天融信
TOPSEC
证券代码:002212

相融共创 赋能未来

融信译站

开发人员软件验证的最低标准指南

天融信科技集团

Barbara Guttman
Vadim Okun
软件及系统部
信息技术实验室

2021 年 7 月

摘要

2021 年 5 月 12 日关于改善国家网络安全的第 14028 号行政命令(EO)，要求美国国家标准与技术研究院（NIST）在 60 天内提出软件测试的最低标准。本文档描述了软件验证技术的 11 项建议，并提供了有关技术的补充信息和进一步信息的参考文献。它建议采用以下技术：

- 威胁建模，寻找设计级的安全问题
- 自动化测试的一致性和最小化的人工参与
- 静态代码扫描，查找高级错误
- 寻找可能的硬编码秘密的启发式工具
- 使用内置检查和保护
- “黑盒”测试用例
- 基于代码的结构测试用例
- 历史测试用例
- 模糊测试
- Web 应用扫描器（如果适用的话）
- 地址包含的代码（库、包、服务）

本文件没有考虑处理软件验证的全部内容，只是推荐广泛适用并形成最低标准的相关技术。
该文件由 NIST 与美国国家安全局（NSA）协商制定。此外，我们通过向 2021 年 6 月初举行的 NIST 行政命令研讨会提交的论文、研讨会上的讨论以及与几位提交人的跟进情况，收到了许多外部组织的意见。

关键词

软件保障；验证；测试；静态分析；模糊测试；代码评审；软件安全。

免责声明

任何提及商业产品或提及商业组织的资料仅供参考；这并不意味着 NIST 推荐或认可，也不意味着所提及的产品一定是用于该目标的最佳产品。

附加信息

有关 NIST 网络安全计划、项目和出版物的更多信息，请访问计算机安全资源中心。有关 NIST 和信息技术实验室（ITL）的其他工作的信息也可在此处获得。
本文件是在美国国家标准与技术协会由联邦政府雇员在执行公务期间撰写。根据《美国法典》第 17 篇第 105 节，本文件不受版权保护，属于公共领域。
如果此文件被使用而表示致谢，我们将不胜感激。

致谢

作者特别感谢 Fay Saydjari 促进了讨论范围，感谢 Virginia Laurenzano 将 DevOps 研究与评估（DORA）原则融入了报告和其他材料，感谢 Larry Wagoner 的大量贡献和评价，感谢 Steve Lipner 的评论和建议，感谢 David A.Wheeler 的大量修订和建议，以及 Aurelien M. Delaitre、William Curt Barker、Murugiah Sapucaya、Karen Scarfone 和 Jim Lyle 的许多贡献。
感谢以下人员审查各种规范、标准、指南和其他材料：Jessica Fitzgerald-McKay、Hialo Muniz 和 Yann Prono；首字母缩略词、词汇表和其他内容，则要感谢：Matthew B. Lanigan、Nhan L. Vo、William C、Totten 和 Keith W. Beatty。
感谢所有向 2021 年 6 月的研讨会提交了适合我们领域的观点文件的人。
在 NIST 和国家安全局(NSA)工作人员的每周电话会议上，更多的人分享了他们的专业知识和见解，使本文件受益匪浅：Andrew White, Anne West, Brad Martin, Carol A. Lee, Eric Mosher, Frank Taylor, George Huber, Jacob DePriest, Joseph Dotzel, Michaela Bernardo, Philip Scherer, Ryan Martin, Sara Hlavaty 和 Sean Weaver。NIST 的 Kevin Stine 也参加了。
同时也感谢 Walter Houser 和 Hialo Muniz 的贡献。

翻译声明：

本文由天融信科技集团翻译整理，原文来自 NIST 公开网站，翻译为公益性质，仅供信息安全产业相关研究人员、管理人员参考，如有错漏敬请指正。

所有注册商标或商标均属于其各自的组织。

目 录

1.简介 **6**
1.1 概述 6
1.2 主旨 6
1.3 范围 6
1.4 验证各方面如何相互关联 7
1.5 文件大纲 8
2.开发人员测试推荐的最低标准 **8**
2.1 威胁建模 9
2.2 自动化测试 9
2.3 基于代码或静态的分析 10
2.4 检查硬编码的秘密 10
2.5 在语言提供的检查和保护下运行 10
2.6 黑盒测试用例 11
2.7 基于代码的测试用例 11
2.8 历史测试用例 11
2.9 模糊测试 12
2.10 WEB 应用程序扫描 12
2.11 检查所包含的软件组件 12
3.技术背景和补充信息 **12**
3.1 补充：内置的语言保护功能 12
3.2 补充：内存安全的编译 13
3.3 补充：覆盖率度量 14
3.4 补充：模糊测试 15
3.5 补充：WEB 应用程序扫描 16
3.6 补充：静态分析 16
3.7 补充：人工属性审查 17
3.8 补充：测试用例来源 18
3.9 补充：高级 BUG 19
3.10 补充：检查包含的软件是否存在已知漏洞 20
4.超越软件验证 **20**
4.1 良好的软件开发实践 20
4.2 良好的软件安装和操作实践 21
4.3 额外的软件保障技术 22
5.调查过的文档 **22**
6.词汇表和缩写 **23**
参考文献 **24**

1. 简介

1.1 概述

为了确保软件足够安全，必须对软件进行良好的设计、构建、交付和维护。在软件开发生命周期（SDLC）中，开发人员尽早进行频繁且彻底的验证是软件安全保障的一个关键要素。在最高的概念层次上，我们可以将验证看做是一种提高软件质量的自觉的约束 [1,p.10]。正如 NIST 的安全软件开发框架（SSDF）所说，验证被用于“识别漏洞并验证是否符合安全要求”[2,PW.7 和 PW.8]。根据 ISO/IEC/IEEE 12207:2017[3,3.1.72]，验证有时被非正式的称为“测试”，它包括许多静态和主动的保障技术、工具以及相关过程。它们必须与其他方法一起使用，以确保高水平的软件质量。
本文件推荐了软件生产商进行软件验证的最低标准。没有一个单一的软件安全验证标准能够涵盖所有类型的软件，并且在支持高效且有效验证的同时既具体又规范。因此，本文档推荐了软件开发商在创建自身流程时使用的指南。为达到最佳效果，流程必须非常具体且需根据软件产品、技术（如语言和平台）、工具链和开发生命周期模型定制。有关验证如何适应更大的软件开发流程的信息，请参阅 NIST 的安全软件开发框架（SSDF）[2]。

1.2 主旨

本文件是对 2021 年 5 月 12 日关于改善国家网络安全的行政命令（EO）14028 号的回答[4]。
本文件回应第 4 节-加强软件供应链安全的第（r）小节：
“……对供应商测试其软件源代码给出最低标准的建议，包括确定建议的手动或自动测试类型（如：代码审计工具、静态和动态分析、软件组份工具和渗透测试）。”[4,4(r)]

1.3 范围

本节将阐明或解释形成本文件范围基础的相关术语。
我们把“软件”定义为可执行的计算机程序。
我们将辅助而重要的材料排除在范围外，如配置文件、文件或执行权限、操作过程和硬件。

除了第 2 节中推荐的最低标准外，许多类型的软件还需要特定的测试机制。例如，实时软件、固件（微码）、嵌入式/网络物理软件、分布式算法、机器学习（ML）或神经网络代码、控制系统、移动应用程序、关键安全系统和加密软件。我们不涉及特定的测试。我们建议对连接到网络的软件和并行/多线程软件使用最低限度的测试技术。
作为特别说明，对关键安全系统的测试要求由各自相关的监管机构提出。

虽然 EO 使用术语“软件源代码”，但其含义更为广泛，包含的软件一般包括二进制文件、字节码和可执行文件（如库和包）。我们也承认，不可能像人类可读的源代码那样既彻底且有效地检验。
除非作为测试参考，我们不考虑验证或确认安全性功能的需求和规范。

