规范/不规范示例见附录 B.10.3。

* + 1. 堆空间耗尽问题

审计指标：应限制堆空间的消耗，防止堆空间耗尽。

审计人员应检查代码是否有导致堆空间耗尽的情形，具体检查项包括但不限于：

a) 是否存在内存泄漏；

b) 是否存在死循环；

c) 不受限制的反序列化；

d) 创建大量的线程；

e) 解压一个较大压缩文件。

* + 1. 重复释放资源

审计指标：应避免重复释放资源。

审计人员应检查代码是否存在重复释放资源的情形。重复释放资源可能会造成系统崩溃。

* + 1. 访问已释放内存

审计指标：不应引用或访问已被释放后的内存。

审计人员应检查代码是否存在内存被释放再次被访问的情形。内存被释放后再次访问会出现非预期行为。

* + 1. 内存释放指针赋值问题

审计指标：应在内存释放立即把指针设置为NULL或指向另一个合法的对象。

审计人员应检查代码中内存释放后指针是否未置空或指向另一个合法对象。

* + 1. 内存管理函数成对调用

审计指标：应调用匹配的内存管理函数。

审计人员应检查代码调用的内存管理函数是否匹配，如malloc/free来分配或删除资源。当内存管理函数不匹配时，可能带来内存损坏或程序崩溃等风险。

* + 1. 条件语句缺失默认情况问题

审计指标：switch等条件语句中不应缺失默认情况。

审计人员应检查代码switch等条件语句中是否存在缺失默认情况的情形。

* + 1. 无法执行的死代码问题

审计指标：不应包含无法执行的死代码

审计人员应检查代码是否存在无法执行的死代码。

* + 1. 返回栈上变量地址问题

审计指标：不应返回栈上的变量地址

审计人员应检查代码在函数中是否存在返回栈变量地址的情形。因栈变量在函数调用结束后就会被释放，再使用该变量地址时会出现意想不到的结果。代码不规范示例见附录 B.10.4。

* + 1. 可达断言

审计指标：不应包含可达断言。

审计人员应检查代码在非测试版本中是否包含可达断言。若服务器处理多个并发连接，而断言在其中某个连接中发生，就会导致所有其他连接断掉，从而导致拒绝服务。

* + 1. 实现不一致函数问题

审计指标：不应使用具有不一致性实现的函数或字符。

审计人员应检查代码是否存在使用了在不同版本具有不一致实现的函数或字符。因使用在不同操作系统或不同版本实现不一致的函数，可能导致代码被移植到不同环境时改变行为。代码不规范示例见附录 B.10.5。

* + 1. 表达式永真或永假问题

审计指标：不应出现表达式永真或永假代码。

审计人员应检查代码是否存在表达式逻辑永真或永假代码的情形。

1. （资料性附录）  
   源代码安全审计报告
   1. 概述

本附录给出了第5章源代码安全审计过程中的审计报告的内容介绍。

* 1. 报告内容

源代码审计报告至少应报告以下内容：

* + 1. 审计总体信息
  + 审计日期
  + 审计团队成员信息
  + 当前开发阶段
  + 源代码的信息应包括但不限于：
    - 被审计源代码的版本号
    - 源代码语言类型
    - 源代码总行数等
    1. 发现的问题
  + 该版本源代码发现的异常情况描述
  + 可能造成的重大后果
    1. 总结
  + 审计结果汇总：
    - 多少条审计条款符合，
    - 多少条审计条款不符合，
    - 不符合的审计条款编号：
  + 改进建议或后续审计的日期

1. （资料性附录）  
   代码示例
   1. 概述

本附录给出了第6章源代码安全审计要求中各审计条款源代码不规范用法示例和规范用法示例。

* 1. 环境与封装
     1. 代码应避免对错误会话暴露数据元素代码示例

不规范用法（C语言）示例：

以下是获取学生信息结构体中的学生姓名，然后将结果打印出来。

struct stu {

char ID[20];

char name[20];

};

char name[20];

void getname( struct stu student )

{

name = student.name;

printf( "name = %s\n", name );

…

}

虽然此方法在单用户环境中会很好地工作，但是如果在同一时间内有两个用户访问该函数，两个请求处理线程会以以下方式交替：

线程1：分配“Dick”给name。线程2：分配给“Jane”给name。

线程1：打印“Jane, thanks for visiting!”。线程2：打印“Jane, thanks for visiting!”。从而将第一个用户的名字显示为第二个用户的名字。

规范用法（C语言）示例：

将结构体请求参数的值存储在局部变量中而不是全局变量中，保证线程的安全性。

struct stu {

char ID[20];

char name[20];

};

void getname( struct stu student )

{

char name[20];

memset( name, 0, sizeof(name) );

name = student.name;

printf( "name = %s\n", name );

…

}

* + 1. 在类进行比较时，不应只使用名称代码示例

不规范用法（java语言）示例：

如下代码示例将对象auth的类名与字符串“com.application.auth.DefaultAuthenticationHandler”进行比较。

// Determine whether object auth has required/expected class object

if (auth.getClass().getName().equals("com.application.auth.DefaultAuthenticationHandler")) {

…

}

规范用法（java语言）示例：

这个符合规则的方案将类对象auth与当前类装载器装载的类对象进行了比较，而不仅仅对类名称进行比较。

// Determine whether object auth has required/expected class name

if (auth.getClass() == com.application.auth.DefaultAuthenticationHandler.class) {

…

}

* + 1. 不应暴露危险的方法或函数代码示例

不规范用法（java语言）示例：

如下代码示例中，方法removeDatabase()将删除输入参数中指定名称的数据库。

public void removeDatabase(String databaseName) {

try {

Statement stmt = conn.createStatement();

stmt.execute("DROP DATABASE " + databaseName);

} catch (SQLException ex) {…}

}

上面示例中的方法是被声明为public，因此会被暴露给应用程序中的任何类。在应用程序内删除一个数据库被视为一个关键的操作，应该限制访问这些潜在危险方法。

规范用法（java语言）示例：

这个符合规则的方案通过声明方法为private来完成，只将它暴露给封闭的类，如下面代码所示：

private void removeDatabase(String databaseName) {…}

* + 1. 不应将不可序列化的对象存储到磁盘上代码示例

不规范用法（java语言）示例：

如下的代码示例中，Customer Entity JavaBean为业务应用程序提供到数据库的客户信息的访问。Customer Entity JavaBean作为会话作用域对象用于将客户信息返回给会话EJB。

@Entity

public class Customer {

private String id;

private String firstName;

private String lastName;

private Address address;

public Customer() {

}

public Customer(String id,String firstName,String lastName) {…}

//GET && SET

…

}

Customer Entity JavaBean是不可序列化对象，当J2EE容器试图将对象写入系统的时候，这可能导致序列化失败和应用程序崩溃。

规范用法（java语言）示例：

该方案，会话作用域的对象实现Serializable接口，以确保对象被正确序列化。

public class Customer implements Serializable {…}

* 1. API调用
     1. 函数功能调用应正确指定参数代码示例

不规范用法（C语言）示例：

使用malloc()、 calloc()或 realloc()函数分配内存时不宜执行零长度的分配，使用malloc(), calloc()或 realloc()分配零字节内存的结果是由编译器定义的。

