**硕士学位论文**

**（学术学位论文）**

基于脆弱点定位的固件漏洞检测系统设计与实现

**DESIGN AND IMPLEMENTATION OF FIRMWARE VULNERABILITY DETECTION SYSTEM BASED ON VULNERABILITY LOCATION**

**黄正聪**

**哈尔滨工业大学**

**2021年6月**

国内图书分类号：TP309.1 学校代码：10213

国际图书分类号：004.05 密级：公开

**工学硕士学位论文**

基于脆弱点定位的固件漏洞检测系统设计与实现

|  |  |  |
| --- | --- | --- |
| 硕士研究生 | ： | 黄正聪 |
| 导 师 | ： | 王莘副教授 |
| 申请学位 | ： | 工学硕士 |
| 学科 | ： | 网络空间安全 |
| 所 在 单 位 | ： | 网络空间安全学院 |
| 答 辩 日 期 | ： | 2021年6月 |
| 授予学位单位 | ： | 哈尔滨工业大学 |

Classified Index: TP309.1

U.D.C: 004.05

Dissertation for the Doctoral Degree in Engineering

**DESIGN AND IMPLEMENTATION OF FIRMWARE VULNERABILITY DETECTION SYSTEM BASED ON VULNERABILITY LOCATION**

|  |  |
| --- | --- |
| **Candidate：** | Huang Zhengcong |
| **Supervisor：** | Associate Prof.Wang Shen |
| **Academic Degree Applied for：** | Master of Engineering |
| **Speciality：** | Cyberspace Security |
| **Affiliation：** | School of Cyberspace Science |
| **Date of Defence：** | June, 2021 |
| **Degree-Conferring-Institution：** | Harbin Institute of Technology |

# 摘 要

在当今时代，物联网设备在现实生活中发挥着重要作用，也是实现万物互联的重要基础。然而，在这些物联网设备给人们带来便利的同时，其中的嵌入式固件系统中存在的威胁也不容忽视。当固件中存在漏洞时，攻击者就可以利用其入侵物联网设备，实现对固件程序中控制流的劫持，达到执行恶意代码、获取机密信息以及固件程序的内存布局信息等攻击目的。

本文将固件的安全问题作为研究的切入点，结合对国内外在固件漏洞检测相关研究现状的分析，基于二进制程序分析技术，通过分析固件样本发现固件的脆弱点，并识别这些脆弱点是否可以被攻击者利用而触发漏洞，进而设计出基于脆弱点定位的固件漏洞检测系统，实现对固件漏洞的检测。

具体来说，一方面，对于脆弱点的定位，需要基于二进制分析技术，分析固件漏洞触发机制与必要条件、收集漏洞脆弱点信息的收集，从而发现固件中存在的可以被触发漏洞的脆弱点，同时，完成相关算法的设计。另一方面，基于脆弱点定位的相关算法，设计与实现基于脆弱点定位的固件漏洞检测系统。其中，固件漏洞检测系统通过其各个模块完成对固件各类型的脆弱点信息的收集，并根据前序工作对脆弱点漏洞触发机制的分析来识别可利用的脆弱点，从而实现基于脆弱点定位的固件漏洞检测。系统将在固件样本中完成对其各个模块的功能测试，从而验证系统的有效性。

关键字：物联网设备；固件漏洞检测；脆弱点定位；可利用性分析

# Abstract

In today's era, Internet of Things devices play an important role in real life and are also an important foundation for the realization of the Internet of Everything. However, while these Internet of Things devices bring convenience to people, the threats in the embedded firmware system cannot be ignored. When there are loopholes in the firmware, an attacker can use it to invade IoT devices to hijack the control flow in the firmware program, and achieve the purpose of executing malicious code, obtaining confidential information, and memory layout information of the firmware program.

This paper takes the security of firmware as the starting point of the research, combined with the analysis of the current research status of firmware vulnerability detection at home and abroad, and based on the binary program analysis technology, the vulnerabilities of the firmware are found by analyzing the firmware samples, and whether these vulnerabilities can be identified. Attackers use it to trigger the vulnerability, and then design a firmware vulnerability detection system based on the location of the vulnerability to realize the detection of firmware vulnerabilities.

Specifically, on the one hand, for the location of the vulnerability, it is necessary to analyze the firmware vulnerability trigger mechanism and necessary conditions based on binary analysis technology, and collect the collection of vulnerability vulnerability information, so as to discover the vulnerability in the firmware that can be triggered. At the same time, complete the design of related algorithms. On the other hand, based on the related algorithms of vulnerable point location, design and implement a firmware vulnerability detection system based on vulnerable point location. Among them, the firmware vulnerability detection system completes the collection of various types of vulnerability information of the firmware through its various modules, and identifies the exploitable vulnerabilities based on the analysis of the vulnerability trigger mechanism based on the previous work, so as to realize the location based on the vulnerability. Firmware vulnerability detection. The system will complete the functional test of its various modules in the firmware sample to verify the effectiveness of the system.

**Keywords:** Internet of Things equipment, firmware vulnerability detection, vulnerability location, exploitability analysis

目 录

[摘 要 I](#_Toc75203903)

[Abstract II](#_Toc75203904)

[第1章 绪 论 4](#_Toc75203905)

[1.1 课题的来源 4](#_Toc75203906)

[1.2 课题的背景和意义 4](#_Toc75203907)

[1.3 国内外研究现状和分析 5](#_Toc75203908)

[1.3.1 静态分析 5](#_Toc75203909)

[1.3.2 动态分析 6](#_Toc75203910)

[1.3.3 综合分析 8](#_Toc75203911)

[1.3.4 国内外研究现状分析 8](#_Toc75203912)

[1.4 课题主要研究工作 9](#_Toc75203913)

[1.5 论文组织结构 12](#_Toc75203914)

[第2章 固件漏洞检测技术基础 14](#_Toc75203915)

[2.1 面向二进制的固件程序分析技术 14](#_Toc75203916)

[2.1.1 固件解析 14](#_Toc75203917)

[2.1.2 边界二进制程序发现 15](#_Toc75203918)

[2.1.3 二进制程序依赖图构建 16](#_Toc75203919)

[2.1.4 二进制程序数据流分析 18](#_Toc75203920)

[2.2 基于污点分析的漏洞检测技术 18](#_Toc75203921)

[2.3 基于符号执行的漏洞检测技术 19](#_Toc75203922)

[2.4 本章小结 21](#_Toc75203923)

[第3章 面向二进制程序的固件脆弱点分析与定位 22](#_Toc75203924)

[3.1 固件的可利用脆弱点概述 22](#_Toc75203925)

[3.1.1 固件脆弱点的漏洞触发机制与识别 22](#_Toc75203926)

[3.1.2 可利用的固件脆弱点的影响 24](#_Toc75203927)

[3.2 固件脆弱点的漏洞触发机制与识别 24](#_Toc75203928)

[3.2.1 不安全函数和整型溢出 24](#_Toc75203929)

[3.2.2 受控循环 26](#_Toc75203930)

[3.2.3 栈溢出 27](#_Toc75203931)

[3.2.4 堆溢出 29](#_Toc75203932)

[3.2.5 双重释放和释放后访问 32](#_Toc75203933)

[3.3 固件脆弱点定位方法 34](#_Toc75203934)

[3.3.1 不安全函数和整型溢出 34](#_Toc75203935)

[3.3.2 受控循环 35](#_Toc75203936)

[3.3.3 栈溢出 36](#_Toc75203937)

[3.3.4 堆溢出 37](#_Toc75203938)

[3.3.5 双重释放和释放后访问 38](#_Toc75203939)

[3.4 本章小结 39](#_Toc75203940)

[第4章 固件漏洞检测系统的设计与实现 40](#_Toc75203941)

[4.1 系统总体设计 40](#_Toc75203942)

[4.2 系统详细设计与实现 42](#_Toc75203943)

[4.2.1 识别可利用的不安全函数相关脆弱点 42](#_Toc75203944)

[4.2.2 识别可诱发漏洞的受控循环 44](#_Toc75203945)