版权所有©天融信 保留一切权利

我们将非正式术语"测试"理解为:为了确保软件按预期执行而对软件本身执行的任何技术或过程，它具有必要的属性且没有重要漏洞。我们替代使用 ISO/IEC/IEEE 的术语"验证"。除了动态分析或运行程序（狭义的"测试"）之外，验证还包括静态分析和代码审计等方法。
我们将软件开发中有助于软件保障的其它关键元素排除在验证处理之外，例如程序员培训、专业知识或认证、来自先前或后续软件产品的证据、过程、正确构建或基于模型的方法、供应链和编译保障技术，以及在运行使用过程中的故障报告。
验证的前提是标准的语言语义、正确且强大的编译或解释引擎，以及可靠而准确的执行环境，如：容器、虚拟机、操作系统和硬件。验证可能在预期的操作环境中进行，但也可能不在预期的操作环境中进行。

请注意，验证必须基于一定的参考资料，如软件规范、编码标准（例如，汽车产业软件可靠性协会(MISRA) C[5]）、属性集合、安全策略或常见弱点清单。

虽然 EO 使用了术语"供应商测试"，但其含义更广泛，也包括开发者。开发者和供应商可能是同一个实体，但许多供应商也会使用来自外部的软件。软件供应商可以对其他实体开发的软件包进行重新验证。尽管 EO 提到了商业软件[4, Sec. 4(a)]，但本指南适用于所有软件开发者，包括政府雇员和开源软件(OSS)的开发人员。本文件中介绍的技术和流程可供软件开发者用于验证其产品中集成的重用软件、客户采购软件、承揽软件的单位或第三方实验室。然而，这些并不是本文件的目标受众，因为这种保障工作应该在开发过程中尽早实施。
本文件提出了"最低标准"。也就是说，本文件不是最有效的实践或推荐实践的指南。相反，它的目的是(1)通过指出开发人员应该已经在使用的技术为软件验证设置一个较低的标准，并且（2）作为未来强制标准的基础。

1.4 验证各方面如何相互关联

本节解释了基于代码的分析和审查与动态分析的关系。软件动态测试的基本过程如图 1 所示。当目标软件达到这个阶段时，应该已经通过了编译器和其他工具的静态分析。在动态测试中，软件在许多测试用例上运行，并对输出进行检查。动态测试的一个优点是它几乎没有误报。有关动态测试的一般模型，请参见[6, Sec. 3.5.1]，它也引用了一些参考文献。

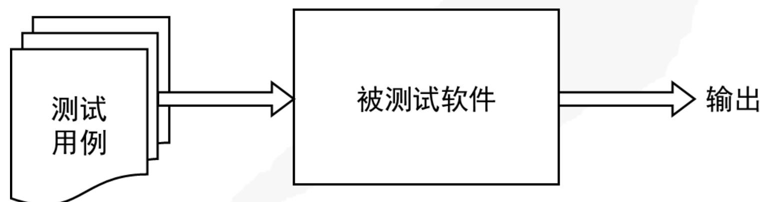

图 1. 基本的动态测试过程是向被测试的软件交付一组测试用例并检查输出

为了准确地执行数千个测试，并进行精确的结果检查，验证必须是自动化的。自动化还允许经常的开展有效的重复验证。
图 2 提供了获得软件保障过程的更多细节。它表明，一些测试用例是由当前测试用例集和代码分析组合而成的，要么是完全通过对软件的静态考虑，要么是通过测试用例执行过程中的覆盖率分析。第 2.6 节和第 2.7 节简要讨论了黑盒和基于代码的测试用例。
代码分析是检查代码本身，以检查其是否具有所需的属性，识别弱点并计算测试完整性的度

版权所有©天融信 保留一切权利

量。它还用于诊断测试期间发现的故障原因。详情见第 3.6 节。
这种分析可以确定测试执行了哪些语句、例程、路径等，并可以产生测试完成程度的度量。代码分析还可以监视异常、内存泄漏、未加密的关键信息、空指针、SQL 注入或跨站点脚本等错误。
在测试过程中，这种混合分析还可以驱动自主测试（参见第 2.9 节和第 2.10 节）并用于交互式应用程序安全测试（IAST）。运行时应用程序自我保护（RASP）在运行期间监视程序的内部安全性。

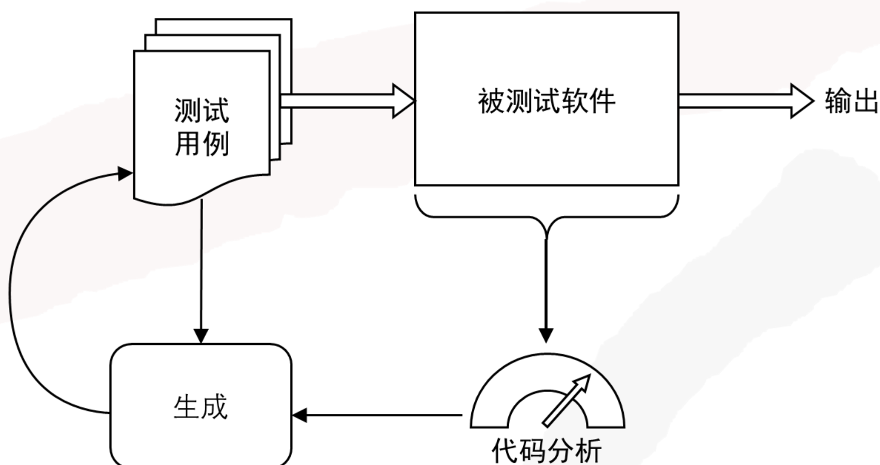

图 2. 一个更详细的验证过程图，添加了一些测试用例是如何生成的，并显示了代码分析是如何适应的

在演变成系统性故障之前处理这些错误。IAST 和 RASP 还可以检查输出，例如：传输中的未加密敏感数据。

1.5 文件大纲

第 2 节从一个简要指南开始，推荐开发人员用此验证他们软件的最低技术标准，然后对技术进行了扩展。第 3 节具有资料性，也就是说，它不属于所推荐的最低标准的一部分。它提供了有关这些技术的背景和补充材料，包括参考资料、更彻底的变更和替代，以及示例工具。第 4 节总结了如何从一开始就能很好地构建软件。最后，第 5 节列出了本文件所参考的资料。

2. 开发人员测试推荐的最低标准

要确保软件按开发人员的意图运行并且充分避免有意设计到软件中或在其生命周期的任何时候意外插入的漏洞，这就需要使用许多相互关联的技术。
本指南为开发人员测试推荐了以下最低标准：

- 进行威胁建模（**2.1**）

对静态和动态分析使用自动测试（2.2），

- 进行静态（基于代码）分析
  - 使用代码扫描器查找高级错误（**2.3**）。

版权所有©天融信 保留一切权利

  - 使用启发式工具寻找硬编码的秘密，并确定可能需要重点人工审查的软件中的小部分（2.4）。
- 进行动态分析（即运行程序）
  - 运行具有内置检查和保护功能的程序（2.5）。
  - 创建“黑盒”测试用例，例如，从规范、输入边界分析，以及威胁建模驱动的测试用例（2.6）。
  - 创建基于代码的（结构化）测试用例。根据需要添加用例，从而达到至少 80%的覆盖率（2.7）。
  - 使用为捕捉以前的错误而设计的测试案例（2.8）。
  - 运行模糊测试（2.9）。如果软件运行一个 web 服务，也应运行一个 web 应用程序扫描器（2.10）。
- 纠正已发现的“必须修复”的错误，并改进流程，以防止将来出现类似的缺陷，或者至少能更早地捕获它们[2，RV.3]。
- 使用类似的技术来确保所包含的库、包、服务等不低于代码的安全性（2.11）。