如下示例中，分配了一个整形动态数组，用于存储size个元素。但是，如果size为0，调用malloc(size)可能会返回对一个长度为0的内存块的引用，而不是返回一个null指针。当（非空）数据复制到这个位置时，就会发生缓冲区溢出。

size\_t size;

/\* Initialize size, possibly by user-controlled input \*/

int \*list = (int \*)malloc(size);

if (list == NULL) {

/\* Handle allocation error \*/

}

else {

/\* Continue processing list \*/

}

规范用法（C语言）示例：

为了保证绝不会把0作为size参数传递给malloc()，应该对size参数进行检查，保证它具有正值。

size\_t size;

/\* Initialize size, possibly by user-controlled input \*/

if (size == 0) {

/\* Handle error \*/

}

int \*list = (int \*)malloc(size);

if (list == NULL) {

/\* Handle allocation error \*/

}

/\* Continue processing list \*/

* + 1. 应避免在释放堆内存前清理不恰当而导致敏感信息暴露代码示例

不规范用法（C语言）示例：

如下代码在包含敏感数据的缓冲区中调用realloc()函数。

cleartext\_buffer = get\_secret();...

cleartext\_buffer = realloc(cleartext\_buffer， 1024);

...

scrub\_memory(cleartext\_buffer,1024);

试图从内存中擦除敏感数据，但使用了realloc()，所以数据的副本仍然被暴露在最初为cleartext\_buffer分配的内存中。

* + 1. 不应对同一端口进行多重绑定代码示例

不规范用法（C语言）示例：

如下代码将服务器套接字绑定到端口21，允许服务器监听该端口的流量。

void bind\_socket( void )

{

int server\_sockfd;

int server\_len;

struct sockaddr\_in server\_address;

/\*unlink the socket if already bound to avoid an error when bind() is called\*/

unlink( "server\_socket" );

server\_sockfd = socket( AF\_INET, SOCK\_STREAM, 0 );

server\_address.sin\_family = AF\_INET;

server\_address.sin\_port = 21;

server\_address.sin\_addr.s\_addr = htonl( INADDR\_ANY );

server\_len = sizeof(struct sockaddr\_in);

bind( server\_sockfd,( struct sockaddr \* ) &s1, server\_len );

}

这段代码可能会导致两个服务器绑定套接字到相同的端口，从而收到彼此的流量。这可能被攻击者用来窃取其他进程（如安全FTP服务器）的报文。

* 1. 指针安全
     1. 不应使用不兼容类型的指针来访问变量代码示例

不规范用法（C语言）示例：

该示例中，通过一个int \*来实现了一个float类型对象的值自加。然而，通过非兼容类型的指针来访问一个对象是未定义行为。

#include <stdio.h>

void f( void )

{

if ( sizeof(int) == sizeof(float) )

{

float f = 0.0f;

int \*ip = (int \*) &f;

(\*ip)++;

printf( "float is %f\n", f );

}

}

规范用法（C语言）示例：

#include <float.h>

#include <math.h>

#include <stdio.h>

void f( void )

{

float f = 0.0f;

f = nextafterf( f, FLT\_MAX );

printf( "float is %f\n", f );

}

* + 1. 应避免使用指针的减法来确定内存大小代码示例

不规范用法（C语言）示例：

下面的示例包含用于确定一个链表中的节点数的方法大小。该方法被传递一个指针指向链表的头。

语言：C和C++

struct node {

int data;

struct node \* next;

};

/\*

\* Returns the number of nodes in a linked list from

\* the given pointer to the head of the list.

\*/

int size( struct node\* head )

{

struct node \* current = head;

struct node \* tail;

while ( current != NULL )

{

tail = current;

current = current->next;

}

return(tail - head);

}

/\* other methods for manipulating the list \*/

...

然而，该方法将创建一个指针，指向列表的末尾，并通过尾指针减去头指针，来确定列表中的节点数目。由于不能保证指针存在于相同的内存区域中，因此使用指针减法这种方式可能返回不正确的结果。

规范用法（C语言）示例：

在此解决方案中使用计数器确定列表中的节点数，如下面的代码所示。

...

int size( struct node\* head )

{

struct node \* current = head;

int count = 0;

while ( current != NULL )

{

count++;

current = current->next;

}

return(count);

}

* 1. 初始化与清理环节
     1. 在程序终止时应执行正确的清理动作代码示例

不规范用法（java语言）示例：

这个不符合规则的代码示例创建了一个新文件，输出了一些文本，并且使用Runtime.exit()直接退出。因而，文件可能会关闭而不包含需要写入的文本。

public class CreateFile {

public static void main(String[] args) throws FileNotFoundException {

final PrintStream out = new PrintStream(new BufferedOutputStream(new FileOutputStream("foo.txt")));

out.println("hello");

Runtime.getRuntime().exit(1);

}

}

规范用法（java语言）示例：

这个符合规则的方案直接在退出之前关闭文件。

public class CreateFile {

public static void main(String[] args) throws FileNotFoundException {

final PrintStream out = new PrintStream(new BufferedOutputStream(new FileOutputStream("foo.txt")));

try {

out.println("hello");

} finally {

out.close();

}

Runtime.getRuntime().exit(1);

}

}

* 1. 错误处理
     1. 应避免未对错误进行检查或未处理检测到的错误代码示例

不规范用法（C语言）示例：

下面的代码尝试为一个字符分配内存,调用 malloc，使用if语句来检查malloc 函数是否失败。

foo = malloc( sizeof(char) ); /\* the next line checks to see if malloc failed \*/

if ( foo == NULL )

{

/\* We do nothing so we just ignore the error. \*/

}

条件语句从表示失败的malloc成功地检测到NULL返回值，但是它不做任何动作来处理问题。

规范用法（C语言）示例：

if 代码块应包含这样的语句：尝试解决该问题，通知用户错误已发生并继续处理或执行一些清理并妥善地终止程序。下面的示例通知用户：malloc 函数没有分配所需的内存资源，并返回一个错误代码。

foo = malloc( sizeof(char) ); /\* the next line checks to see if malloc failed \*/

if ( foo == NULL )

{

printf( "Malloc failed to allocate memory resources" );

return(-1);

}

* + 1. 不应返回错误的状态编码代码示例

不规范用法（java语言）示例：

这个不符合规则的代码示例中，Java servlet中遇到IOException 时返回HTTP 404 状态代码。404 代码通常表明一个不存在的资源，在这种情况下会有一些误导

try {

// something that might throw IOException

…

} catch (IOException ioe) {

response.sendError(SC\_NOT\_FOUND);

}

规范用法（java语言）示例：

这个符合规则的方案对程序中的错误返回正确的错误值，防止意外情况的发生。

try {

// something that might throw IOException

…

} catch (IOException ioe) {

response.sendError(SC\_INTERNAL\_SERVER\_ERROR);

}

* 1. 时间和状态
     1. 应避免关键状态数据被外部控制代码示例

不规范用法（C语言）示例：

#define DIR "/restricted/directory"

char cmd[500];

sprintf(cmd, "ls -l %480s", DIR);

/\* Raise privileges to those needed for accessing DIR. \*/

RaisePrivileges(…);

system(cmd);