[4.2.3 获取栈和堆的实时执行状态信息 46](#_Toc75203946)

[4.2.4 识别可利用的栈和堆溢出类脆弱点 49](#_Toc75203947)

[4.3 系统测试 50](#_Toc75203948)

[4.3.1系统验证方案 50](#_Toc75203949)

[4.3.2 实验结果与分析 51](#_Toc75203950)

[4.4 本章小结 58](#_Toc75203951)

[结 论 59](#_Toc75203952)

[参考文献 61](#_Toc75203953)

[哈尔滨工业大学学位论文原创性声明和使用权限 65](#_Toc75203954)

[致 谢 66](#_Toc75203955)

# 第1章 绪 论

## 1.1 课题的来源

本课题来源于国防基础科研项目（编号JCKY2018603B006）XXXX漏洞挖掘技术。

## 1.2 课题的背景和意义

在互联网领域中，固件是可以给设备的特定硬件提供控制功能的特殊计算机软件，以标准化的形式为复杂的设备应用程序提供运行环境，并且能够使得硬件保持相当的独立性，此外，固件也可以作为相比之下较为简单设备的操作系统，从而负责这些设备所有控制、监视和数据操作功能的执行[1]。

当前几乎所有的电子设备中都含有固件，对于这些固件当中漏洞的识别与修复是目前计算机领域的一个热点。就当前的物联网技术来说，其在云计算、智慧产业等领域中有着广泛的应用。而与此同时，其所面临的安全形势也不容忽视。据一份来自惠普研究的报告显示，70%的物联网设备容易含有漏洞，容易受到攻击，其中，这些固件漏洞类型包括密码安全性问题，加密和普遍缺乏粒度用户访问权限等[2]。因此，对于互联网领域中固件漏洞的检测的需求极为迫切。

同时，对于这些固态漏洞的检测方法，在其原理层面上所会涉及到相关的问题也需要得到相关的研究，也就是说，使用特定固件漏洞检测方法后，可能因为技术、设计等方面的缺陷使得固件漏洞被忽视，或者说固件漏洞检测的方法本身并不能够满足根除固件漏洞的需求等[3]。

本课题的研究内容基于二进制的固件静态分析技术，首先通过固件脆弱性分析来定位其中存在的潜在脆弱点，进而在深入分析中收集固件的执行状态信息来判断脆弱点是否满足可以被攻击者利用并触发漏洞的条件，从而检测固件中的漏洞。这样，可以通过发现嵌入式固件设备中的可利用的脆弱点。

当这些固件的脆弱点满足它们的漏洞触发条件后，攻击者就可以直接通过嵌入式设备中负责与外部交互的程序中特定程序点输入受控的数据，从而完成固件程序控制流的劫持，或者是直接获取固件程序的机密信息与关键内存布局信息等。

所以，在完成本课题的研究工作之后，可以发现嵌入式设备的固件系统中存在的可以触发漏洞的脆弱点，一方面，可以为固件的安全性评估方面，提供固件的脆弱性信息，从而有利于后续完善固件系统；另一方面，则可以避免攻击者通过网络的手段，攻击这些物联网设备中固件这一关键组成部分，从而避免引发相关安全事件（例如，获取隐私信息或是破坏物联网设备等），进而损害相关使用人员的利益。

## 1.3 国内外研究现状和分析

对于固件漏洞检测方法，按照是否需要执行程序，可以将检测方法划分为以下三类。

静态分析方法可以省略在实际的设备或模拟工具上执行固件而完成漏洞的检测[3]，其应用领域主要是比较不同设备和供应商固件映像之间的漏洞同源性。其缺点时分析能力较弱，并且检测的准确率较低。

动态分析方法通过在实际设备上执行程序，获取运行的实时信息，从而发现漏洞。动态方法的优点是分析过程具有较高的准确率。而缺点在于其每次仅仅分析一条路径上的一次具体输入，不能使每次都能够分析唯一的路径，因此在可扩展性和实用性方面有着局限性。

基于动态分析的固件漏洞检测方法主要包含符号执行、仿真器模糊化等[5]。其中，符号执行是在程序执行的过程中通过将符号状态值来替代具体的值来完成程序分析的技术，可以确定程序输入所对应的执行路径以及相应的约束条件，可以作为静态分析的参考[4]。而模糊技术则是通过向被测对象输入大量的畸形数据，识别其中的异常从而发现漏洞[36]。

综合分析方法通过结合动态与静态分析的分析方法使用几种不同的方法来检测固件中的漏洞。这样，使得漏洞检测方法在具备可扩展性的同时也能有较高的检测准确率。综合分析一般是在静态分析发现固件敏感点后，利用动态分析方法判断其是否满足漏洞触发条件，进而发现固件漏洞。

### 1.3.1 静态分析

在国外对固件漏洞检测关于静态分析方法的研究内容主要包含基于代码属性图的漏洞建模与发现、基于控制流图的结构识别函数以及面向二进制代码的漏洞检测等。

基于代码属性图的漏洞建模与发现方法引入了一种称为代码属性图的源代码表示，它融合了经典程序分析的概念，即抽象语法树、控制流图和程序依赖图，形成一个联合的数据结构，这种全面的表示有利于为常见漏洞的建模，通过对其的遍历可以识别缓冲区溢出、整数溢出、内存泄漏以及格式字符串漏洞[6]。

基于控制流图的结构识别函数主要思想是根据相应控制流图的结构计算函数间的相似度。另外，为了使计算量最小化，其采用了一种基于数值特征的有效预滤波器来快速识别小组候选函数，因此能够在大型代码库中高效地搜索类似的函数。具体说，即是从容易受到攻击的函数开始，在不同的操作系统、处理器体系结构、或是编译器之上完成对二进制程序中类似函数的识别 [7]。

在嵌入式固件系统的漏洞检测研究领域中，二进制代码的漏洞检测的瓶颈是依赖于人工选择的具有代表性的特征，为了解决这一难点，研究人员结合当前深度学习的相关技术，设计了最大散度序列自动编码器，其能够在混淆二进制固件中易于受到攻击和难于受到攻击的二进制程序的情况下，保留其关键的原始信息[15]。

而在国内的研究中，基于静态的固件漏洞检测研究方向包括基于代码相似度的分阶段的固件漏洞检测方法以及基于同源性分析的嵌入式设备固件漏洞检测。现有的基于简单特征匹配的漏洞检测技术无法达到高精度的固件漏洞检测，而直接使用控制流图匹配的方法成本太高。为了解决精确高效的问题，研究者提出了一种基于代码相似度的分阶段的固件漏洞检测方法[9]。第一阶段，利用神经网络的功能嵌入分析功能间的相似性，有效地实现大规模固件安全检测。第二阶段，计算功能本地调用流图之间的相似度，用于细粒度固件安全分析，提高漏洞检测的准确性。另外，基于同源性分析的嵌入式设备固件漏洞检测方法通过对固件分类,并采用二进制差量分析，字符串常量匹配，模糊哈希3种方法分析第三方库同源性,从而检测同类固件中存在的漏洞，研究中的实验证实了该方法能够在不同的固件设备中发现缓冲区溢出与越界以及远程命令注入漏洞等[11]。

### 1.3.2 动态分析

在国外对固件漏洞检测关于动态分析方法的研究内容主要包含基于符号执行和仿真器模糊化的固件漏洞检测等几种类型。

Firmadyne可以用于大规模分析基于Linux的固件，它从互联网上的供应商网站上抓取并下载固件图像，在解包固件映像和提取嵌入式文件系统之后，基于软件的完整系统仿真可以自动大规模地动态分析嵌入式二进制文件，如图1-1所示，其主要由四个组件组成，包括网络爬虫、固件文件系统提取，初始仿真以及动态分析[8]。