本节其余部分提供了有关推荐最低标准的各方面的附加信息。

2.1 威胁建模

我们建议尽早使用威胁建模，以便识别设计级的安全问题并集中验证。威胁建模方法创建了一个抽象的系统和潜在攻击者的概要文件，其中包括他们的目标和方法以及潜在威胁的目录[7]，另见[8]。Shevchenko 等人[7]列举了 12 种威胁建模方法，指出软件的需求应驱动所使用的方法。威胁建模应该在开发过程中多次进行，特别在开发新功能时，以捕获新威胁并改进建模[9]。2019 年 8 月的国防部企业级 DevSecOps 参考设计文件包含了威胁建模如何融入软件开发（Dev）、安全（Sec）和运营（Ops）的图表[10, Fig. 3]。DevSecOps 是一种有组织的软件工程文化和实践，专注于统一与软件相关的开发、安全和运营方面。
正如威胁评估或威胁场景所示，测试用例应该在影响最大的领域更加全面。威胁建模还可以指出哪些输入向量最受关注。测试这些特定输入的变化应具有更高的优先级。威胁建模可能会揭示某些小的代码片段（通常少于 100 行）会带来重大风险。此类代码可能需要人工代码审查来回答具体问题，如“软件是否需要授权？”、“软件接口是否检查并验证输入？”请参阅第 3.7 节了解更多关于人工审查的信息。

2.2 自动化测试

对验证的自动支持可以像脚本一样简单。脚本可以重新运行静态分析，然后在一组输入上运行程序，捕获输出并比较输出与预期结果。它也可以像一个设置环境、运行测试、然后检查是否成功的工具一样复杂。一些测试工具将界面驱动到 web 应用程序，让测试人员指定高级命令，例如“单击此按钮”或“将以下文本放入框中”，而不是将鼠标指针移动到呈现屏幕上的某个位置并传递事件。先进的工具会生成哪些代码通过测试的报告，或通过测试的模块及子系统数量的摘要。

我们建议自动化验证

- 确保静态分析不会报告新的弱点，
- 持续运行测试，
- 检查结果是否准确，以及
- 尽量减少对人力和专业知识的需求。

自动化验证可以集成到现有的工作流或问题跟踪系统中[2，PO.3]。因为验证是自动化的，所以可以经常的重复验证，例如在每次提交时或问题解决之前。

2.3 基于代码或静态的分析

尽管有混合分析，但分析通常可分为两种方法：1.基于代码或静态的分析（例如：静态应用程序安全测试——SAST）；2.基于执行或动态的分析（例如：动态应用程序安全测试——DAST）。纯粹基于代码的分析是独立于程序执行的。静态代码扫描器对所编写的代码进行推理，其方式与人工代码审计员的方式大致相同。
扫描器可能解决的问题包括：

- 此软件是否始终满足所需的安全策略？
- 是否满足重要的属性？
- 任意输入是否会导致故障？

我们建议使用静态分析工具来检查代码中的各种漏洞（参见第 3.9 节）并检查符合组织的编码标准。对于多线程或并行处理软件，请使用能够检测竞态条件的扫描器。参见第 3.6 节的示例工具和更多指南。
静态扫描器的复杂程度不一，从简单地搜索任何废弃函数的使用，到寻找指示潜在漏洞的模式，再到能够验证一段代码是否忠实地实现了通信协议。除了闭源工具之外，还有功能强大的免费和开源工具，它们提供了广泛的分析辅助工具，例如：导致冲突的控制流和数据值。
静态源代码分析应该在完成代码编写后立即进行。小段的代码可以在大段可执行的代码完成前进行检查。

2.4 检查硬编码的秘密

我们建议使用启发式工具来检查硬编码的密码和私有加密密钥。这些工具是可行的，因为将这些作为参数的函数或服务具有特定的接口。动态测试不太可能发现这种不需要的代码。
虽然减少恶意代码发生几率的主要方法是完整性措施，但启发式工具可以辅助提供对可疑的小段代码的识别，进而可能触发手动检查。
第 3.7 节列出了在扫描或审查期间可能被检查的其他属性。

2.5 在语言提供的检查和保护下运行

编译和解释的编程语言都提供了许多内置的检查和保护。在开发期间和发布的软件中使用这

版权所有©天融信 保留一切权利

些功能[2，PW.6.2]。同时启用硬件和操作系统安全以及漏洞缓解机制（参见第 3.1 节）。
对于使用非内存安全语言编写的软件，请考虑使用增强内存安全性的技术（参见第 3.2 节）。
解释型语言通常内置有重要的安全执行措施，尽管可以启用额外的措施。此外，你可以使用静态分析器，有时称为“linter”，它检查危险函数、有问题的参数和其他可能的漏洞（参见第 3.6 节）。
即使有了这些检查，程序也必须执行。除非程序具有最小的输入空间，否则执行所有可能的输入是不可能的。因此，开发人员必须选择或构造要使用的测试用例。静态代码分析可以在测试用例之间增加保障，但仍需要选择性的执行测试。许多原则可以指导测试用例的选择。

2.6 黑盒测试用例

“黑盒”测试并不基于实现或特定代码。相反，它们是基于功能规范或要求、负面测试（无效输入和测试软件不应该做的事情）[11, p. 8-5, Sec. 8.B]，拒绝服务和过载（第 3.8 节），输入边界分析，以及输入组合[12，13]。
在通用安全原则所指明的敏感或关键的安全领域，测试用例应该更加全面
如果你可以正式证明此类错误不会发生，那么上面描述的一些测试可能就不需要了。此外，严格的过程度量可能表明，某些测试所带来的好处相对于成本来说是很小的。

2.7 基于代码的测试用例

基于代码或结构的测试用例是基于代码的实现细节的。例如，假设软件需要处理多达 100 万个项。程序员可以决定实现软件来处理静态分配表中的 100 个或更少的项，但是如果有超过 100 个项，则需要动态分配内存。对于这个实现，用例中恰好有 99、100 和 101 项是很有用的，这样可以测试在不同方法之间切换的 bug。内存对齐问题可能表示需要额外的测试。这些重要的测试用例不能仅仅通过考虑规范来确定。
基于代码的测试用例也可能来自覆盖率指标。如图 2 所示，当运行测试时，软件可能会记录代码中被执行或“覆盖”的分支、块、函数调用等。然后用工具分析这些信息来计算度量。还可以添加额外的测试用例来增加覆盖率。
大多数代码应该在单元测试期间执行。我们建议执行测试套件至少达到 80%的语句覆盖率[14]（参见第 3.3 节）。

2.8 历史测试用例

一些测试用例是专门为发现特定 bug 的存在（后来是不存在的）而创建的，这些有时被称为“回归测试”。这些测试用例是在流程成熟到足够覆盖它们之前的一个重要测试源。也就是说，要以采用基于“第一原则”的保障方法检测 bug 为止。一个更好的选择是采用一种保障方法，例如：选择完全排除 bug 的语言。
从生产操作中记录的输入，也可能是测试用例的良好来源。

版权所有©天融信 保留一切权利

2.9 模糊测试

我们建议使用模糊测试，参见第 3.4 节，它执行自动主动测试。也就是说，模糊测试在测试期间会产生大量的输入。通常只有极小部分的输入会触发代码问题。
此外，这些工具仅执行常规检查，以确定软件是否正确处理了测试。通常情况下，只监视广泛的输出特性和总体行为，例如：应用程序崩溃。
通用性的优势在于，这样的工具可以在很少的人为监督下尝试大量的输入。这些工具可以用容易触发 bug 的输入来编程，例如：很长或空的输入和特殊字符。

2.10 Web 应用程序扫描

如果软件提供 web 服务，请使用动态应用程序安全测试（DAST）或交互式应用程序安全测试（IAST）工具，例如：web 应用程序扫描器（参阅第 3.5 节）来检测漏洞。
与模糊测试一样，web 应用程序扫描器在运行中会产生输入。web 应用扫描器可监视一般异常行为。混合型或 IAST 工具还可以监视程序执行中的内部错误。当输入导致一些可检测的异常时，该工具可以使用输入的变化来探测故障。

2.11 检查所包含的软件组件

使用本节推荐的验证技术来确保所包含的代码至少与本地开发的代码一样安全[2，PW.3]。
一些保障可能来源于自行证明或部分自行证明的信息，例如核心基础设施计划（CII）最佳实践标志[15]或可信的第三方检查。
必须根据已知漏洞的数据库对软件组件进行持续监控；随时报告现有代码中的新漏洞。
软件组份分析（SCA）或源代码分析器（OA）工具可以帮助你识别软件使用的开源库、套件、包、捆绑包、工具包等。这些工具可以帮助确定真正导入的软件，识别重用的软件（包括开源软件），并指出过时的软件或存在的已知漏洞（参见第 3.10 节）。