DropPrivileges(…);

…

程序本意是执行一个命令，显示一个受限目录的内容，然后执行其他操作。假定它是通过运行setuid权限来绕过操作系统的权限检查。代码看起来似乎没有任何问题，因为目录和命令都被设置为固定值，攻击者无法修改。攻击者只能看DIR目录下的内容，这符合程序设计的预期。最后，程序员也限制了提升特权后代码的执行范围。

但是程序并没有修改“PATH”环境变量，因此如下攻击将会生效：设置PATH为用户可控制的目录，如“my/dir”；创建恶意程序命名为“ls”, 并且将该程序放在”/my/dir”；运行上面的代码；当system( )被执行到的时候，shell会查询”PATH”来找到”ls”程序；结果，找到了恶意程序“/my/dir/ls” . 找不到”/bin/ls”, 因为“PATH”不是“/bin/”；最终结果是，代码提升权限，执行了恶意程序。

不规范用法（java语言）示例：

程序员通常信任隐藏字段的内容，认为用户不会查看它们或操作其内容。攻击者则会推翻这些假设。他们将检查写入隐藏字段的值，更改它们，或使用攻击数据替换这些内容。

这个不符合规则的代码示例：

Hidden hidden = new Hidden(element);

如果隐藏字段包含敏感信息，那么就会像捕捉该页面的其余部分一样捕捉这些信息。这会导致敏感信息在用户不知情的情况下在浏览器缓存中被隐藏起来。

规范用法（java语言）示例：

这个符合规则的方案清理数据中的敏感信息

element.removeSensitiveData();

Hidden hidden = new Hidden(element);

* + 1. 应避免隐蔽时间通道代码示例

不规范用法（C语言）示例：

int validate\_password( char \*actual\_pw, char \*typed\_pw )

{

if ( strlen( actual\_pw ) != strlen( typed\_pw ) )

{

return(1);

}

int len = strlen( actual\_pw );

for ( int i = 0; i < len; i++ )

{

if ( actual\_pw[i] != typed\_pw[i] )

{

return(1);

}

}

return(0);

}

在上面代码中，攻击者可以观察到密码正确的用户身份验证需要多长时间。当攻击者尝试自己的值时，他可以先尝试各种长度的字符串。当他找到一个正确字符串的长度时，计算将需要更长的时间，因为循环至少要运行一次。此外，攻击者可能会逐个破解密码的字符，因为当他猜对一个字符的值时，比他猜错时将需要更长的计算时间。通过几百次的猜测这样的攻击可以突破即使是最复杂的密码。

* + 1. 应有session过期机制代码示例

不规范用法（java语言）示例：

会话持续时间越长，攻击者危害用户账户的机会就越大。当会话处于活动状态时，攻击者可能会强力攻击用户的密码、破解用户的无线加密密钥或者通过打开的浏览器强占会话。如果创建大量的会话，较长的会话超时时间还会阻止系统释放内存，并最终导致拒绝服务。

这个不符合规则的代码示例设置setMaxInactiveInterval为负值，使会话保持无限期的活动状态。

HttpSession sesssion = request.getSession(true);

sesssion.setMaxInactiveInterval(-1);

规范用法（java语言）示例：

设置session的有效期在一个安全的时间内，如设置为30min。

HttpSession sesssion = request.getSession(true);

sesssion.setMaxInactiveInterval(1800);

* + 1. 信号处理函数不应包含非异步安全代码序列代码示例

不规范用法（C语言）示例：

下面代码在信号处理程序中调用了free()函数，并通过log\_message()函数调用了fputs ()函数，这两个C语言标准库函数都不是异步安全的。

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

enum { MAXLINE = 1024 };

char \*info = NULL;

void log\_message( void )

{

fputs( info, stderr );

}

void handler( int signum )

{

log\_message();

free( info );

info = NULL;

}

int main( void )

{

if ( signal( SIGINT, handler ) == SIG\_ERR )

{

/\* Handle error \*/

}

info = (char \*) malloc( MAXLINE );

if ( info == NULL )

{

/\* Handle Error \*/

}

while ( 1 )

{

/\* Main loop program code \*/

log\_message();

/\* More program code \*/

}

return(0);

}

规范用法（C语言）示例：

信号处理程序应该尽可能简洁，理想情况下应该是无条件地设置一个标志，然后返回。这个规范性用例设置类型为volatile sig\_atomic\_t的标志并返回；log\_message()和free()函数直接从main()中调用：

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

enum { MAXLINE = 1024 };

volatile sig\_atomic\_t eflag = 0;

char \*info = NULL;

void log\_message( void )

{

fputs( info, stderr );

}

void handler( int signum )

{

eflag = 1;

}

int main( void )

{

if ( signal( SIGINT, handler ) == SIG\_ERR )

{

/\* Handle error \*/

}

info = (char \*) malloc( MAXLINE );

if ( info == NULL )

{

/\* Handle error \*/

}

while ( !eflag )

{

/\* Main loop program code \*/

log\_message();

/\* More program code \*/

}

log\_message();

free( info );

info = NULL;

return(0);

}

* + 1. 应用正确的存储持续期声明线程间共享的对象代码示例

不规范用法（C语言）示例：

下面代码将变量val的地址传递给一个子线程，由其打印出来。该变量具有自动存储持续期。根据执行的顺序，子线程可能在父线程中的生命期之后引用它。这可能导致子线程访问无效的内存位置。

#include <threads.h>

#include <stdio.h>

int child\_thread( void\* val )

{

int \*res = (int \*) val;

printf( "Result: %d\n", \*res );

return(0);

}

void create\_thread( thrd\_t \*tid )

{

int val = 1;

if ( thrd\_success != thrd\_create( tid, child\_thread, &val ) )

{

/\* Handle error \*/

}

}

int main( void )

{

thrd\_t tid;

create\_thread( &tid );

if ( thrd\_success != thrd\_join( tid, NULL ) )

{

/\* Handle error \*/

}

return(0);

}

规范用法（C语言）示例：

将变量val 声明为静态变量，其生命周期是整个程序的执行期。因此，它可以被任何线程安全地访问。

#include <threads.h>

#include <stdio.h>

int child\_thread( void \*val )

{

int \*result = (int \*) val;

printf( "Result: %d\n", \*result );

return(0);

}

void create\_thread( thrd\_t \*tid )

{

/\* 声明 val 为静态变量 \*/

static int val = 1;

if ( thrd\_success != thrd\_create( tid, child\_thread, &val ) )

{

/\* Handle error \*/

}

}

int main( void )

{

thrd\_t tid;

create\_thread( &tid );

if ( thrd\_success != thrd\_join( tid, NULL ) )

{

/\* Handle error \*/

}

return(0);

}

* + 1. 应避免检查时间与使用时间(TOCTOU)资源冲突代码示例

不规范用法（C语言）示例：

下面的代码检查一个文件，然后更新其内容。

struct stat \*sb;

…

lstat( "…" sb ); /\* it has not been updated since the last time it was read \*/

printf( "stated file\n" );

if ( sb->st\_mtimespec == … )

{

print( "Now updating things\n" );

updateThings();