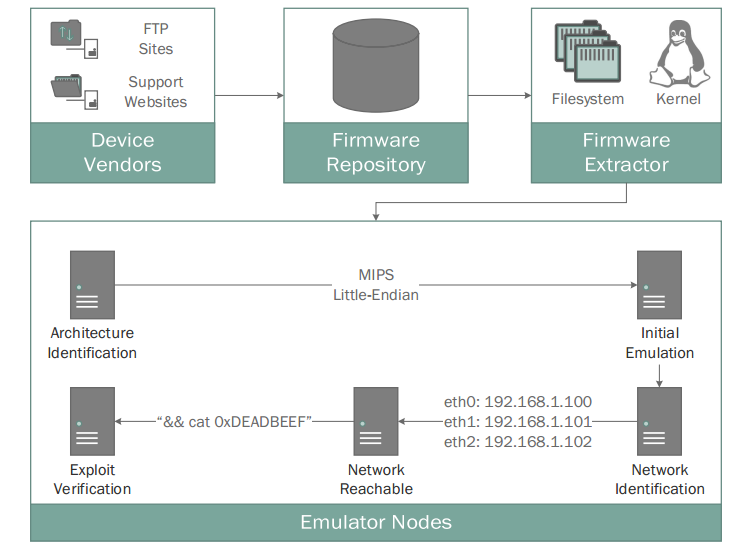


图1-1 FIRMADYNE体系结构图

FIE和Firmalice是国外应用符号执行思想实现固件漏洞检测的系统。基于KLEE的开源工具实现的FIE可以发现固件当中内存安全违规以及外围设备误用漏洞，其利用状态截断和内存模糊方法可以大幅度地提升代码覆盖率，而其局限性主要源于配置错误以及本机部署于符号执行引擎的差异，这些可以导致固件漏洞检测的最终效果产生误差[5]。而其所依赖的符号执行引擎KLEE是为自动分析MSP430微控制器这一广泛使用的固件设备所设计的，其可以完成对设备中的简单固件程序所有可能路径的分析过程，而使用其的局限性是要获取目标设备的固件程序源码，并且人工验证所有报告的漏洞[18]。

Firmalice是第一个面向二进制程序的固件符号话分析系统，基于符号执行引擎angr实现，设计了新的检测固件绕过认真缺陷的模型，同时不需要再实际的设备上执行程序，可以发现恶意硬编码的凭证、缺少身份认证的程序访问点以及被故意隐藏的身份验证接口，而其局限性则是为设备制定安全策略时需要人工操作的介入，因此其可扩展性便受到限制，无法用于对嵌入式设备的大规模分析[16]。

而在国内研究中，以动态的方式检测固件漏洞也主要是利用模糊技术。

利用模糊技术进行固件的漏洞检测时，受到嵌入式设备的运行环境及其本身资源条件的限制，使得发现的异常无法及时确认并利用。因此，研究者提出一种基于污点跟踪的固件漏洞定位方法，该方法的优点是可以在仿真环境中完成动态分析过程，从而快速定位异常位置，确定触发异常的原理，评估利用方法[12]。

另外，为解决目前嵌入式设备固件漏洞检测方法硬件依赖程度高，通用性不强的问题，研究者提出一种基于内存模糊测试的嵌入式设备固件漏洞检测方法，此方法根据嵌入式设备运行环境封闭的特点，采用绕过执行的方法仿真运行嵌入式程序，动态监视并控制嵌入式程序的执行，实现嵌入式程序的内存模糊测试[20]。

### 1.3.3 综合分析

Costin等研究者提出了一种新的方法来对物联网设备内的嵌入式web接口进行大规模的安全分析，他们设计了一个框架来实现固件的可伸缩和自动化分析。该框架的开发正是为了使用纯软件方法和现成的静态和动态分析工具来发现嵌入式设备中的漏洞[17]。通过静态分析，研究人员在185个固件映像中发现9271个问题；通过动态分析，在web界面中发现了225个高危漏洞。

DIFUZE使用通过静态分析提取的接口信息来模糊移动内核驱动程序，然而，这项技术是针对内核驱动程序定制的，不适用于二进制程序[10]。

VUzzer是一种应用程序感知的进化模糊测试策略，它不需要任何有关应用程序或输入格式的先验知识，同时，为了最大化覆盖范围并探索更深入的路径，其利用基于静态和动态分析的控制和数据流特征来推断应用程序的基本属性[38]。

面向二进制代码的漏洞检测工具Bats综合和动态和静态分析技术。具体来说，通过利用动态符号执行技术对二进制程序的中间代码完成安全性分析之后，再使用静态分析技术识别对应的高层语义结果在二进制程序代码中的表现，其实现的核心包括将二进制代码提升至易于分析的中间程序语言Vine IR以及对符号执行获取的路径进行数据流分析，最终的实验结果表明，可以将符号执行技术应用于二进制代码漏洞检测，并且这种综合分析方法能够实现较好的漏洞检测效果[13]。

### 1.3.4 国内外研究现状分析

综合以上对国内外对于固件漏洞检测方法的相关研究现状，目前的研究重点主要聚焦于其研究所设计实现的固件漏洞检测架构的检测有效性以及性能等方面，在此方面这些方法基本都能取得良好的效果[14][19]。

总体来看，国内外固件检测中的研究方向而言，缺少了从固件本身为切入点，就其脆弱性进行深入分析，从而发现相关固件中可以触发漏洞的潜在威胁，这样，就不能有效的对固件漏洞本身所带来的威胁影响以及内部漏洞威胁的关联实现有效的识别，也就不利于后续对这些固件漏洞的修补工作。

就相关研究人员设计实现的基于污点分析引擎的固件静态检测系统Karonte来说，这种方法的漏洞检测过程是经过固件拆包、边界二进制程序发现、构造二进制程序依赖图和二进制程序数据流图之后，通过不安全交互检测模块发现不安全函数触发的溢出类漏洞和受到外部数据控制迭代条件的循环诱发的拒绝服务类漏洞。在这些研究人员的最终报告中，阐明了可以在之前几个对固件预处理与面向二进制分析技术的基础之上，为完善系统的不安全交互检测模块中，这一模块的设计与实现基于之前的各个模块中获取的固件分析结果和相关污点数据信息，根据相关脆弱性点的漏洞触发原理来识别特定类型的漏洞。

因此，综合以上对国内外固件漏洞研究现状的分析，本研究课题将以面向二进制的程序分析和静态污点分析技术作为基础，以嵌入式设备的固件系统本身作为论文研究的切入点，发现固件中存在的脆弱性点，同时，通过对这些脆弱性对应的原理和触发机制进行相应的研究，得出对应的漏洞触发条件，并通过静态分析方法识别出在固件敏感地址的操作（例如，在外部引入的数据在固件程序中传播内存地址上的读取和写入），进而发现固件当中可以为外部攻击者所利用来触发漏洞的程序点，最终实现基于脆弱点定位的固件漏洞检测。

## 1.4 课题主要研究工作

课题针对当前在互联网领域中得到广泛的物联网设备中，嵌入式固件系统中存在的安全隐患这一背景，通过当前国内外对固件漏洞检测的各方面研究方向进行综合分析之后，选取基于脆弱点定位的固件漏洞检测方向作为研究课题，完成固件漏洞检测系统的设计，并在固件数据集上完成对已经完成实现的系统进行各个模块的测试从而验证系统的功能。

具体来说，本课题的研究工作主要包括以下几个方面。

（1）基于面向二进制分析的固件分析技术，深入分析固件这些脆弱性点能够诱发漏洞的原理，确定在对嵌入式固件系统的分析步骤中，需要收集的各项脆弱性信息。

对于首项工作，关键在于在完成对数据集中固件样本的预处理步骤获取二进制程序集后，从各个脆弱点类型的能够触发漏洞的原理入手，设计出切实可行的固件脆弱性分析方案，从而为后续收集固件中关键的脆弱性相关信息、发现固件中存在的脆弱性点以及最终在结合面向二进制的固件漏洞分析技术后发现固件的脆弱点并判断其是否满足漏洞触发的条件。其中，以漏洞触发的诱因作为分类依据，固件的脆弱点可以分为不安全函数和整型溢出类脆弱点、受控循环类脆弱点、栈和堆类脆弱点以及释放类脆弱点在这项工作中，首先，从由固件样本中获取的二进制程序作为分析对象，分析以上提到的所有会被外部攻击者利用的脆弱性点，同时，通过面向二进制的程序分析方法，阐述脆弱性点得以触发漏洞的根本原理，最后给出在固件中需要收集的关键信息以及具体的识别方法，从而为后续可利用的固件脆弱性点的识别提供重要依据。