3. 技术背景和补充信息

本节仅供参考，不是建议的最低标准的一部分。它提供了有关技术和方法的更多细节。子节包括诸如变体、额外警告和注意事项、示例工具和相关标准、指南或参考文献的表格等信息。

3.1 补充：内置的语言保护功能

编程语言有各种内置保护措施，可以防止某些漏洞，对编写不良或不安全的代码给出警告，或者在执行过程中保护程序。例如，许多语言在默认情况下是内存安全的。也有一些语言只提供用以激活保护措施的标志和选项。应尽可能多的使用此类保护措施[2，PW.6.2]。
例如，gcc 具有启用的标志

- 运行时缓冲区溢出检测，
- C++字符串和容器的运行时边界检查，
- 地址空间布局随机化（ASLR），

- 提高堆栈溢出检测的可靠性，
- 堆栈粉碎保护器，
- 控制流量完整性的保护，
- 拒绝潜在的不安全格式字符串参数，
- 拒绝缺失的函数原型，以及
- 报告许多其他的警告和错误。

类似地，Visual Studio 2019 选项“/sdl”支持与上面描述的 gcc 类似的检查。

尽管可以启用其他措施，解释型语言通常内置有重要的安全执行措施。作为解释语言的一个例子，Perl 有一个由“-T”命令行标志启用的“污染(taint)”模式，该模式“打开各种检查，例如检查路径目录以确保它们不可被其他人写入。”[16，10.2]。“-w”命令行选项与 Perl 安全文档 perlsec[17]中解释的其他措施同样有所助益。JavaScript 有一个“use strict”指令来指示代码应该在“严格模式”下执行。例如，在严格模式下不能使用未声明的变量[18]。

此外，你可以使用静态分析器（有时称为“linter”）来检查解释语言中是否存在危险的函数或有问题的参数（参见第 3.6 节）。

除了语言本身提供的功能外，你还可以使用硬件（HW）和操作系统（OS）机制来确保控制流的完整性，例如：Intel 的控制流强制技术（CET）或 ARM 指针验证和登录点。有一些编译器选项可以创建操作码，这样，如果软件运行在启用了这些选项的硬件、操作系统或进程上就会调用这些机制。所有生产中的 x86 和 ARM 芯片都已有并将有这种能力。现在大多数操作系统也都支持这些能力。

用户在技术更新时应该利用好 HW 和 OS 的机制，确保他们正在升级的 HW 或 OS 包含这些基于 HW 的特性。这些机制有助于防止开发过程中无法通过验证检测到的内存崩溃 bug 被利用。

技术、原则或指令	参考文献
“应用警告标志”	[11, p. 8-4, Sec. 8.B]
使用堆栈保护	[19]
防止执行数据存储器原则 17	[20, p. 9]

表 1. 内置语言保护的相关标准、指南或参考文献

3.2 补充：内存安全的编译

有些语言（如 C 和 C++）不是内存安全的。一个很小的内存访问错误可能会导致诸如权限升级、拒绝服务、数据损坏或数据泄露等漏洞。
许多语言在默认情况下是内存安全的，但在需要时（例如：关键的性能要求）有禁用这些安全性的机制。在可行的情况下，使用内存安全语言并限制禁用内存安全机制。
对于不是内存安全的软件编写语言，可以考虑使用自动源代码转换或编译器技术来加强内存安全。
要求将内存映射到固定（硬编码的）地址，会破坏地址空间布局随机化（ASLR）。这应该通过启用适当的编译标志来缓解。（参见第 3.1 节）

示例工具

Baggy Bounds Checking、CodeHawk、SoftBoundCETS 和 WIT。

技术、原则或指令	参考文献
“应用警告标志”	[11, p. 8-4, Sec. 8.B]
使用堆栈保护	[19]
元素 A“在实施阶段避免/检测/删除特定类型的漏洞”	[21, p. 9–12]
FPT AEX EXT.1 反利用功能“应用程序不应请求显式的内存地址映射，除非[分配：显式例外列表]。”	[22]

表 2. 内存安全编译的相关标准、指南或参考文献

3.3 补充：覆盖率度量

除了最简单的程序外，所有程序都难以实现详尽的测试，但彻底测试对于减少软件漏洞又很有必要。覆盖率标准是一种定义需要测试什么以及何时达到测试目标的方法。例如，“语句覆盖率”度量代码中至少执行一次的语句，即“覆盖”的语句。

检查覆盖率可以识别代码中没有经过彻底测试的部分，因此更容易出现 bug。覆盖率的百分比（例如 80%的语句覆盖率）是衡量测试套件彻底性的标准。测试用例可以添加到未执行的代码或路径中。低覆盖率表示测试不充分，但非常高的覆盖率也很难达到[14]。

语句覆盖率是被广泛使用的最弱的标准。例如，考量一个只有"then"分支的"if"语句（也就是没有"else"分支）。知道"then"分支中的语句被执行，并不能保证任何测试都探究了当条件为 false 且主体(body)根本没有执行时会发生什么情况。“分支覆盖率”要求每个分支都被执行。在没有提前退出的情况下，完全分支覆盖意味着完全的块覆盖，因此它比块覆盖更强。数据流和变异是更强的覆盖标准[23]。

一般来说，“……所有测试覆盖率标准可以归结为四种数学结构上的几十个标准：输入域、图形、逻辑表达式和语法描述（语法）。”[1,p.26,2.4]基于输入域的测试的一个例子是组合测试[12,13]，它将输入空间划分为组，并测试组的所有 n-way 组合。块、分支和数据流覆盖是图覆盖的标准。基于逻辑表达式的标准，如修正条件决策覆盖率（MCDC），需要对表达式进行各种真实赋值。

语法描述标准以变异测试为例，这种测试特意且系统地创建带有潜在语法错误的微小语法变化的软件变体。例如，“小于”操作符（<）可以替换为“大于或等于”（>=）。如果一个测试集将原始程序与每个微小的变化区分开来，则该测试集就能充分地执行该程序。变异测试可以应用于规范和程序。

注意：代码可以使用某些标志编译以测量覆盖率，然后使用不同的标志再次编译以进行发布。需要确保用于构建交付产品的源代码和任何包含的二进制文件与那些已验证和度量的覆盖率相匹配。

版权所有©天融信 保留一切权利

技术、原则或指令	参考文献
“测试覆盖分析”	[11, p. 8-6, Sec. 8.B]
“相关的度量标准”	[24]
[ST3.4]“影响覆盖分析”	[25, p. 78]

表 3. 覆盖度量的相关标准、指南或参考文献

3.4 补充：模糊测试

在软件开发过程中，常态化执行模糊测试和相关的自动化随机测试生成技术是非常有用的。对不断变化的代码库进行连续模糊测试有助于提前发现意外的 bug。“模糊测试对于发现漏洞非常有效，因为大多数现代程序都有非常大的输入空间，而该空间的测试覆盖率相对较小”[26]。发布前的模糊测试是特别有用的，因为它使恶意方无法使用相同的工具来挖掘可利用 bug。

模糊测试大多数都是自动化的过程，但也可能有相对较少的持续人工操作。它通常需要一个工具，将生成的输入馈送给被测软件。在某些情况下，可以使用单元测试。模糊测试是计算密集型的，在大规模执行时能达到最佳效果。

分别对组件进行模糊测试可以提高效率，并可提高代码覆盖率。在这种情况下，还必须将整个系统作为一个整体进行模糊测试，以调查组件在一起使用时是否正常工作。

模糊测试的一个关键优势是，它通常生成实际的 bug 正向测试，而不仅仅是静态警告。当模糊测试发现故障时，触发的输入可以被保存并添加到常规测试语料库中。开发人员可以使用导致故障的执行跟踪去理解和修复 bug。当故障是非确定性的时候，例如，在存在线程、多个交互进程或分布式计算的情况下，情况可能就不是这样了。