}

文件可能在检查与遍历目录数（lstat）期间已经被更新了，尤其是printf 具有延迟。

* + 1. 应避免加锁检查缺失代码示例

不规范用法（C语言）示例：

下面的函数将尝试获取锁在共享资源上执行操作。

void f( pthread\_mutex\_t \*mutex )

{

pthread\_mutex\_lock( mutex );

/\* 访问共享资源 \*/

pthread\_mutex\_unlock( mutex );

}

代码不检查pthread\_mutex\_lock()返回值是否正确,导致未定义行为。

规范用法（C语言）示例：

int f( pthread\_mutex\_t \*mutex )

{

int result;

result = pthread\_mutex\_lock( mutex );

if ( 0 != result )

return(result);

/\* 访问共享资源 \*/

return(pthread\_mutex\_unlock( mutex ) );

}

* 1. 通用安全特性
     1. 权限、特权与访问控制问题
        1. 不应违背最小特权原则代码示例

不规范用法（C语言）示例：

下面的代码调用了chroot()函数，将应用程序限制在APP\_HOME下的文件系统子集，以防止攻击者通过程序访问位于其他地方的未经授权的文件。然后，代码打开一个用户指定的文件并处理文件的内容。

chroot( APP\_HOME );

chdir( "/" );

FILE\* data = fopen( argv[1] "r+" );

…

在打开文件之前，把应用程序的操作限制在主目录之内是一个相对安全的方法。然而，不使用一些非零值对函数 setuid() 进行调用，意味着应用程序仍在使用没有必要的 root 权限进行操作。任何由攻击者对应用程序实施的成功盗取都会导致发生权限扩大的攻击，因为所有的恶意操作都将以超级用户的权限执行。如果应用程序把权限等级降低到一个非 root 用户，则会显著减少很多潜在的破坏。

规范用法（C语言）示例：

为减少此类程序带来的系统安全风险, 在使用完root 权限后应立即放弃root 权限，恢复调用者原有权限。

chroot( APP\_HOME );

chdir( "/" );

/\* Do some important stuff \*/

FILE\* data = fopen( argv[1], "r+" );

…

setuid( old\_uid ); /\* 降低权限 \*/

/\* Do some non privileged stuff. \*/

…

* + - 1. 应创建具有正确访问权限的文件代码示例

不规范用法（C语言）示例：

fopen（）函数并不允许程序员显式地指定文件访问权限。在这个代码例子中，如果调用fopen（）创建了一个新文件，文件的权限取决于编译器。

char \*file\_name;

FILE \*fp;

/\* Initialize file\_name \*/

fp = fopen( file\_name, "w" );

if ( !fp )

{

/\* Handle error \*/

}

规范用法（C语言）示例：

函数fopen\_s()可以用于创建具有受限权限的文件，如果一个文件被创建时，模式字符串的第一个字符不是‘u’，但是得到底层系统的支持，这个文件的权限应该是防止系统中的其他用户访问这个文件。如果这个文件被创建时，模式字符串的第一个字符是‘u’，当这个文件被关闭时，它应该具有系统默认的文件访问权限。

u字符可以看成是“umask”的缩写，意味着他们的权限与使用fopen（）创建的文件的权限相同。在下面代码示例中，u模式字符被省略，因此这个文件用受限的权限打开（不管umask是什么值）。

char \*file\_name;

FILE \*fp;

/\* Initialize file\_name \*/

errno\_t res = fopen\_s( &fp, file\_name, "wx" );

if ( res != 0 )

{

/\* Handle error \*/

}

* + - 1. 应避免关键资源的不正确权限授予代码示例

不规范用法（C语言）示例：

下面的代码在创建文件并写入文件"Hello world"之前，将进程的 umask 设置为 0。

#define OUTFILE "hello.out"

umask( 0 );

FILE \*out;

/\* Ignore CWE-59 (link following) for brevity \*/

out = fopen( OUTFILE "w" );

if ( out )

{

fprintf( out "hello world!\n" );

fclose( out );

}

在 UNIX 系统上运行此程序后， 运行"ls -l"命令可能会返回以下输出：

-rw-rw-rw- 1 username 13 Nov 24 17：58 hello.out

"rw-rw-rw-"字符串指示所有者（ower）、 组（group）和所有用户（world）都能读取文件和写入文件。

规范用法（C语言）示例：

创建的文件umask一般使用027权限，即文件权限是750，rxwr-x---(所有者全部权限，属组读写，其它人无权限)。

#define OUTFILE "hello.out"

umask( 027 );

FILE \*out;

/\* Ignore CWE-59 (link following) for brevity \*/

out = fopen( OUTFILE, "w" );

if ( out )

{

fprintf( out, "hello world!\n" );

fclose( out );

}

* + - 1. 应确保正确的行为次序，避免在解析与净化处理之前进行授权代码示例

不规范用法（C语言）示例：

例如在检测授权之前，如果web服务器不完全解析请求的URLs，攻击者可能会绕过授权保护字符串。

* + - 1. 应避免使用单一因素认证机制代码示例

不规范用法（C语言）示例：

下面的代码示例中，若给定的密码与存储的密码匹配，用户就可以登录。此代码没有采用多个身份认证方法，如果攻击者可以盗取或猜出用户的密码，他们就可以得到账户的整个访问权限。

unsigned char \*check\_passwd( char \*plaintext )

{

ctext = simple\_digest( "sha1" plaintext strlen( plaintext ) … );

/\* Login if hash matches stored hash \*/

if ( equal( ctext secret\_password() ) )

{

login\_user();

}

}

规范用法（C语言）示例：

使用多个相互独立的认证机制，这样可以确保即使其中一个方法被攻破，系统自身仍然是安全的。

* + - 1. 应避免不恰当地信任反向DNS代码示例

不规范用法（C语言）示例：

下面代码使用 DNS 查找以决定入站请求是否来自受信宿主，若攻击者损害 DNS 缓存，则可获得受信任的状态。

struct hostent \*hp; struct in\_addr myaddr;

char \* tHost = "trustme.example.com";

myaddr.s\_addr = inet\_addr( ip\_addr\_string );

hp = gethostbyaddr( (char \*) &myaddr sizeof(struct in\_addr)AF\_INET );

if ( hp && !strncmp( hp->h\_name tHost sizeof(tHost) ) )

{

trusted = true;

} else {

trusted = false;

}

规范用法（C语言）示例：

执行适当的正向和反向 DNS 查找，以检测 DNS 欺骗（DNS spoofing）。

struct hostent \*hp;

struct in\_addr myaddr;

char \* tHost = "trustme.example.com";

myaddr.s\_addr = inet\_addr( ip\_addr\_string );

hp = gethostbyaddr( (char \*) &myaddr, sizeof(struct in\_addr), AF\_INET );

if ( hp && !strncmp( hp->h\_name, tHost, sizeof(tHost) ) )

{

/\* DNS正向解析 \*/

if ( strcmp( myaddr.s\_addr, nslookup( hp->h\_name ) ) == 0 )

{

/\* DNS反向解析 \*/

if ( strcmp( tHost, reverse\_nslookup( myaddr.s\_addr ) ) == 0 )

{

trusted = true;

}

}

} else {

trusted = false;

}

/\* DNS正向解析 \*/

char \* nslookup( char \*hostname )