（2）在面向二进制的固件脆弱性分析并且在结合静态污点分析和符号执行相关技术的基础之上，设计出切实可行的固件脆弱点相关信息收集以及这些脆弱点的可利用性识别的方案完成固件脆弱点定位后，实现固件漏洞检测系统，最后在固件数据集上完成对系统的各个功能模块进行测试以验证系统的有效性。

对于这项工作，关键在于充分利用好首项工作中对面向二进制的固件脆弱点原理分析的理论基础，设计出针对上文提到的几种固件的脆弱点信息获取与可利用性识别方案，从而在已有固件脆弱点原理的前提之下完成对固件可利用的脆弱性的发现。除此之外，在本项工作中对已经设计实现的各个功能模块的验证同样重要，因此，本研究将采用固件样本数据集[22]当中典型的嵌入式设备固件样本来完成对系统的验证测试，也就是判断系统是否能够获取到所需的所有脆弱性信息，进而判断在敏感区域内的固件脆弱点是否满足触发漏洞的必要条件，从而发现嵌入式固件系统中的漏洞。其中，基于脆弱点定位的固件漏洞检测系统的预处理和固件样本分析模块将在已有的研究的基础上实现，包括面向二进制的固件分析技术[21]以及静态污点分析和符号执行引擎[24]。具体来说，预处理和分析模块完成对输入的固件样本的拆包解析后获取二进制程序集，然后，发现其中直接与外界交互的边界二进制程序（带有对引入外部数据的程序点的标识），随后，基于静态污点技术和符号执行技术构建固件二进制程序依赖图以及二进制数据流图，前者描述固件样本中二进制程序集中程序之间的相互依赖关系，后者则以进程之间的共享通信数据键作为程序的联系来表示它们之间的数据流向。

因此，本课题中的研究重点即为完成固件脆弱点信息收集与漏洞的可利用性识别模块。针对前文对固件几种脆弱性点原理的分析，需要将这一模块依次分成四个部分来设计与实现，具体包括：不安全函数脆弱性点的可利用性识别、可以诱发拒绝服务类漏洞的受控循环的识别、栈和堆脆弱性点相关的实时执行状态信息的获取以及这些栈和堆相关脆弱性的脆弱性点识别。

在这些模块的关键部分当中，不安全函数脆弱性点的可利用性识别部分是在原有研究基础之上，加入本文的整型溢出脆弱性点的相关理论完成对模块的进一步改进；而能够受控循环识别上，改进是在能够诱发缓冲区溢出漏洞的循环相关理论的基础之上，添加相关在对诱发漏洞循环的判别操作。而其余的模块便是本研究课题的核心，即获取栈和堆脆弱性点的实时执行状态信息以及对应脆弱性点的可利用性识别。在脆弱性信息获取模块中，需要根据前文面向二进制的固件脆弱性分析部分获取必要的能够发现固件中的脆弱性点以及可以判断它们触发漏洞的相关信息。进而，在可利用性识别模块中，在检测到由外部数据在固件中的数据流向路径这些敏感区域内有相关程序操作之后，随机触发相应的回调功能函数，从而提取相关脆弱性信息来判断相关脆弱性点是否会触发漏洞。

因此，在最终的系统模块验证小节中利用固件数据集进行相关试验后，需要重点关注本文设计实现模块的功能是否得以完全实现，包括：

（1）获取了固件中关于栈和堆脆弱性信息以及固件的指令架构（即整型数据的范围），这些信息能够作为之后的可利用性识别模块的重要判断依据

（2）准确识别了在固件二进制程序敏感区域上的相关操作，能够结合已有的脆弱性信息判断脆弱点是否会触发相关漏洞（整型溢出、栈溢出、堆溢出、双重释放以及释放后访问漏洞）。

结合主要工作的内容，本文的主要贡献如下。

（1）基于二进制程序分析的技术对固件的几种脆弱性类型进行分析，详细阐述了固件脆弱性的漏洞触发原理，从而提供了要判断这些脆弱点能够被触发漏洞需要收集的关键信息的依据，也是后续设计与实现的面向二进制的固件漏洞检测系统能够以脆弱性分析方法发现其中漏洞的理论基础。

（2）完成了栈和堆以及相关脆弱性信息的获取、脆弱性点的定位以及脆弱点可利用性识别模块的设计，通过在固件数据集上的实验，验证了这些关键模块的有效性，即成功获取了固件中关于栈和堆脆弱性信息以及固件的指令架构、准确识别了在固件二进制程序敏感区域上的相关操作，在这些基础上系统能够通过脆弱性可利用识别功能模块来发现固件中的漏洞。

总的来说，在完成本课题的研究工作之后，可以发现嵌入式设备的固件系统中存在的可以触发漏洞的脆弱性点，一方面，可以为固件的安全性评估方面，提供固件的脆弱性信息，从而有利于后续完善固件系统；另一方面，则可以避免攻击者通过网络的手段，攻击这些物联网设备中固件这一关键部分，从而避免引发相关安全事件。

## 1.5 论文组织结构

论文主体划分为四个部分，其组织结构如图1-2所示。如图所示，本文各章节所介绍的研究内容具体如下。



图1-2 本文的组织结构图

第1章绪论将会介绍研究课题的相关背景与研究意义、固件漏洞检测的国内外研究现状与分析以及论文的主要工作和贡献。

第2章是面向二进制的固件漏洞检测技术基础内容。本章将介绍课题研究所依赖的基础技术，包括面向二进制的固件分析技术（固件解析、边界二进制程序发现、二进制程序依赖图构建以及二进制程序数据流分析）以及基于污点分析和符号执行的固件漏洞检测技术。

第3章是面向二进制程序的固件脆弱性分析。本章首先从固件脆弱点的可利用性展开介绍，进而对固件的各项脆弱性点的漏洞触发机制和识别原理展开详细分析，包括不安全函数和整型溢出脆弱性点、受控循环类脆弱性点、栈和堆溢出类脆弱性以及释放类脆弱性点，进而在这些原理之上设计出形式化的识别方法。

第4章是基于脆弱点定位的固件漏洞检测系统的设计与实现。本章先从系统整体设计出发，介绍系统各个模块所负责的相应功能以及工作流程，随后详细介绍课题完成的堆和栈脆弱性点信息的获取以及可利用性识别模块。最后，通过在固件数据集上的嵌入式固件系统样本进行了对系统的各个功能模块进行了实验测试，验证系统的有效性。

# 第2章 固件漏洞检测技术基础

本文第2章将会着重介绍研究课题设计的固件漏洞检测系统在二进制层面上相关的固件漏洞检测的技术，从而为本系统提供实现的技术基础。其中，第1节将从固件结构解析、边界二进制程序发现、二进制程序依赖图构建以及二进制程序数据流分析这四个方面展开介绍面向二进制的固件分析技术，第2节将介绍静态污点分析技术，第3节将介绍符号执行技术，最后一节将对本章节的内容进行总结。

## 2.1 面向二进制的固件程序分析技术

在固件二进制程序分析方面，本研究课题设计实现的固件漏洞检测系统基于检测嵌入式固件中跨二进制程序不安全交互固件静态检测工具karonte的相关技术，其中包括固件解析、边界二进制程序发现、二进制程序依赖图构建以及跨二进制程序数据流分析[21]。

### 2.1.1 固件解析

固件是那些写入到可擦写只读存储器（EROM）或电可擦可编程只读存储器(EEPROM)中的程序，也就是嵌入式设备中的软件程序。

对于嵌入式系统的固件，其结构如图2-1所示。如图2-1所示，固件通常由固件头、主体以及其他附加数据组成，其中，主体部分包括启动引导程序（Bootloader）、操作系统内核以及根文件系统组成[25]。