模糊测试的方法可以根据它们产生输入的方式分为两类：基于变异的(mutation-based)和基于生成(generation-based)的。也就是说，基于变异的模糊测试修改现有的输入，例如：以单元测试的输入生成新的输入。基于生成的模糊测试从描述良好格式输入的形式化语法生成随机输入。同时使用这两种方法，可以充分发挥变异模糊测试和生成模糊测试的优势。使用这两种方法可以覆盖更大的测试用例场景集，提高代码覆盖率，并增加发现代码审查等技术所遗漏的漏洞的几率。

基于变异的模糊测试很容易建立，因为它只需要很少或根本不需要描述结构。对已有输入的变异可能是随机的，也可能遵循启发式的。无指导的模糊测试通常只是浅显地探索执行路径。例如，日期字段的完全随机输入不太可能有效。即使是大多数两位数的日（DD）、三个字母的月缩写（Mmm）和四位数的年（YYYY）的大多数随机输入也将被拒绝。把天数限制在 1-31 天，把月份限制在 1 月、2 月、3 月等，把年份限制在从现在算起的 20 年内，可能仍然不能完成闰世纪计算或更深层的逻辑。基于生成的模糊测试可以通过程序验证来实现更深入的测试，但设置则通常需要更多的时间和专业知识。

现代的基于变异的模糊测试比无指导的模糊测试能更深入地探索执行路径，通过使用工具化和符号化执行等方法来获取尚未探索的路径。以覆盖率为导向的模糊测试（如：AFL++、Honggfuzz 和 libFuzzer）旨在最大化代码覆盖率。

为进一步提高效果，设计审查（应该在开发初期首先开展）可能会表明哪些输入向量是最值得关注的。应该优先对这些特定的输入进行模糊测试。

为了增加检测故障的几率，模糊测试通常与特殊工具一起使用。例如，内存问题可以通过诸如地址消除器（ASAN）或 Valgrind 之类的工具来检测。这种检测可能会造成巨大的开销，

但即便故障不会导致崩溃，也是可以检测到越界内存访问的。

示例工具

American Fuzzy Lop Plus Plus (AFL++)、Driller、dtls fuzzer、Eclipser、Honggfuzz、Jazzer、libFuzzer、Mayhem、Peach、Pulsar、Radamsa 和 zzuf。

技术、原则或指令	参考文献
PW.8：测试可执行代码以识别漏洞并验证是否符合安全要求	[2]
“模糊测试”	[11, p. 8-5, Sec. 8.B]
畸形输入测试（模糊测试）	[27, slide 8]
[ST2.6]“根据应用程序 API 定制的模糊测试”	[25, p. 78]

表 4. 模糊测试的相关标准、指南或参考文献

3.5 补充：Web 应用程序扫描

使用 DAST 和 IAST 工具，如：web 应用程序扫描器，测试运行中的软件。这些工具可以与用户界面（UI）和渲染包集成，以便软件接收按钮点击事件、选择和字段中的文本提交，这与运行中的情况完全一样。然后这些工具监视问题提示的细节，例如：错误消息中的内部表名。许多 web 应用扫描器都包含模糊化。

互联网和 web 协议需要大量复杂的处理，这历来都是严重漏洞的来源。

渗透测试是一种“测试方法，在这种方法中，评估人员通常在特定的限制条件下工作，试图绕过或破坏信息系统的安全特性。”[28]。也就是说，它是人利用工具、技术以及他们的知识和专长来模拟攻击者，以便检测漏洞和利用。

示例工具

Acunetix、AppScan、AppSpider、Arachni、Burp、Contrast、Grabber、IKare、Nessus、Probely、SQLMap、Skipfish、StackHawk、Vega、W3af、Wapiti、WebScarab、Wfuzz 和 Zed 攻击代理（ZAP）。

技术、原则或指令	参考文献
“Web 应用程序扫描器”	[11, p. 8-5, Sec. 8.B]
以“测试/编码阶段的安全测试”细分“系统测试”	[24]
[ST2.1]“将黑盒安全工具集成到 QA 过程中”	[25, p. 77]
3.12.1e“进行渗透测试[任务：组织定义的频率]，利用自动扫描工具和使用主题专家的特殊测试。”	[29]

表 5. Web 应用程序扫描的相关标准、指南或参考文献

3.6 补充：静态分析

静态分析或静态应用程序安全测试（SAST）工具，有时称为“扫描器”，检查源代码或二进制代码，以警告可能存在的弱点。使用这些工具可以实现早期、自动化的问题检测。一些工具可以从集成开发环境（IDE）中访问，并为开发人员提供即时的反馈。扫描器可以发现诸如缓冲区溢出、SQL 注入和违反组织编码标准等问题。其结果可以突出显示受影响的精确文

件、行号，甚至是执行路径，以帮助开发人员修正。
组织应该选择和标准化静态分析工具，并根据他们使用该工具的经验、正在开发的应用程序和报告的漏洞来建立“必须修复”bug 列表。你可以参考已发布的头部 bug 列表（参见第 3.9 节），来创建一个进程专用的“必须修复”bug 列表。
SAST 的扩展性很好，因为测试可以在大型软件上重复运行，就像整个系统的每日构建或在开发人员的 IDE 中一样。
SAST 工具也有缺点。某些类型的漏洞是很难发现的，例如：身份验证问题、访问控制问题和不安全加密技术的使用。在绝大多数的工具中，存在警告误报的情况，有些则在软件环境中是无关紧要的。此外，工具通常无法确定一个脆弱性是实际的还是在应用程序中已得到缓解。工具用户应该使用工具所提供的警告来确定消除和优先级机制，对工具结果进行分流，并集中精力纠正最重要的缺陷。
由于代码风格、启发式方法和漏洞类别相对过程重要性，扫描器具有不同的优势。你可以通过运行多个分析器，并关注每个扫描仪最适合的弱点类别，来实现最大的效益。
许多分析程序允许用户编写规则或模式来提升其效果。

示例工具

Astrée、Polyspace Bug Finder、Parasoft C/C++测试、Checkmarx SAST、CodeSonar、Coverity、Fortify、Frama-C、Klocwork、SonarSource 和 SonarQube 处理许多常见的编译语言。
对于 JavaScript，JSLint、JSHint、PMD 和 ESLint。

技术、原则或指令	参考文献
PW.8：测试可执行代码以识别漏洞并验证是否符合安全要求	[2]
“源代码质量分析器” “源代码缺陷分析器”	[11, p. 8-5, Sec. 8.B]
[CR1.4]“使用自动工具和手动审查” [CR2.6]“使用带有定制规则的自动化工具”	[25, pp. 75–76]

表 6. 静态分析的相关标准、指南或参考文献

3.7 补充：人工属性审查

正如第 3.6 节中所讨论的，静态分析工具可以扫描许多属性和潜在问题。有些属性不适合计算机识别，因此可能需要人工检查。这种检查可能会更有效，因为扫描显示了可能的问题或需要关注的位置。
代码的原始作者以外的其他人可以对其进行审查，以确保

- 执行边界检查[19]，
- 设置数据的初始值[19]，
- 仅允许授权用户访问敏感事务、功能和数据[30,p. 10, Sec. 3.1.2]（可能包括检查用户功能是否独立于系统管理功能[30, p.37,Sec.3.13.3]），
- 限制失败的登录尝试[30, p. 12, Sec. 3.1.8]，
- 对一定时间内不活动的会话进行锁定[30, p. 13, Sec. 3.1.10]，

- 在特定条件下自动终止会话[30, p. 13, Sec. 3.1.11],，
- 具有“促进组织系统内有效信息安全”的体系结构[30, p. 36, Sec. 3.13.2]，
- 不将内存映射到硬编码位置，参见 Sec. 3.2，
- 对敏感数据进行加密传输[30, p. 14, Sec. 3.1.13]和加密储存[30, p. 15, Sec. 3.1.19].
- 使用标准的服务和应用程序接口（API）[2,PW.4]，
- 具有安全的默认配置[2,PW.9]，以及
- 具有保持最新的文档化界面。

文档化的界面包括输入、选项和配置文件。为了减少攻击面，界面应该很小[31, p. 15]。威胁建模可能指示某些代码具有重大风险。对小段代码(通常少于 100 行)进行集中的手动检查可能对成本有利。审查可以回答具体的问题。例如，软件是否需要授权?软件界面是否包含输入检查和验证?