{

.…

/\* 执行shell指令nslookup hostname 返回ip \*/

…

}

/\* DNS反向解析 \*/

char \* reverse\_nslookup( char \*ip )

{

…

/\* 执行shell指令nslookup -qt=ptr ip 返回域名 \*/

…

}

* + - 1. 通信信道应正确指定目的地代码示例

不规范用法（C语言）示例：

软件可能不正确地解析电子邮件或IP地址，并发送敏感数据到非预期的目的地。

* + 1. 密码学安全问题
       1. 在密码块链接(CBC)加密模式中应使用随机初始化向量代码示例

不规范用法（C语言）示例：

下面的代码中，加密数据时使用CBC模式，初始化向量IV总为0，使得生成的密码文本可预测，容易受字典攻击。

EVP\_CIPHER\_CTX ctx;

char key[EVP\_MAX\_KEY\_LENGTH];

char iv[EVP\_MAX\_IV\_LENGTH];

RAND\_bytes( key b );

memset( iv 0 EVP\_MAX\_IV\_LENGTH );

EVP\_EncryptInit( &ctx EVP\_bf\_cbc() key iv );

* + 1. 随机数安全问题
       1. 应采用能产生充分信息熵的算法或方案代码示例

不规范用法（C语言）示例：

void generateSessionID( int usrID )

{

srandom( usrID )

return(random() );

}

这段代码的功能是给用户 session产生唯一的随机的ID。 因为伪随机数生成器的种子永远是用户ID，所以产生的session ID 将会永远相同。攻击者因此可以预测用户session ID 并劫持该session.

* + 1. 数据真实性验证问题
       1. 应避免数据源或通信源验证错误代码示例

不规范用法（C语言）示例：

对于微软Windows 98, NT 4.0, 2000, and XP系统，域名解析器的缺省配置中将QueryIpMatching 设置为0，这导致Windows接受主机的DNS更新而不进行查询，使得远程攻击者可以实施DNS缓存中毒攻击。

* + - 1. 应避免接受外部混杂在在可信数据中的不可信数据代码示例

不规范用法（C语言）示例：

在微软Windows NT和Windows 2000系统中，信任域接收来自可信域的授权信息，但没有验证可信域是否对所有的SID具有权限，这使得攻击者可以在信任域获得域管理员权限,通过注入非可信域的SID到来自可信域的授权数据中实现。

* + - 1. 应避免缺失或不恰当完整性检查代码示例

不规范用法（C语言）示例：

软件使用的传输协议，不包括在传输过程中验证数据完整性的机制，如校验和。如果完整性检查值或"校验和"在协议被省略，就没有办法确定传输中的数据是否已损坏。协议中校验和的缺少，将会使数据的第一层应用层的数据检查缺失。

规范用法（C语言）示例：

添加相应大小的校验和到协议中，确保收到的数据在解析和使用前进行了验证；确保协议设计中的校验和被正确计算并在消息发送前添加到每条消息中。

* + - 1. 不应信任系统事件数据代码示例

不规范用法（java语言）示例：

此示例代码中，当授权的用户激活按钮，会打印出秘密信息：

public void actionPerformed(ActionEvent e) {

if (e.getSource() == button) {

System.out.println("print out secret information");

}

}

这个不符合规则的代码示例不会尝试阻止未经授权的用户激活按钮。即使在应用程序的UI界面，按钮对未经授权的用户呈现出来的是非功能性的，攻击者可以很容易发送一个假按钮按下事件到应用程序窗口，并公开秘密信息。

* + - 1. 安全决策不应依赖未经验证和完整性检查的Cookie代码示例

不规范用法（java语言）示例：

这个不符合规则的代码示例从浏览器cookie读取一个值来确定用户的角色。

Cookie[] cookies = request.getCookies();

for (int i =0; i< cookies.length; i++) {

Cookie c = cookies[i];

if (c.getName().equals("role")) {

userRole = c.getValue();

}

}

攻击者很容易修改本地存储cookie中的“role”值，这允许了特权升级。

* 1. 数据处理
     1. 应防止以大小写混合的方式绕过净化和验证代码示例

不规范用法（java语言）示例：

这个不符合规则的代码示例：一个XSS净化方法使用安全的等价来替代了用户提供的输入的script标签。

public String preventXSS(String input, String mask) {

return input.replaceAll("script", mask);

}

只有当输入为“script”的时候，代码才会运行而当输入为“SCRIPT”或者“ScRiPt”时并不会通过这样的方法来过滤，这就允许了XSS攻击。

* + 1. 不应混用具有泛型和非泛型的原始数据类型代码示例

不规范用法（java语言）示例：

这个不符合规则的代码示例可以编译，但会产生未经检查的警告，因为List.add()方法使用的是原始数据类型而不是参数类型（addToList()方法中的list参数。

class ListUtility {

private static void addToList(List list, Object obj) {

list.add(obj); // Unchecked warning

}

public static void main(String[] args) {

List<String> list = new ArrayList<String> ();

addToList(list, 1);

System.out.println(list.get(0)); // Throws ClassCastException

}

}

上述代码执行时会抛出异常。之所以发生这个情况，并不是因为一个List<String>会接受一个Integer，而是因为由list.get(0)返回的值不是一个正确的类型（是一个Integer而不是String类型）。或者说，代码会抛出异常，而这个异常发生在实际导致错误的操作执行之后的某一个时刻，这样会使调试变得复杂。

规范用法（java语言）示例：

这个符合规则的方案通过改变addToList()方法的签名加强正确的类型检查来强化类型安全性。

class ListUtility {

private static void addToList(List<String> list, String str) {

list.add(str); // No warning generated

}

public static void main(String[] args) {

List<String> list = new ArrayList<String>();

addToList(list, "1");

System.out.println(list.get(0));

}

}

* + 1. 不应将结构体的长度等同于其各个成员长度之和代码示例

不规范用法（C语言）示例：

下面代码假设buffer结构的长度等于其各个成员的长度之和，但由于结构填充的缘故，buffer结构的实际长度可能更大。

enum { buffer\_size = 50 };

struct buffer {

size\_t size;

char bufferC[buffer\_size];

} buff;

/\* … \*/

void func( const struct buffer \*buf )

{

/\*

\* Incorrectly assumes sizeof(struct buffer) =

\* sizeof(size\_t) + sizeof(buff.bufferC)

\*/

struct buffer \*buf\_cpy = (struct buffer \*) malloc(

sizeof(size\_t) + sizeof(buff.bufferC)

);

if ( buf\_cpy == NULL )

{

/\* Handle malloc() error \*/

}

/\*

\* With padding, sizeof(struct buffer) may be greater than

\* sizeof(size\_t) + sizeof(buff.bufferC), causing some data

\* to be written outside the bounds of the memory allocated.