图2-1 嵌入式固件系统结构

其中，固件头则是用于存储嵌入式设备固件的相关参数，如生产厂家、设备型号、设备版本号、校验和等

启动引导程序是在操作系统运行之前执行的一段小程序，通过完成硬件设备的初始化、建立内存空间映射表的过程，建立适当的系统软硬件环境，为最终调用操作系统内核做好准备。所以，启动引导程序主要完成初始化相关硬件以及引导加载操作系统内核两项工作。

内核是操作系统的核心模块，在系统的体系结构中，它维系着进程、内存、设备驱动程序、文件和网络系统等的正常运转。内核关系到整个存在系统的运行效率与稳定状态，是连接应用程序和硬件的桥梁。

根文件系统是内核启动时的第一个被挂载的文件系统，保存着内核代码映像文件。在完成对其的挂载后，系统会将基本的脚本和服务加载到内存中并执行，从而完成初始化过程。

在系统接收到固件样本的输入后，将先对固件映像完成预处理操作，即利用固件解析工具binwalk完成堆固件样本的拆包，从而完成固件的二进制程序的获取。binwalk是一个快速且易于使用的固件解析工具，可以将其用于固件分析，逆向工程以及固件映像的提取等方面[26]。

### 2.1.2 边界二进制程序发现

所谓边界二进制程序，就是嵌入式设备固件系统中直接与用户交互的二进制程序文件，例如网络服务器以及后端应用程序等。嵌入式固件设备的交互流程如图2-2所示。



图2-2嵌入式固件交互流程

从图2-2中可以看出，这些边界二进制程序文件可以接收外部网络的数据，同时，通过解析器可以处理用户的请求并发出响应。

边界二进制文件发现模块通过分析经过预处理后的固件样本，识别并导出边界二进制程序集合，这些二进制程序包含接收外部输入的用户请求的相关处理逻辑。这样，这些导出的边界二进制程序集合中的程序存在被外部攻击者的所利用的条件，即将其能够控制的数据引入到固件中[21]。因此，在边界二进制程序集合中的文件包含了能够引入攻击者控制的数据程序点的标识。

根据固件静态检测工具karonte，在边界二进制程序检测方面，主要通过二进制函数基本块中所对应的关键指标来计算出解析分数（parse score）[27]。然后，根据这些二进制程序的解析分数，应用DBSCAN算法对二进制文件进行聚类，最后将解析分数最高的聚类视为边界二进制数据集[28]。

其中，二进制函数基本块的关键指标包括：基本块数（bb）、内存比较操作数（cmp）、分支数（br）、网络相关的关键字数（net）以及接收recv和内存比较操作之间的数据流（conn）。

对于二进制块b的解析分数，计算方法如公式（2-1）和（2-2）。其中，psj是二进制块b的第j次函数的解析得分。

|  |  |
| --- | --- |
|  | (2-1) |
|  |  |
|  | (2-2) |

具体说，根据PIE（嵌入式固件系统中的解析器识别），相关指标初始化如下。

（1）net初始化为0，而当每次与代码中存在的网络编码字符串进行比较时，使得的值增加1。

（2）conn初始化为0（其值为1的情况：套接字读取和内存比较操作之间存在数据流）。

（3）每个常数ki设置为最大化解析检测能力，而kn和kc的选择则是根据它们各自提示引用网络编码关键字和解析网络数据的二进制文件的函数功能的效果[27]。

### 2.1.3 二进制程序依赖图构建

在完成对固件中负责直接与外部设备进行交互的边界二进制程序的定位，并且为其附带上引入外部数据的程序点的标识后，系统将在这些二进制程序集上构建二进制程序依赖图（Binary Dependency Graph）。二进制依赖图描述固件样本中二进制程序之间的数据依赖关系，也是作为之后数据流分析功能模块中对固件跨二进制程序数据流分析的基础。构建二进制程序依赖图主要包含如下三个步骤。

首先，在上文所述的边界二进制程序发现模块中获取的边界二进制程序集基础之上，将从这些二进制程序中与外界交互的程序点引入的污点数据与网络相关的关键字进行比较，同时，利用静态污点分析技术检测这些二进制程序之间是否依赖于相同的进程间通信范式来共享数据，从而完成通信过程。因此，在这一步骤中的关键点包括：判断相关二进制程序是属于引入外部数据类程序（如图2-2所示的web服务器）还是属于数据解析和处理类程序；检索二进制程序在数据交互中所使用的共享数据键；利用共享数据键来扫描固件样本，从而找到依赖于同一数据键的其他二进制程序。

然后，通过通信范式发现模块（Communication Paradigm Finders）来检测进程中通信的范式。通信范式发现模块是为检测和描述二进制共享数据所使用的通信范式提供必要逻辑的实例。要实现这一功能，通信范式发现模块需要考虑二进制程序以及路径，也就是二进制基本块序列，并检查该路径是否包含必要的代码以通过CPF表示的通信范式共享数据[21]。若该路径满足这一条件，该模块将通过以下特定于功能来收集通信范式的详细信息。

（1）恢复通信数据键。

范式发现模块恢复在关联的通信范式下引用由二进制文件设置或检索的通信数据键。

（2）确定数据流向。

通信范式发现模块识别相关程序点标识，在这些程序点上，将访问由收集的数据键表示的数据。如果存在这样的程序点，则其可以确定每个程序点在通信流中的作用，即确定其是引入类还是数据解析和处理类二进制程序。

（3）扩大二进制程序集。

通信范式发现模块查找固件样本中的其他二进制文件，这些二进制文件引用先前标识的任何数据键。这些二进制文件很可能与当前正在考虑的二进制文件共享数据，因此需要进行进一步分析。 然后，结合不同通信范式收集的信息，以在二进制程序依赖图中创建边，从而恢复跨不同二进制程序的数据流。

最后，在完成上述步骤确定二进制程序之间的交互与依赖关系等信息之后，将共享数据键值作为引入外部数据类程序和数据解析和处理类程序的边来完成二进制程序依赖图的构建。

### 2.1.4 二进制程序数据流分析

完成对描述二进制程序依赖关系的二进制程序依赖图的构建之后，为了发现在二进制程序之间的可以触发漏洞的不安全交互，需要在二进制程序依赖图的基础之上添加二进制程序数据流向的信息。

可以考虑到，如果完全遍历二进制程序依赖图中所有可能的路径，可能会导致路径爆炸问题[29]，因此，可以认为对来自于外部用户的数据施加较少约束的路径更可能会被攻击者利用，从而作为漏洞的触发路径。所以，跨二进制程序的数据流分析关键在于检索二进制依赖图中从引入类到数据解析和处理类程序的的路径约束后，把应用最不严格约束的路径作为边来构建二进制程序流图（Binary Flow Graph）。

为了检索此类路径，需要收集二进制程序跨程序路径上对相应数据约束的集合，并把其中约束最小的作为到其他二进制程序的传播路径。所以，二进制程序流图通过对多个二进制文件之间共享的数据施加了最不严格的约束集来扩展二进制程序依赖图。

在二进制程序流中，（[b1，loc1，cp1，c1]，[b2，loc2，cp2，c2]，k）表示与数据键k相关联的数据可以通过通信范例cp1使用约束集c1在位置loc1处的二进制b1流出，流入通过带有约束集c2的通信范式cp2到达位置loc2处的二进制b2。其中，[b，loc，cp，c]表示[二进制程序，位置，通信范式，约束]，k表示通信数据键。二进制程序数据流图的构建算法基于混沌迭代的概念，它由初始化和约束传播两个阶段组成[30]。

## 2.2 基于污点分析的漏洞检测技术

所谓污点分析，就是追踪与分析污点数据信息在程序中流动的技术。在固件漏洞检测中，将从固件外部输入的数据标记为污点后，通过污点分析技术来追踪污点数据流可以获取这些外部数据在固件内部的路径流向，从而可以判断这些可以被攻击者控制的数据是否会到达可以触发漏洞的程序点，进而触发固件漏洞[32]。