技术、原则或指令	参考文献
“集中人工抽查”	[11, p. 5-6, Sec. 5.A]
3.14.7e“使用[任务：组织定义的安全关键或基本软件、固件和硬件组件]验证[任务:组织定义的验证方法或技术]的正确性	[29]

表 7. 人工属性检查的相关标准、指南或参考文献

3.8 补充：测试用例来源

通常情况下，测试是基于规范或要求的，目的是确保软件完成它本应完成的工作，以及过程经历，也是为了确保以前的错误不会再犯。额外的测试用例可基于以下原则：

- 威胁建模——集中在影响最严重的领域，
- 通用安全原则——发现安全漏洞，例如：无法检查凭据，因为这些漏洞通常不会导致操作失败（崩溃或错误输出），
- 负面测试——确保软件对无效输入的行为合理，且不做它不该做的事，例如：确保用户无法执行未经授权的操作[11, p. 8-5, Sec. 8.B]，
- 组合测试——发现在处理某些 n 元组类型的输入时发生的错误[12,13]，以及
- 拒绝服务和过载——确保软件具有弹性。

拒绝服务和过载测试也称为压力测试，同时还要考虑算法攻击。算法可能在典型负载或预期过载情况下工作得很好，但攻击者可能会造成比实际使用中出现的负载高很多个数量级的负载。
额外监视负面测试期间的执行和输出，例如使用交互式应用程序安全测试(IAST)工具或运行时应用程序自我保护(RASP)工具。

版权所有©天融信 保留一切权利

技术、原则或指令	参考文献
“简单攻击建模” “负面测试”	[11, pp. 8-4 and 8-5, Sec. 8.B]
以“测试/编码阶段的安全测试”细分“单元测试”并进而细分“系统测试” 以“安全测试活动”细分“风险分析”	[24]
AM1.2“创建数据分类分级方案和清单” AM1.3“识别潜在攻击者” AM2.1“构建与潜在攻击者相关的攻击模式和滥用用例” AM2.2“创建特定技术的攻击模式” AM2.5“建立并维护一个前 N 名的潜在攻击列表” AM3.2“创建并使用自动化来模拟攻击者”	[25, pp. 67–68]
[ST1.1]“确保 QA 执行边缘/边界值条件测试” [ST1.3]“以安全需求和安全功能驱动测试” [ST3.3]“以风险分析结果驱动测试”	[25, pp. 77–78]
[SE1.1]“使用应用程序输入监控” [SE3.3]“使用应用程序行为监控和诊断”	[25, pp. 80 and 82]
3.11.1e 采用[*任务：组织定义的威胁情报来源*]作为风险评估的一部分，以指导和告知组织系统的开发、安全架构、安全解决方案的选择、监控、威胁狩猎以及响应和恢复活动 3.11.4e 在系统安全计划中记录或引用所选择的安全解决方案、安全解决方案的基本原理和风险判定	[29]

表 8. 测试用例来源的相关标准、指南或参考文献

3.9 补充：高级 Bug

有许多高优先级的 Bug 和缺陷，如：常见缺陷列举（CWE）/SANS Top 25 最危险软件错误[32,33]、CWE 缺陷风向标[34]或开放式 Web 应用程序安全项目（OWASP）Top 10 Web 应用程序安全风险[35]。
这些列表以及发现 bug 的经验，有助于开发人员在验证和过程改进期间选择需要重点关注的 bug 类别。

技术、原则或指令	参考文献
“UL 与网络安全”	[27, slide 8]
在安全方面，“代码质量规则”列出了 36 个“父”CWEs 和 38 个“子”CWEs。为了提高可靠性，它列出了 35 个“父”CWEs 和 39 个“子”CWEs。	[36] [37]

表 9. 高级 Bug 的相关标准、指南或参考文献

版权所有©天融信 保留一切权利

3.10 补充：检查包含的软件是否存在已知漏洞

你需要对所包含的代码（例如：闭源软件、自由和开源软件、库和包）有与你开发的代码一样足够的保障。如果你缺乏有力保障，我们建议你对包含的代码执行与自有代码等同的测试。包和库的早期版本可能存在已知的漏洞，这些漏洞会在后续的版本中得到纠正。
用软件组份分析（SCA）和来源分析（OA）工具扫描代码库，以确定包含哪些代码。他们还会检查所包含的代码中报告的任何漏洞[11,App. C.21, p.C-44]。NIST 国家漏洞数据库（NIST National Vulnerability database, NVD）是一个被广为使用已知漏洞的公开数据库，它使用常见漏洞和暴露（Common Vulnerability and exposure, CVE）来识别漏洞。有些工具可以配置为阻止下载有安全问题的软件，并推荐其他的下载方式。
因为库(libraries)与工具的数据库匹配，所以它们不会识别数据库中缺少的库。

示例工具

Black Duck、Binary Analysis Tool (BA T)、Contrast Assess、FlexNet Code Insight、FOSSA、JFrog Xray、OW ASP Dependency-Check、Snyk、Sonatype IQ Server、SourceClear、WhiteHat Sentinel SCA 和 WhiteSource Bolt。

技术、原则或指令	参考文献
“来源分析器”	[11, App. C.21, p. C-44]
“UL 与网络安全”	[27, slide 8]
3.4.3e 使用自动发现和管理工具来维护系统组件的最新、完整、准确和随时可用的详细清单	[29]

表 10. 用于检查包含的软件是否存在已知漏洞的相关标准、指南或参考文献

4. 超越软件验证

良好的软件必须从一开始就做好。验证只是交付满足操作安全需求的软件的一个要素。使用上述软件保障技术是提高企业供应链安全性的最少步骤。第 4.1 节描述了一些通用的软件开发实践，以及保证如何融入到更大的安全软件开发和运营主题中。即使是具有可靠安全特性的软件，如果其安装、操作或维护的方式引入了漏洞，也可能被对手利用。第 4.2 节描述了一些可改进软件保障的趋势和技术。第 4.3 节描述了良好的安装和操作原则。软件开发和安全技术都在不断发展。

4.1 良好的软件开发实践

理想情况下，软件设计上是安全的，而且设计和实现的安全性都可以得到证明、记录和维护。随着时间的推移，软件开发乃至整个软件开发生命周期都发生了改变，但是一些基本原则仍适用于所有情况。NIST 制定了一份网络安全白皮书，“通过采用安全软件开发框架 (SSDF) 来降低软件漏洞的风险”[2]，该白皮书提供了有关这些基本原则的概述和参考。该文件是一个正在进行的项目的一部分；详见 https://csrc.nist.gov/Projects/ssdf。SSDF 介绍了一个基于既定的安全软件开发实践文件的基本、健全和安全的软件开发框架。为了使验证最有效，它应该是更大的软件开发过程的一部分。SSDF 实践分为四组：

- 组织准备（PO）：确保组织的人员、流程和技术在组织层面做好准备，在某些情况下，为每个单独的项目开发安全的软件。
- 保护软件（PS）：保护软件的所有组件免受篡改和未经授权的访问。
- 生产安全性良好的软件（PW）：生产安全性良好的软件，在其发布版本中具有最小的安全漏洞。
- 响应漏洞（RV）：识别软件版本中的漏洞，对这些漏洞进行适当的应对，并防止以后再次出现类似的漏洞。

在 DevOps 的背景下，安全发展的企业具备以下特征：

- 企业创造了一种“安全是每个人的责任”的文化。这包括将安全专家纳入到开发团队中，培训所有开发人员使其了解如何设计和实现安全软件，并让开发人员和安全人员能使用自动化工具跟踪漏洞。
- 企业使用工具进行自动化的安全检查，通常称为：安全即代码[38]。
- 除了典型的系统指标外，企业还跟踪威胁和漏洞。
- 企业在安全团队、开发人员和运营人员之间共享软件开发任务信息、安全威胁和漏洞知识。