\*/

memcpy( buf\_cpy, buf, sizeof(struct buffer) );

/\* … \*/

free( buf\_cpy );

}

规范用法示例：

考虑结构的填充字节之后，可以防止这些类型的错误。

enum { buffer\_size = 50 };

struct buffer {

size\_t size;

char bufferC[buffer\_size];

} buff;

/\* … \*/

void func( const struct buffer \*buf )

{

struct buffer \*buf\_cpy =

(struct buffer \*) malloc( sizeof(struct buffer) );

if ( buf\_cpy == NULL )

{

/\* Handle malloc() error \*/

}

/\* … \*/

memcpy( buf\_cpy, buf, sizeof(struct buffer) );

/\* … \*/

free( buf\_cpy );

}

* + 1. 应避免超界或下溢回绕错误代码示例

不规范用法（C语言）示例：

#include <stdio.h>

#include <stdbool.h>

main( void )

{

int i;

i = -2147483648;

i = i - 1;

return(0);

}

这个示例有整数下溢的问题，i的值已是最低负值，所以减去1后，新的i值是2147483647。

* + 1. 应避免除零错误代码示例

不规范用法（C语言）示例：

下面示例中的函数把两个数值进行相除而没有验证输入作为分母的值是否为零。这将导致一个试图被零除的错误。

double divide( double x, double y )

{

return(x / y);

}

规范用法（C语言）示例：

下列代码通过验证分母的输入值，来确保除零错误不会发生，这避免导致意想不到的结果。

double divide( double x, double y )

{

if ( 0 == y )

{

/\* 处理错误并返回 \*/

}

return(x / y);

}

* + 1. 数值范围比较时，不应遗漏边界值检查代码示例

不规范用法（C语言）示例：

int getValueFromArray( int \*array, int len, int index )

{

int value;

/\*

\* check that the array index is less than the maximum

\* length of the array

\*/

if ( index < len )

{

/\* get the value at the specified index of the array \*/

value = array[index];

}

/\*

\* if array index is invalid then output error message

\* and return value indicating error

\*/

else {

printf( "Error: index is: %d\n", index );

value = -1;

}

return(value);

}

此方法仅验证给定的数组索引小于数组的最大长度，但不会检查最小值。这将允许负值被接受作为输入的数组索引，导致越界读取，并且可能允许访问敏感的内存。

规范用法（C语言）示例：

应检查输入的数组索引以验证数组在所需的最大值和最小范围内。在此示例中if语句应该修改为包括最小范围检查，如下所示。

…

/\*

\* check that the array index is within the correct

\* range of values for the array

\*/

if ( index >= 0 && index < len )

{

/\* … \*/

}

…

* + 1. 应避免信息泄露代码示例

不规范用法（C语言）示例1：

下面代码示例是检查登录用户名密码是否正确的提示信息。

void func( char \* username, char \* password )

{

if ( strcmp( username, check\_username ) == 0 )

{

if ( strcmp( password, check\_password ) == 0 )

{

printf( "Login Successful\n" );

}else{

printf( "Login Failed - incorrect password\n" );

}

}else{

printf( "Login Failed - unknown username\n" );

}

}

在上面的代码中，当用户输入错误的用户名但密码正确和当输入用户名是正确的，但密码是错误的情况，有不同的消息。 这种差异使攻击者了解到登录功能状态，攻击者可以通过尝试不同的值，直到不正确的密码返回一个有效的用户名。

规范用法（C语言）示例1：

void func( char \* username, char \* password )

{

if ( strcmp( username, check\_username ) == 0 && strcmp( password, check\_password ) == 0 )

{

printf( "Login Successful\n" );

}else{

printf( "Login Failed\n" );

}

}

安装程序一般会对错误条件有一个响应页。错误页面不应披露有关敏感操作的成功或失败的信息。例如，登录页面不应该确认登录名正确和密码不正确。攻击者尝试随机账户名称可能猜出其中一部分。确认该账户存在会使登录页面更容易受到暴力攻击。

不规范用法（C语言）示例2：

下面的代码将打印path 环境变量到标准错误流中：

char\* path = getenv("PATH");

…

sprintf(stderr， "cannot find exe on path %s\n"， path);

规范用法（C语言）示例2：

应用程序永远不应生成内部如堆栈跟踪和错误消息的详细信息，除非该信息由最终用户直接提交到终端用户不可查看的日志。所有的错误消息文本被写入到日志文件之前应该是 HTML 实体编码的，以防止针对日志查看器的潜在攻击。

* + 1. 应避免对数据结构控制域的删除代码示例

不规范用法（C语言）示例：

下面的代码示例创建一个null结尾字符串并打印内容，字符串foo为9个字符和一个null终结符提供空间，但是10个字符被写入foo结果，字符foo没有null终止，调用printf()时，会产生不可预知的、可能危险的结果。

char \*foo;

int counter;

foo = calloc( sizeof(char) \* 10 );

for ( counter = 0; counter != 10; counter++ )

{

foo[counter] = 'a';

printf( "%s\n" foo );

}

* + 1. 应避免对数据结构控制域的意外增加代码示例

不规范用法（C语言）示例：

char \*foo;

foo = malloc( sizeof(char) \* 5 ) ;

foo[0] = 'a';

foo[1] = 'a';

foo[2] = atoi( getc( stdin ) ) ;

foo[3] = 'c';

foo[4] = '\0';

printf( "%c %c %c %c %c \n" foo[0] foo[1] foo[2] foo[3] foo[4] );

printf( "%s\n" foo ) ;

第一个print语句将打印每个字符，由空格分隔。然而，如果一个非整数通过 getc 从 stdin 中读取，那么atoi将不进行转换并返回 0。当 foo作为字符串时，在字符 foo [2] 中的0 将作为 NULL 终止符，foo [3] 将永远不会被打印。

* + 1. 应避免内存缓冲区边界操作的限制不恰当代码示例

不规范用法（C语言）示例：

int getValueFromArray( int \*array int len int index )

{

int value;

/\* check that the array index is less than the maximum// length of the array \*/

if ( index < len )

{

/\* get the value at the specified index of the array \*/

value = array[index];

}

/\*

\* if array index is invalid then output error message

\* and return value indicating error

\*/

else {

printf( "Value is： %d\n" array[index] );

value = -1;

}

return(value);

}

代码只验证给定的数组索引是否小于数组的最大长度但不检查最小值。这将允许接受一个负值作为输入数组索引，从而导致边界外读取并且可能允许访问敏感内存。

规范用法（C语言）示例：

应检查输入数组索引来验证数组所需的最大和最小范围内。在下面代码中if语句应被修改为包括一个最小范围检查，如下所示。

…

/\*

\* check that the array index is within the correct

\* range of values for the array

\*/

if ( index >= 0 && index < len )

{

/\* … \*/

}

…

* + 1. 应保证字符串的存储具有足够的空间容纳字符数据和结尾符代码示例

不规范用法（C语言）示例：

下面代码中的循环把数据从 src 复制到 dest，但是，null终止符可能被不正确地写到dest尾部之后的字节中，

#include <stddef.h>

void copy( size\_t n, char src[n], char dest[n] )

{

size\_t i;

for ( i = 0; src[i] && (i < n); ++i )

{

dest[i] = src[i];

}

dest[i] = '\0';

}

规范用法（C语言）示例：

规范性代码是对循环终止条件进行修改，在dest的尾部添加null终止符。

#include <stddef.h>

void copy( size\_t n, char src[n], char dest[n] )

{

size\_t i;

for ( i = 0; src[i] && (i < n - 1); ++i )

{

dest[i] = src[i];

}

dest[i] = '\0';