污点分析主要由三个部分组成，结合固件漏洞检测的实际具体如下[35]：

（1）识别在固件边界程序引入外部数据的程序点（source点），并将相应的外部数据标记为污点数据。

（2）根据进程间通信的共享数据键追踪并分析污点数据在固件中的传播。

（3）在可能会触发漏洞的程序点（sink点），判断相关是否会受到受控的污点数据影响而触发漏洞。

在本研究课题中所使用的污点引擎是在BootStomp的相关研究工作中提出的静态污点分析引擎，该静态污点分析引擎可用于跨二进制数据流分析以及不安全交互检测[24]。

一方面，给定漏洞触发点s（例如返回不可信数据的函数）和数据引入程序点p，静态污点分析引擎将从程序点p开始执行符号路径探索，然后，每次探索到s点时，污点引擎会为从触发点s接收数据的内存位置分配一个新的污点标识。

另一方面，静态污点分析引擎将跟随着二进制程序数据流来传播污点信息，当污点数据内存地址被非污点数据所覆盖或其值受到约束的时候，引擎将通过删除污点标识的方法来取消对这些内存位置的赋值。

与一般的静态污点引擎相比，BootStomp中的污点引擎包含一下两项改进：

（1）应用了路径优先级策略。

路径优先策略通过对更重要的路径进行优先处理，解决了基于路径探索的污染引擎在处理隐式控制流时所面临的污点标记不足问题（undertaint problem）。在污点分析工作流程内，如果更有价值的变量在路径p1中被标记为污点而在路径p2中没有被标记为污点，则路径p1被认为比路径p2更值得关注[31]。

（2）引入了污点标签依赖项概念。

在污点引擎中，在具有不同标签的污点变量之间创建依赖关系（即污点标签依赖关系），进而，通过跟踪这些依赖关系对于在多标签污点跟踪系统中制定有效的污点策略具有重要作用，从而避免了过度污点标记问题（overtainting problem）[33]。

## 2.3 基于符号执行的漏洞检测技术

所谓符号执行，就是将符号表达式作为具体值来模拟程序执行，因此，符号执行可以应用于程序的测试与验证以及漏洞的检测。在漏洞检测领域中，符号执行通过符号表达式作为执行参数来模拟程序的执行，将程序的输出以这些符号表达式来表示，进而完成语义分析，也就是漏洞检测的基础[34]。

符号执行的具体步骤包括：将符号表达式代替具体值作为程序的输入；在程序执行的过程，收集路径约束（Path Constraint）组成路径表达式；分析路径表达式，完成对路径安全性的评估。

符号执行可以分为以下两种类型：仅对单个功能的代码进行分析的过程内分析、分析当前功能入口点的当前功能的调用信息和环境信息的进程间分析。当将符号执行用于代码漏洞静态检测时，它更多地是关于程序的整体分析，同时也着重于与安全性相关的代码检测。

在符号执行中，每一条路径都是一个由布尔值 true 和 false 组成的序列，其中第 i 个 true（或false）表示在该路径的执行中遇到的第 i 个条件语句。这些路径可以由符号执行树来描述。这种树形结构包含变量值、路径约束等执行状态信息。在符号执行树中，节点表示程序中的语句，边则表示程序中语句之间的先后执行次序或跳转关系。

一个具体的符号执行示例如图2-3所示。



图2-3 代码与其对应的符号执行树

如图2-3所示，三条路径分别可以被输入{x = 0, y = 1}、{x = 2, y = 1} 和 {x = 40, y = 20} 触发，所对应的路径约束pc分别是(x == 2\*y)、(x ≠ 2\*y and x <= y+10)、(x ≠ 2\*y and x > y+10)。

因此，在符号执行中，对于程序中带有条件的控制跳转语句，可以通过符号表达式将这些条件转换为对符号取值范围的路径约束，进而分析约束来判断程序的路径是否可行，这种分析过程也称为路径的可行性分析。可行性分析是符号执行的关键部分，结合其他固件漏洞分析技术使用后，通过符号执行的路径可行性分析过程可以排除不满足条件的程序执行路径。

## 2.4 本章小结

本章旨在介绍面向二进制的固件漏洞检测技术，即本文研究课题的基础，为后续研究课题的固件脆弱性分析和固件检测系统的设计与实现提供了理论和技术的支撑。围绕这一主题，本章首先沿着固件分析的工作流程对所涉及到的面向二进制的固件分析的相关技术做了重点介绍。进一步地，对于在面向二进制的漏洞检测技术中的静态污点分析和符号执行技术也做了相应的介绍。

# 第3章 面向二进制程序的固件脆弱点分析与定位

本文第3章将开始对课题的第一项重点研究工作，即面向二进制的固件脆弱点定位的相关原理进行介绍。

具体来说，本章第1节先从固件中可利用脆弱点的基本概念与原理方面展开，并就其漏洞触发机制与识别以及固件可利用脆弱点所带来的影响两个方面做重点介绍。本章第2节将就固件脆弱点的原理展开研究，深入分析相关脆弱性如何能够触发漏洞，其中，这些脆弱点包括不安全函数和整型溢出、受控循环类、栈溢出类、堆溢出类以及释放类脆弱点。本章第3节将对各类型的固件脆弱点的识别方法进行形式化的归纳。本章最后一节将对章节的内容进行小结。

## 3.1 固件的可利用脆弱点概述

结合第2章的内容，固件的脆弱性分析是指对固件之中敏感内存地址区域上，发现出满足可以在一定条件后，可以使得固件程序出现漏洞的内存地址。而这些位置也就是固件的脆弱性点，即脆弱点。

### 3.1.1 固件脆弱点的漏洞触发机制与识别

在引入外部的不安全的受控数据后，固件中存在的脆弱点若是满足被攻击者利用的条件，就被触发相应的固件漏洞。因此，所谓可利用的脆弱性点，就是指攻击者可以通过这些脆弱性点来在固件中诱发漏洞。其中，诱发漏洞的类型包括在任意内存地址执行任意代码或是读取特定数据。以下将会就如何利用这些脆弱性相应的触发漏洞机制识别方法，以及这些漏洞会给嵌入式设备的固件系统带来的实际破坏这两个方面展开来介绍固件中可利用的脆弱性。

正如以上的研究内容所述，脆弱性点的漏洞触发机制的一般模式可以归纳为：如果当固件内的敏感内存区域中执行了相关操作，同时，没有设置相应的保护机制来应对某些特殊情况，那么固件当中的这些敏感区域内也就会存在潜在的脆弱性点，并且它们可以被攻击者利用，进而触发漏洞。

通过在完成对固件中二进制程序的分析收集固件脆弱性信息，并结合对应的漏洞触发机制，就可以发现固件当中的漏洞。因此，这些漏洞的发现可以划分为以下三个步骤：

（1）固件二进制程序污点路径构造。利用静态污点分析技术，构造污点数据在固件当中的路径流向，保证外部的受控数据沿着生成的路径传播。同时，需要给路径添加长度限制以保证路径的有限性，从而避免路径爆炸等问题。

（2）敏感内存注释。在这一步中，将完成用于重点分析的敏感内存区域的构建。

（3）漏洞检测。这一步中，需要检查在相应的程序状态下执行的操作是否导致敏感内存区域被来自外部输入的数据（即污点数据）写入，同时判断是否会触发固件漏洞。如果满足漏洞触发条件，则将报告相应的漏洞以及将触发此漏洞的外部输入数据的相关信息。

因此，针对以上所述的固件脆弱性的触发机制的一般模式以及漏洞检测的一般步骤，要实现对固件中漏洞的检测，需要完成如下几个关键步骤来确定固件是否存在脆弱性，收集相关的漏洞触发条件信息以及对这些实际存在的固件脆弱性点可利用性的识别[39]。

（1）完成固件的敏感内存区域的识别。在基于静态污点分析方法完成对固件二进制程序间的数据流分析之后，可以得到从某个带有能够引入外部数据的边界二进制程序开始，外部污点数据在固件二进制程序之间的数据流向关系图，也就是二进制程序数据流图（BFG）。所谓敏感内存区域，就是上述的污点数据流向路径的固件二进制程序内存地址中，如果含有如内存复制类（memcpy）等危险函数，使得攻击者可以通过受控数据来使得固件触发漏洞，那么固件的这些内存地址就成为敏感的内存区域。