4.2 良好的软件安装和操作实践

综上所述，即使是没有发现安全漏洞的软件，如果在软件的安装、操作或维护过程中引入了漏洞，也可能被对手利用。本文中没有直接解决的一些问题包括：错误配置、违反文件权限策略、违反网络配置以及接受了伪造或更改的软件。请特别参阅即将出版的“行政命令（EO）14028 关键软件使用的安全措施”，其中涉及补丁管理、配置管理和持续监控以及其他安全措施，并列出了参考文献。

配置文件：由于软件应用程序和网络环境的不同，许多计算机应用程序、服务器进程和操作系统的参数和初始设置都是可配置的。通常情况下，安全验证无法预测意外的设置。当使用限制性设置时，系统和网络运营商通常会为了便于更困难或不可行的任务而更改设置。特别是在访问授权和网络接口的情况下，更改配置设置可能会引入关键漏洞。软件发布应包括安全的默认设置和偏离于这些设置的警告。安全验证应包括所有有效的设置和（可能）确保运行时检查可发现无效的设置。应警告或通知甲方，除明确允许的设置以外，其他的设置将使开发人员的安全声明无效。

文件权限：应使用最小权限原则来建立文件所有权和读取、写入、执行和删除文件的权限。无论软件经过多么彻底的验证，如果它可以被修改或文件可以被未经授权的实体访问，安全性都会受到损害。更改文件权限的能力需要限制在明确授权的主体上，这些主体的认证方式要与软件被破坏的影响相称。文件权限在维护安全生命中的作用需要明确。

网络配置：安全配置是指，为了减少网络漏洞，而在构建和安装计算机和网络设备时实施的安全措施。正如文件权限对软件的持续完整性至关重要一样，网络配置也会限制对软件的未授权访问。验证需要覆盖所有有效的网络配置设置，并（可能）确保运行时检查可发现无效的设置。网络配置在界定安全声明的适用性方面的作用需要明确。

操作配置：软件是在其使用环境中被利用的。添加或删除依赖于软件产品或该产品所依赖的组件可以验证或否定软件和系统运行安全性所依赖的假设。特别是在源代码的情况下，操作代码本身依赖于编译器和解释器等组件。在这种情况下，软件的安全性可能会因其他产品而

失效。验证需要在与预期的操作配置一致的环境中进行。开发人员需要明确说明，安全声明对实现软件或操作配置的其他方面的任何依赖。必须保持供应链的完整性。

4.3 额外的软件保障技术

软件验证随着新方法的开发而不断改进，并且方法可以适应不断变化的开发和操作环境。但仍然存在一些挑战，例如，应用形式化的方法来证明设计糟糕的代码的正确性。可增加基于验证的安全保障的近期进展包括：

- 应用机器学习减少自动安全扫描工具的误报，并增加这些工具可检测的漏洞。
- 调整为自动化 web 界面测试而设计的工具，例如：Selenium，以便为应用程序生成安全测试。
- 提高基于模型的复杂系统安全测试的可扩展性。
- 在以下方面改进自动化 web 应用程序安全评估工具：
  - 会话状态管理
  - 脚本解析
  - 逻辑流
  - 自定义统一资源定位符（URL）
  - 特权提升
- 应用可观测性工具在云环境中提供安全保障。
- 调整当前的安全测试，以实现云服务安全保障。

其他减少软件漏洞的其他技术在 NIST-IR 8151[39]“显著减少软件漏洞”中有所描述。

5. 调查过的文档

本节列出了我们为编写本文档而调查过的一些标准、指南、参考文献等。我们把它们列出来是为了给未来的工作提供一个从哪里开始或快速了解哪些可能被忽略的思路。我们对相关的参考文献进行分组。

Donna Dodson、Murugiah Souppaya 和 Karen Scarfone，“通过采用安全软件开发框架（SSDF）降低软件漏洞风险”，2013 年[2]。

David A. Wheeler and Amy E. Henninger，“2016 用于软件漏洞检测、测试和评估的尖端资源（SOAR）”，2016 年[11]。

Steven Lavenhar，“代码分析”，2008 年[19]。
C.C.Michael、Ken van Wyk 和 Will Radosevich，“基于风险和功能的安全测试”，2013 年[24]。

UL，“物联网安全 20 大设计原则”，2017 年[20]。

 版权所有©天融信 保留一切权利

Tom Haigh 和 Carl E.Landwehr，“医疗设备软件安全的代码构建规范”，2015 年[21]。
Ulf Lindqvist 和 Michael Locasto，“物联网的代码构建规范”，2017 年[31]。
Carl E.Landwehr 和 Alfonso Valdes，“电力系统软件安全的代码构建规范”，2017 年[40]。

“应用软件版本 1.3 的保护配置文件”，2019 年[22]。

Ron Ross、Victoria Pillitteri、Gary Guissanie、Ryan Wagner、Richard Graubart 和 Deb Bodeau，“保护受控非机密信息的增强安全要求：NIST SP800-171 的补充”，2021[29]。
Ron Ross、Victoria Pillitteri、Kelley Dempsey、Mark Riddle 和 Gary Guissanie，“保护非联邦系统和组织中的受控非机密信息”，2020 年[30]。
Ron Ross、Victoria Pillitteri 和 Kelley Dempsey，“评估受控非机密信息的增强安全要求”，2021 年[41]。

Bill Curtis、Bill Dickenson 和 Chris Kinsey，“CISQ 推荐指南：ADM 服务级别协议的有效软件质量度量”，2015 年[42]。“编码质量规则”，2021 年[36]。

6. 词汇表和缩写

术语	定义
网络安全	保护系统、网络和程序免受数字攻击的实践。
软件源代码	该软件最初以纯文本输入，例如人类可读的字母数字字符。
API	应用程序接口（Application Program Interface）
CVE	常见漏洞和暴露（Common Vulnerabilities and Exposures）
CWE	常见缺陷列举（Common Weakness Enumeration）
DAST	动态应用程序安全测试（Dynamic Application Security Testing）
EO	行政命令（Executive Order）
HW	硬件（Hardware）
IAST	交互式应用程序安全测试（Interactive Application Security Testing）
MISTRA	汽车产业软件可靠性协会（Motor Industry Software Reliability Association）
NIST	国家标准与技术研究院（National Institute of Standards and Technology）
NSA	国家安全局（National Security Agency）
NVD	国家漏洞数据库（National Vulnerability Database）
OA	源分析器（Origin Analyzer）
OS	操作系统（Operating System）
OSS	开源软件（Open Source Software）

版权所有©天融信 保留一切权利

OWASP	开放式 Web 应用程序安全项目（Open Web Application Security Project）
RASP	运行时应用程序自我保护（Runtime Application Self-Protection）
SAST	静态应用程序安全测试（Static Application Security Testing）
SCA	软件组份分析（Software Composition Analysis）
SDLC	软件开发生命周期（Software Development Life Cycle）
SSDF	安全软件开发框架（Secure Software Development Framework）
URL	统一资源定位符（Uniform Resource Locator）

参考文献

[1] Ammann P，Offutt J（2017）*软件测试导论*（剑桥大学出版社）第二版。https://doi.org/10.1017/9781316771273

[2] Dodson D，Souppaya M，Scarfone K（2013）《通过采用安全软件开发框架（SSDF）降低软件漏洞风险》（马里兰州盖瑟斯堡国家标准与技术研究院），网络安全白皮书。https://doi.org/10.6028/NIST.CSWP.04232020

[3] （2017）ISO/IEC/IEEE 12207：系统和软件工程软件生命周期过程（ISO/IEC/IEEE）第一版。

[4] Biden JR Jr （2021）改善国家网络安全，https://www.federalregister. gov/d/2021-10460。行政命令 14028，86 FR 26633，文件编号 202110460，2021 年 5 月 17 日查阅。

[5] 汽车产业软件可靠性协会（2013）*MISRA C:2012：关键系统 C 语言使用指南*（MIRA Ltd，Nuneaton，Warwickshire CV10 0TU，UK）。

[6] Black PE（2019）《统计软件的正式方法》（马里兰州盖瑟斯堡国家标准与技术研究院），IR 8274。https://doi.org/10.6028/NIST.IR. 8274

[7] Shevchenko N，Chick TA，O'Riordan P，Scanlon TP，Woody C（2018）《威胁建模：可用方法总结》，https://resources.sei.cmu.edu/asset files/ WhitePaper/2018 019 001 524597.pdf.查阅日期：2021 年 6 月 8 日。

[8] Shostack A（2014）《威胁建模：安全设计》（John Wiley&Sons，Inc.）。

[9] Keane J（2021），个人通信。

[10] 首席信息官（2019）国防部企业 DevSecOps 参考设计（国防部）1.0 版。查阅日期：2021 年 6 月 8 日 https://dodcio. defence.gov/Portals/0/Documents/DoD Enterprise DevSecOpsReference Design v1. 0 Public Release.pdf.