* + 1. 不应对环境变量的长度作出假设代码示例

不规范用法（C语言）示例：

下面代码把getenv()返回的字符串复制到一个固定长度的缓冲区。

void f()

{

char path[PATH\_MAX]; /\* Requires PATH\_MAX to be defined \*/

strcpy( path, getenv( "PATH" ) );

/\* Use path \*/

}

平台假设$PATH是已经定义了的，定义PATH\_MAX并确保paths并没有超过PATH\_MAX的特性。环境变量$PATH并不需要比PATH\_MAX字符少，如果它超过了PATH\_MAX字符，可能会导致一个缓冲区溢出。因此，如果$PATH是未定义的，那么strcpy()将试图来解引用一个空指针。

规范用法（C语言）示例：

在下面代码中，strlen()函数用于计算字符串的长度，并动态分配所需要的空间。

void f()

{

char \*path = NULL;

/\* Avoid assuming $PATH is defined or has limited length \*/

const char \*temp = getenv( "PATH" );

if ( temp != NULL )

{

path = (char \*) malloc( strlen( temp ) + 1 );

if ( path == NULL )

{

/\* Handle error condition \*/

} else {

strcpy( path, temp );

}

/\* Use path \*/

}

}

* + 1. 应避免未检查输入数据大小就进行缓冲区复制代码示例

不规范用法（C语言）示例：

下列代码没有确认字符串所指向的数据大小是否适合本地缓冲区，就使用危险的strcpy()函数盲目地对数据进行了复制。如果攻击者可以影响字符串参数的内容，就可能会导致缓冲区溢出。

void manipulate\_string( char\* string )

{

char buf[24];

strcpy( buf, string );

/\* … \*/

}

规范用法（C语言）示例：

对输入缓冲进行长度验证。

void manipulate\_string( char\* string )

{

char buf[24];

if ( strlen( string ) >= 24 || strlen( string ) <= 0 )

{

/\*错误处理\*/

}else{

strcpy( buf, string );

}

/\* … \*/

}

* + 1. 应避免使用错误的长度值访问缓冲区代码示例

不规范用法（C语言）示例：

代码使用方法strncpy，将源字符串复制到dest字符串中。

…

char source[21] = "the character string";

char dest[12];

strncpy(dest, source, sizeof(source)-1);

…

strncpy函数中源字符串用sizeof确定字符数，这会导致缓冲区溢出，因为源字符串的长度大于目的缓冲区dest字的大小。

规范用法（C语言）示例：

应根据目的缓冲区的大小决定字符串的复制长度。

…

char source[21] = "the character string";

char dest[12];

strncpy(dest, source, sizeof(dest)-1);

…

* + 1. 相对路径遍历代码示例

不规范用法（C语言）示例：

软件使用外部输入来构造一个应该在一个受限制目录中的路径名。但是软件并没有规范化类似“..”的序列，导致路径名被解析到受限路径外，序列主要形式如 ‘../filedir’、 ‘/../filedir’、‘dir/../filename’、‘dir/../../filename’、‘..\filedir’ 、‘\..\filename’、 ‘\dir\..\filename’、 ‘\dir\..\..\filename’、 ‘…’（三个点号），‘….’（多个点号）、‘….//’、‘…/…//’等。

* + 1. 绝对路径遍历代码示例

不规范用法（C语言）示例：

软件使用外部输入来构造一个应该在一个受限制目录中的路径名，但是软件并没有规范化类似“/abs/path”的绝对路径序列(如/absolute/pathname/here' 、'\absolute\pathname\here' 、'C：dirname' 、'\\UNC\share\name\'(WindowsUNC共享)等)，导致路径名被解析到受限路径外。

* + 1. 应避免执行的命令或加载的库文件来自非可信源或在非可信环境中执行代码示例

不规范用法（C语言）示例：

下面取自于含有特殊权限的应用程序的代码，其使用的是一个注册表项去确定它被安装的目录，并且根据指定目录的一个相对路径来加载一个库文件。

…

RegQueryValueEx( hkey "APPHOME"

0 0 (BYTE \*) home & size );

char\* lib = (char \*) malloc( strlen( home ) + strlen( INITLIB ) );

if ( lib )

{

strcpy( lib home );

strcat( lib INITCMD );

LoadLibrary( lib );

}

…

这段代码允许攻击者修改一个注册表值去指定一个包含恶意版本的INITLIB的不同路径，从而允许攻击者通过提高了的应用程序权限去加载任意的库。因为程序没有验证从环境中读取的值，所以如果一个攻击者能够控制APPHOME的值，他就可以欺骗应用程序让其运行恶意代码。

* + 1. 应检查循环条件输入，避免过度循环导致拒绝服务代码示例

不规范用法（C语言）示例：

void iterate( int n )

{

int i;

/\* 未对n进行检查 \*/

for ( i = 0; i < n; i++ )

{

foo();

}

}

void iterateFoo()

{

unsigned int num;

scanf( "%u" & num );

iterate( num );

}

规范用法（C语言）示例：

void iterate( int n )

{

int i;

/\*对n 的值进行检查 \*/

if （ n > MAX\_VALUE

{

return ；

}

for ( i = 0; i < n; i++ )

{

foo();

}

}

void iterateFoo()

{

unsigned int num;

scanf( "%u" & num );

iterate( num );

}

* + 1. 应避免外部控制的格式化字符串代码示例

不规范用法（C语言）示例：

下面的代码使用snprintf()将命令行参数复制缓冲区。

int main( int argc, char \*\*argv )

{

char buf[128];

/\* … \*/

snprintf( buf, 128, argv[1] );

}

此代码使攻击者能够查看到堆栈的内容并使用包含格式化指令序列的命令行参数修改堆栈内容。

规范用法（C语言）示例：

确保传递不被用户控制的静态格式字符串给函数，同时传给函数相应数目的参数。

* + 1. 应对方法或函数参数进行验证代码示例

不规范用法（java语言）示例：

在这个不符合规则的代码示例中，setState()和useState()没有验证它们的参数。恶意的调用程序可能会传递给库一个非法的state，从而破坏库的状态，并导致出现漏洞。

private Object myState = null;

// Sets some internal state in the library

void setState(Object state) {

myState = state;

}

// Performs some action using the state passed earlier

void useState() {

// Perform some action here

}

当内部的状态包含或是指向敏感的系统关键数据时，这样的漏洞是特别严重的。

规范用法（java语言）示例：

这个符合规则的方案对参数进行了验证，同时在使用内部的状态前也进行了检查。这种方式促进了程序执行的一致性并减少了潜在的漏洞。

private Object myState = null;

// Sets some internal state in the library

void setState(Object state) {

if (state == null) {

// Handle null state

}

// Defensive copy here when state is mutable

if (isInvalidState(state)) {

// Handle invalid state

}

myState = state;

}

// Performs some action using the state passed earlier

void useState() {

if (myState == null) {

// Handle no state (e.g., null) condition

}

…

}

* + 1. 应恰当处理未定义值代码示例

不规范用法（java语言）示例：

这个不符合规则的代码示例中，一个地址参数被读取和清除空格。

String address = request.getParameter("address").trim();

如果地址参数的值是null，servlet将会抛出NullPointerException。

* + 1. 应避免对缺失参数处理不当代码示例

不规范用法（C语言）示例：

void some\_function( int foo, ... )