（2）完成能够使得脆弱性能被攻击者利用的特殊情况的识别。在完成固件样本中的二进制程序中敏感内存区域的识别操作之后，需要判断固件样本中的二进制程序是否缺少相关的程序保护机制，进而可能使固件中出现一些诸如会使得固件本身的脆弱性得以被利用的场景。举例来说，对于从外部引入的污点数据若是没有得到正确的处理，或是缺少对这些污点数据的严格约束条件，那么在这种特殊的情况之下，固件中的脆弱性就会在攻击者的某些行为下触发漏洞，从而给物联网设备带来不良影响。

（3）完成敏感区域中进行相关敏感操作的识别。完成两个前序步骤的相关工作之后，需要将对满足被攻击者利用条件的固件脆弱性所对应的这些内存地址上发生的相关操作进行准确识别，从而发现固件脆弱性触发漏洞的最后一个必要条件。结合本课题的实际工作中的研究方向，对于固件脆弱性得以触发的敏感操作，需要考虑的主要是对敏感内存地址的访问操作，也就是内存的读取与写入操作。

（4）完成固件脆弱性点的可利用性的识别函数的设计。识别到这些对敏感内存地址的访问操作之后，需要为这些内存地址添加程序断点，在这一程序断点指定相应的回调函数，从而完成脆弱性的是否会触发漏洞的最终判断。因此，要完成漏洞检测，也就需要完成固件脆弱性点可利用性判别功能函数的设计，并将其指定为回调函数，进而在完成对相关脆弱性执行状态信息的获取之后，能够将固件所有已知的脆弱性相关信息统一整合起来，进而发现固件中存在的漏洞。

### 3.1.2 可利用的固件脆弱点的影响

结合本节开头所介绍的内容，在固件当中的脆弱性满足可利用的条件之后就会触发漏洞，而触发的这些固件漏洞类型可以使攻击者达到入侵相应物联网设备的目的。相应的，在完成入侵这些设备的行为之后，攻击者便能够在这些固件中相应的漏洞触发内存地址实现执行任意代码或是读取信息这些预期目标。

一方面，执行任意代码是指攻击者可以通过外部输入的数据（例如文件系统、网络套接字、环境变量等），劫持应用程序的控制流，进而将这些受控数据作为应用程序的代码来执行。也就是说，在攻击者利用固件内的脆弱性完成对程序控制流的劫持之后，可以将其引导至攻击者得以控制的敏感区域。在这些内存地址中，攻击者可以通过固件边界二进制程序的引入程序点来输入自己所控制的数据，从而能够使其在固件当中执行，最终达到攻击者的预期目标。

另一方面，读取数据允许外部攻击者通过应用程序输出获取机密信息（例如，加密密钥等），或是内存布局的信息（例如：堆栈地址、堆地址、动态库加载地址等），通过这些信息，可以帮助攻击者绕过一些有效的应用程序保护机制，如ASLR（地址空间布局随机化）或DEP（数据执行保护）。这样，即使是在最新和加固的操作系统上的嵌入式固件系统程序之中，外部攻击者也能够完成他们的攻击行为。

## 3.2 固件脆弱点的漏洞触发机制与识别

在固件脆弱性分析小节中，对各个类型的固件脆弱性的详细介绍将从基本原理开始，包括相应固件脆弱性的安全隐患来源以及具体触发机制等，进而针对性的设计出切实可行的固件可利用脆弱性的识别方案，也就是从脆弱性的原理出发，完成对固件脆弱性相关的执行状态信息的收集之后，设计脆弱性点可利用性的识别功能模块，从而发现固件当中的漏洞。

### 3.2.1 不安全函数和整型溢出

在嵌入式固件系统程序中，通过将外部引入的受控数据用作参数而调用的相关不安全函数，在某些条件下会使得固件出现诸如缓冲区溢出之类的漏洞，因此，这类不安全的函数就是固件当中的一种典型的脆弱性。

具体而言，当来自外部输入的受控数据缺少严格的限制条件时，攻击者就可以利用这一安全隐患将危险数据引入到固件程序当中，进而，当这些受控数据到达固件程序之中不安全函数的内存位置之后，就可以利用不安全函数本身的安全隐患，从而触发固件漏洞。举例来说，就不安全函数当中内存复制类函数（例如，memcpy等）而言，当攻击者所控制的数据流入这些不安全函数的内存地址之后，根据调用函数的相关参数信息，如果目标缓冲区的边界范围不足以容纳受到约束后的外部受控数据的大小，那么攻击者就可以触发固件的溢出类漏洞。

因此，结合上文所述的源于不安全函数的威胁而触发的缓冲区溢出类漏洞的实例，对于这类脆弱性的定位以及漏洞的发现主要依赖于以下几个关键部分。

（1）确定不安全函数及其在固件程序中的内存地址。针对嵌入式固件系统的二进制程序，能够触发漏洞的存在安全隐患的函数主要包括字符串处理函数、字符读取函数以及内存操作类函数。

对于字符串处理函数，安全威胁主要来自于未能够给将要处理的参数添加长度约束，这样，在没有大小限制的情况下就会触发缓冲区溢出漏洞。这类函数主要有字符串复制函数strcpy、字符串拼接函数strcat以及字符串格式化函数sprintf与vsprintf等等。

字符读取类函数的漏洞触发原理与字符串处理类函数类似，主要源于对从外部读入的字符的边界没有约束，没有做边界检查，所以在一定条件下，通过这类函数可以触发缓冲区溢出类漏洞。这类函数主要有读取字符类gets、getchar等，格式化读取字符scanf、fscanf、vscanf等以及读取环境变量函数getenv。

内存操作类函数主要包括内存复制类函数（memcpy）和内存空间申请函数（new）。其中，前者是由于在调用这类内存复制类函数时，其中的参数可以是外部引入的数据，同时可以省略对其施加必要的条件约束，因而可以触发缓冲区溢出类漏洞。除此之外，调用后者这些内存空间申请函数时，攻击者可以通过输入数据指定分配的敏感内存区域的大小，进而在后续状态中，如果执行了对这些区域的操作，且长度超过这些敏感区域的边界时，就会诱发缓冲区溢出漏洞。

（2）追踪受控数据在固件内的流向。在确定不安全函数在固件二进制程序中的内存地址之后，利用面向二进制的程序分析方法以及污点分析和符号执行引擎，确定自外部输入的受控数据在固件当中各个二进制程序的流向，从而发现那些流入到不安全函数脆弱性的内存地址当中路径，进而得出可能会触发漏洞的受控数据路径的集合。

（3）获取污点数据的路径约束。在前序步骤中获取的敏感路径的基础之上，通过符号执行技术获取它们对应的路径约束，同时结合当前调用的函数操作以及引入的污点数据，进而发现固件程序中由于不安全函数类脆弱性触发的漏洞。

另一方面，考虑到嵌入式固件本身的特性（例如指令架构等），若相关操作的数据越过了固件本身所限制的数据长度，那么将会有整型溢出等问题。具体来说，对于某一固件体系结构，整型数据类型一般具有固定大小的长度（例如32位长度），所以对应的整型变量的数据就具有一个最大值（2^32）。在固件中，如果存在超过这一最大值的运算才操作，那么就会产生变量值回绕（wrap-around）的情况，而这种情况也就被成为整型溢出。一般来说，整型溢出本身是不可利用的，但其通常会使得其他类型脆弱性具备漏洞触发的条件，其中就包括不安全函数类脆弱性、栈溢出以及堆溢出类脆弱性。

具体来说，大多数整数溢出是由于这类不安全函数中未加严格的约束就将外部的受控数据作为执行参数而诱发的，因此，通过跟踪引入的污点数据，并检查在使用这些漏洞触发函数中污点数据是否有与固件本身的结构相关的路径约束条件，进而可以完成对整形溢出诱发的缓冲区漏洞的识别[39]。对于整型溢出可触发的其他漏洞类型将在以下各小节中进行介绍。