[11] Wheeler DA，Henninger AE（2016）用于软件漏洞检测、测试和评估的尖端（SOAR）2016（国防分析研究所），P-8005。2021 年 5 月 19 日查阅。网址 https://www.ida.org/researchand-publications/publications/all/s/st/stateoftheart-resources-soar-for-softwarevulnerability-detection-test-and-evaluation-2016.

[12] Kuhn R，Kacker R，Lei Y，Hunter J（2009）组合软件测试。计算机 42（8）：94–96。https://doi.org/10.1109/MC.2009.253. 可在 https://tsapps. nist.gov/publication/get pdf.cfm? pub id=903128

[13] Kuhn DR，Bryce R，Duan F，Ghandehari LS，Lei Y，Kacker RN（2015）*组合测试：理论与实践*（Elsevier），*计算机进展*，第 99 卷，第 1 章，第 1-66 页。https://doi.org/10.1016/bs.adcom.2015.05.003

[14] Cornett S（2013）最小可接受代码覆盖率，https://www.bullseye.com/ minimum.html。

版权所有©天融信 保留一切权利

查阅日期：2021 年 7 月 5 日。
[15] （2018）CII 最佳实践标志计划，https://bestpractices.coreinfrastructure.org/. 查阅日期：2021 年 6 月 25 日。
[16] Wheeler DA（2015）安全编程指南，https://dwheeler.com/secureprograms/Secure-Programs-HOWTO/. 查阅日期：2021 年 6 月 24 日。
[17] （2013）perlsec，https://www.linux.org/docs/man1/perlsec.html. 查阅日期：2021 年 6 月 24 日。
[18] （2021）JavaScript 严格使用，https://www.w3schools.com/js/js_strict.asp. 查阅日期：2021 年 6 月 25 日。
[19] Lavenhar S（2008）代码分析，https://us-cert.cisa.gov/bsi/articles/best-practices/ code-analysis/code-analysis.查阅日期：2021 年 5 月 5 日。
[20] UL （2017） 物联网安全 20 大设计原则， https://ims.ul.com/sites/g/files/qbfpbp196/files/2018-05/iot-security-top-20-design-principles.pdf.查阅日期：2021 年 5 月 12 日。
[21] Haigh T，Landwehr CE（2015）《医疗设备软件安全的代码构建规范》，https://ieeecs-media.computer.org/media/technical-activities/CYBSI/docs/ BCMDSS.pdf 查阅日期：2021 年 4 月 15 日。
[22] （2019）应用软件版本 1.3 的保护配置文件，https://www.niapccevs.org/MMO/PP/PP-APP-v1.3.pdf。查阅日期：2021 年 4 月 27 日。
[23] Zhu H，Hall PAV，May JHR（1997）软件单元测试覆盖率和充分性。*ACM 计算调查* 29（4）：366–427。https://doi.org/10.1145/267580.267590
[24] Michael C，van Wyk K，Radosevich W（2013）《基于风险和功能的安全测试》，https://us-cert.cisa.gov/bsi/articles/best-practices/security-testing/risk-basedand-functional-security-testing. 查阅日期：2021 年 5 月 4 日。
[25] Migues S，Steven J，Ware M（2020）在成熟度模型中构建安全性（Building security in maturity model，BSIMM）–version 11（Synopsys），下载时间：2021 年 4 月 3 日。网址 https://www.bsimm. com/download.html。
[26] Okun V，Fong E（2015）《软件保障的模糊测试》。*Crosstalk。国防软件工程杂志* 28:35-37。
[27] UL（2011）UL 和网络安全，https://alamembers.com/Portals/1/ 10 3 Joseph UL and Cybersecurity.pdf 查阅日期：2021 年 5 月 12 日。
[28] （2015）CNSSI 4009 国家安全系统委员会（CNSS）词汇表，https://www.serdp-estcp.org/Tools-and-Training/Installation-Energy-and-Water/Cybersecurity/Resources-Tools-and-Publications/Resources-and-Tools-Files/CNSSI-4009-Committee-on-National-Security-Systems-CNSS-Glossary 查阅日期：2021 年 7 月 8 日。
[29] Ross R、Pillitteri V、Guissanie G、Wagner R、Graubart R、Bodeau D（2021）《保护受控非保密信息的增强安全要求：NIST SP800-171（马里兰州盖瑟斯堡国家标准与技术研究院）的补充》，SP 800-172。https://doi.org/10.6028/NIST.SP.800-172
[30] Ross R，Pillitteri V，Dempsey K，Riddle M，Guissanie G（2020）《保护非联邦系统和组织中的受控非机密信息》（马里兰州盖瑟斯堡国家标准与技术研究院），SP 800-171r2。https://doi.org/10. 6028/NIST.SP.800-171r2
[31] Lindqvist U，Locasto M（2017）《物联网的代码构建规范》，https://ieeecs-

media.computer.org/media/technical-activities/CYBSI/docs/ Building -Code-IoT online.pdf。查阅日期：2021 年 4 月 28 日。
[32] （2021）关于 CWE，https://cwe.mitre.org/about/. 查阅日期：2021 年 5 月 25 日。
[33] （2021）CWE/SAN 前 25 个最危险的软件错误，https://www.sans.org/ top25-software-errors/.查阅日期：2021 年 5 月 26 日。
[34] （2020）2020 CWE 最危险的 25 个软件缺陷，https://cwe.mitre.org/ top25/查阅日期：2021 年 5 月 26 日。
[35] （2020）OWASP top 10，https://owasp.org/www-project-top-ten/. 查阅日期：2021 年 5 月 26 日。
[36] 信息和软件质量联合会（2021）编码质量规则，https://www.it-cisq.org/coding rules/ 查阅日期：2021 年 5 月 13 日。
[37] （2019）CISQ 自动源代码质量度量（信息和软件质量联盟）中包含的缺陷列表，2021 年 5 月 13 日查阅。
[38] Boyer J（2018）《安全即代码：为什么安全的 DevOps 需要思想的转变》，https://simpleprogrammer.com/security-code-secure-devops/. 查阅日期：2021 年 6 月 10 日。
[39] Black PE，Badger L，Guttman B，Fong E（2016）《显著减少软件漏洞：向白宫科技政策办公室（国家标准与技术研究院）报告》，NISTIR 8151。https://doi.org/ 10.6028/NIST.IR.8151
[40] Landwehr CE，Valdes A（2017）《电力系统软件安全的代码构建规范》，https://ieeecs-media.computer.org/media/technical-activities/CYBSI/docs/ BCPSSS.pdf 查阅日期：2021 年 4 月 28 日。
[41] Ross R，Pillitteri V，Dempsey K（2021），《评估受控非保密信息的增强安全要求》（马里兰州盖瑟斯堡国家标准与技术研究院），SP 800-172A（草案）。https://doi.org/10.6028/NIST.SP.800172A-draft. 查阅日期：2021 年 5 月 21 日。
[42] Curtis B，Dickenson B，Kinsey C（2015）《CISQ 建议指南：ADM 服务级别协议的有效软件质量度量》（信息与软件质量联盟），查阅日期：2021 年 5 月 13 日。

翻译声明：

本文由天融信科技集团翻译整理，原文来自 NIST 公开网站，翻译为公益性质，仅供信息安全产业相关研究人员、管理人员参考，如有错漏敬请指正。