{

int a[3];

int i;

va\_list ap;

va\_start( ap, foo );

for ( i = 0; i < sizeof(a) / sizeof(int); i++ )

{

a[i] = va\_arg( ap, int );

printf( "\n %d \n", a[i] );

}

va\_end( ap );

}

int main( int argc, char \*argv[] )

{

some\_function( 17, 42 );

}

每次运行该函数时，尽管只传递两个参数，它也将打印出栈上两个参数之后4个字节的内容。

* + 1. 应对输出日志中的特殊字符进行过滤和验证代码示例

不规范用法（java语言）示例：

下面的代码示例在接收到非法请求的时候，会记录用户的用户名。这时没有执行任何输入净化。

if (loginSuccessful) {

logger.severe("User login succeeded for: " + username);

} else {

logger.severe("User login failed for: " + username);

}

如果没有净化，那么可能会出现日志注入攻击。

规范用法（java语言）示例：

这个符合规则的方案在登录之前会净化用户名输入，从而防止注入攻击。

if (!Pattern.matches("[、]+", username)) {

// Unsanitized username

logger.severe("User login failed for unauthorized user");

} else if (loginSuccessful) {

logger.severe("User login succeeded for: " + username);

} else {

logger.severe("User login failed for: " + username);

}

* + 1. 应对HTTP头的web脚本语法中的特殊字符进行过滤和验证代码示例

不规范用法（java语言）示例：

这个不符合规则的代码示例中,用户控制的数据被添加到HTTP header和返回给客户端。由于数据没有经过净化恶意用户可能会注入导致客户端浏览器执行的危险脚本标签。

response.addHeader(HEADER\_NAME, untrustedRawInputData);

规范用法（java语言）示例：

这个符合规则的方案对储存到HTTP header中的数据进行验证，若含有危险字符则不返回客户端。

if(Pattern.matches("[A-Za-z0-9\_]+", untrustedRawInputData)){

response.addHeader(HEADER\_NAME, inputData);

}

* 1. 代码质量
     1. 应及时释放动态分配的内存代码示例

不规范用法（C语言）示例：

如果read()调用不返回预期的字节数，则会泄露已分配的内存块：

char\* getBlock( int fd )

{

char\* buf = (char \*) malloc( BLOCK\_SIZE );

if ( !buf )

{

return(NULL);

}

if ( read( fd, buf, BLOCK\_SIZE ) != BLOCK\_SIZE )

{

return(NULL);

}

return(buf);

}

规范用法（C语言）示例：

char\* getBlock( int fd )

{

char\* buf = (char \*) malloc( BLOCK\_SIZE );

if ( !buf )

{

return(NULL);

}

if ( read( fd, buf, BLOCK\_SIZE ) != BLOCK\_SIZE )

{

free( buf );

return(NULL);

}

return(buf);

}

* + 1. 应遵守正确的行为次序避免早期放大攻击数据代码示例

不规范用法（C语言）示例：

以下示例是打印用户请求的指定文件的内容。

bool printFile( char \*username, char \*filename )

{

char \* file = NULL;

/\* 读取文件 \*/

file = file\_get\_content( filename );

/\* 检查是否有权限 \*/

if ( access( filename, R\_OK ) == 0 )

{

printf( "%s", file );

return(true);

}else{

printf( "You are not authorized to view this file\n" );

}

return(false);

}

该代码首先读取指定文件到内存中，然后查看用户是否有访问权限，如果用户被授权查看该文件，则打印该文件的内容。如果用户不被允许查看文件，则文件到内存中的读取则是浪费资源，并且是不必要的。

规范用法（C语言）示例：

bool printFile( char \*username, char \*filename )

{

char \* file = NULL;

/\* 检查是否有权限 \*/

if ( access( filename, R\_OK ) == 0 )

{

/\* 读取文件 \*/

file = file\_get\_content( filename );

printf( "%s", file );

return(true);

}else{

printf( "You are not authorized to view this file\n" );

}

return(false);

}

* + 1. 在调用子进程之前应关闭敏感文件描述符，避免子进程使用这些描述符来执行未经授权的I/O操作代码示例

不规范用法（C语言）示例：

以下示例没有遵循规则，因为fopen（）调用打开的文件在func（）函数返回之前没有关闭：

int func( const char \*filename )

{

FILE \*f = fopen( filename, "r" );

if ( NULL == f )

{

return(-1);

}

/\* … \*/

return(0);

}

规范用法（C语言）示例：

以下代码示例中，f指针指向的文件在返回到调用者之前关闭：

int func( const char \*filename )

{

FILE \*f = fopen( filename, "r" );

if ( NULL == f )

{

return(-1);

}

/\*… \*/

if ( fclose( f ) == EOF )

{

return(-1);

}

return(0);

}

* + 1. 不应返回栈上的变量地址代码示例

不规范用法（C语言）示例：

下面的错误代码示例返回一个栈地址。

char\* getName() {

char name[STR\_MAX];

fillInName(name);

return name;

}

* + 1. 不应使用具有不一致性实现的函数或字符代码示例

不规范用法（C语言）示例：

以下代码实现清屏的功能，但是只适用于Linux环境下，如果在Windows环境下就会出错。

int main(){

// Linux环境下

system("clear");

// Windows环境下

//system("cls");

return 0;

}

规范用法（C语言）示例：

在设计阶段尽量考虑可能需要移植的环境，然后在程序编写当中加以考虑。

#ifdef \_\_GNUC\_\_

#include<unistd.h>

char \*clear = "clear";

#endif

#ifdef \_MSC\_VER

#include<windows.h>

char \*clear = "cls";

#endif

int main(){

system(clear);

return 0;

}

参 考 文 献

[1] GB/T 28169-2011 嵌入式软件 C语言编码规范

[2] ISO/IEC 9899:2011 Information technology -- Programming languages – C

[3] SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition)

[4] The CERT Oracle Secure Coding Standard for Java

[5] Common Weakness Enumeration, A Community-Developed Dictionary of Software Weakness Types

[7] GJB 5369-2005 航天型号软件C语言安全子集

[8] GB/T 20983-2007 信息安全技术 网上银行系统信息安全保障评估准则

[9] OWASP Secure Coding Practices - Quick Reference Guide

[10] ISO/IEC TS 17961 2013 Information technology -- Programming languages, their environments and system software interfaces -- C secure coding rules

[11] NASA-GB-A301 Software Quality Assurance Audits Guidebook

[12] NASA-GB-8719.13 NASA Software Safety Guidebook

[13] QJ 3128-2001 航天型号软件开发规范

[14] HS/T 28-2010 海关信息系统信息安全风险评估规范

[15] HS/T 33-2011 NET安全编码规范

[16]GB/T 17901.1-1999 信息技术 安全技术 密钥管理 第1部分：框架

[17]GB/T 17901.2-1999 信息技术 安全技术 密钥管理 第2部分：采用对称技术的机制

[18]GB/T 17901.3-1999 信息技术 安全技术 密钥管理 第3部分：采用非对称技术的机制

[19]GB/T 17964-2008 信息安全技术 分组密码算法的工作模式

[20]GB/T 25056-2010 信息安全技术 证书认证系统密码及其相关安全技术规范