### 3.2.2 受控循环

在嵌入式固件系统中的循环如果可以受到外部攻击者的影响，那么其就可以被攻击者利用并触发拒绝服务等漏洞，也就是说，在这种情况下固件中存在受控循环类脆弱性。具体来说，攻击者通过引入外部受控数据，利用其在固件二进制程序之间的数据传播路径来索引到这些污点路径上的循环，这样，这些循环的迭代条件就能够受到外部输入的污点数据所影响，攻击者也就能够实现对循环迭代次数的直接控制，进而利用固件的这些受控循环类脆弱性点来触发诸如拒绝服务类的漏洞。

结合以上对固件的受控循环脆弱性的漏洞触发机制的简要分析，这些脆弱性得以被利用的必要条件可以归纳为以下几个方面。

（1）污点路径上存在循环。首先，结合面向二进制的固件数据流分析技术获取到污点路径，当这些路径之上存在循环时，那么固件中相应的内存地址便是包含可能被利用的循环类脆弱性点的敏感内存地址。

（2）循环的迭代条件依赖于污点数据。然后，固件如果存在与循环相关的敏感内存地址时，这些位置上的循环的迭代条件依赖于外部的污点数据，这样，攻击者就能够通过控制外部的数据输入来完成对循环迭代次数的控制。

（3）受控循环的相关操作能够诱发漏洞。最后，在以上所述的已经确定位于敏感内存区域内的循环完全依赖于外部引入的污点数据后，如果满足循环相关的重复操作能够使得固件为了应对这些操作请求而无法提供正常的服务时，那么便会触发拒绝服务类的漏洞。

因此，通过上述对固件的受控循环类脆弱性的漏洞触发机制的归纳分析，可以得出，受控循环的判断关键在于确定循环迭代条件是否完全依赖于外部引入的污点数据，具体方法可以是通过判断为了执行循环内路径目标中所列出的分支是否取决于外部数据。因此，针对这类脆弱性以及其可利用性识别的核心在于以下几个方面。

（1）确定污点数据传播路径。首先，从固件当中能够引入外部的边界二进制程序开始，基于面向二进制的固件静态分析技术依次完成对固件中二进制程序的依赖关系和数据流向关系图的构建，结合静态污点分析技术获取引入的外部数据在固件中的污点传播路径。

（2）识别在敏感内存区域上的循环。对于这类位于敏感内存区域之上循环的识别，可以利用污点分析和符号执行引擎，将路径流经的内存地址作为分析对象，通过分析这一地址之前的历史路径、后续路径的集合以及长度等信息，判断这些敏感内存地址是否在污点路径上周期性的出现，进而完成对位于敏感内存区域上的循环的识别。

（3）确定循环的迭代条件。在特定敏感内存区域上识别到循环后，需要把这些控制循环迭代次数的条件确定下来。对于循环迭代的依赖条件的识别，可以从固件中发现循环相应的内存地址出发，根据当前已经执行的跳转或后续的操作，从而利用固件二进制分析确定为执行上述的分支所必须满足的约束条件。

（4）确定循环的迭代条件的直接依赖对象。要判断在敏感内存区域上的循环是否会触发漏洞，需要确定其是否能被外部攻击者引入的数据所影响。因此，可以通过上述步骤中获取到的循环迭代条件以及该循环路径上对应的污点数据确定其是否存在依赖关系，识别能够诱发拒绝服务类问题的受控循环脆弱性，从而完成对固件漏洞的检测。

### 3.2.3 栈溢出

对于嵌入式固件系统，在敏感区域上执行内存访问操作，即读取或写入数据后，如果出现了栈溢出的问题，并且操作相关的数据将栈相应内存位置上的返回地址覆盖时，那么就会触发栈溢出的漏洞，也就是说，此时固件中存在的栈相关的脆弱性是可以利用的。

具体来说，攻击者通过输入将其所能够控制的数据到固件程序中，然后针对在固件程序中调用的函数，将其返回地址覆盖，进而完成对程序控制流的劫持，并将其引导至攻击者期望的位置，从而触发相关漏洞。

因此，从以上对栈溢出类脆弱性的漏洞触发机制的简要分析可知，嵌入式固件程序中某些敏感内存地址中的函数的调用栈的存储的返回地址值被内存访问操作所影响，从而使得程序的控制流有被攻击者劫持的安全隐患时，那么就可以固件内存在栈溢出类脆弱性，同时栈溢出类脆弱性可以被攻击者利用从而触发漏洞。具体来说，固件中存在的栈溢出类脆弱性得以被利用的必要条件如下。

（1）在敏感的内存区域上涉及到了函数调用相关操作。在来自外部输入的数据进入到固件程序中后，其在固件中程序之间流经的内存地址构成了固件程序的敏感内存区域。这些敏感内存区域为攻击者利用固件之中存在的脆弱性点提供了基本的环境条件。进一步地，如果在这些敏感内存地址中，程序调用了相关函数，即涉及到了程序的调用栈，那么这时固件中就可以说存在了栈溢出类脆弱性点，在特殊条件下这些脆弱性点可能会给固件带来安全问题。

（2）在敏感内存操作上执行了内存访问操作。在确定固件敏感内存区域的前提条件下，如果固件程序在这些内存地址上执行了内存访问操作，那么固件中对应的调用栈的相关参数会受到其影响。因此，要完成脆弱性点可利用性的进一步确认，应当结合固件二进制分析的相关技术，进而实现对这些敏感操作的识别。

（3）对敏感内存的直接访问操作影响到了调用栈的返回地址值。当对固件敏感区域的内存访问操作触发了栈溢出时，溢出部分的数据将会覆盖其他的堆栈内存地址。在这种情况下，并不能完全确定栈脆弱性点能够被攻击者利用。举例而言，溢出的污点数据仅仅是覆盖了少数堆栈内存地址的值，而没有触及到调用栈所存储的函数返回地址，那么栈溢出脆弱性还是不可以利用的。因此，需要通过固件二进制分析技术判断内存访问前后，函数调用栈的返回地址值是否发生改变，从而发现固件的漏洞。

因此，根据上述触发固件当中脆弱性的漏洞触发机制以及必要条件，要实现对固件内栈溢出脆弱性点的可利用性识别，核心主要包括以下几个方面。

（1）确定固件的敏感内存区域。首先，需要利用面向二进制的固件分析技术，结合污点分析和符号执行引擎，对从固件边界二进制程序引入外部数据的程序点开始，完成对这一污点数据的追踪，从而确定外部受控数据在固件内二进制程序中流经的内存地址，进而获取其在固件中所涉及到的内存地址的集合，确定固件中敏感的内存区域。

（2）发现在固件敏感区域上的内存访问操作。然后，根据上述步骤所获取到固件内具体的敏感区域内存地址，利用污点分析和符号执行引擎，为在地址之上执行的内存访问操作（内存读取、内存写入）添加程序断点。同时，需要为固件二进制程序的对应程序断点添加漏洞识别功能函数，并将其指定为回调函数，进而为后续实现固件漏洞的检测提供基础条件。

（3）获取相关地址在内存访问前后的值。最后，需要收集函数调用栈的返回地址信息，从而判断在敏感内存区域上的访问操作是否影响到的函数调用栈的返回地址值。具体来说，如果识别到函数调用操作，应该将当前的敏感内存地址作为键，首先获取函数调用栈的返回地址信息，并利用其得到对应地址内的具体值，再加上栈标识作为一个元组元素存入对象容器中；相应的，识别到函数返回操作则将对应返回地址的容器内的元素删掉。如果识别到敏感区域内存访问操作时，则通过遍历对象的存入栈脆弱性信息元素，获取其中的返回地址，从而读取对应内存地址的值，并将其与元素当中存储的返回值作比较，进而判断固件内的栈溢出脆弱性点是否可以被攻击者利用来劫持程序控制流。

简而言之，如果一个函数调用的地址位于污点数据的传播路径上，同时在由于内存访问操作使得函数返回地址值发生了改变，那么固件中也就存在可以利用的栈溢出脆弱性。

### 3.2.4 堆溢出

在嵌入式固件系统中，程序如果在某一内存地址区域执行了堆分配操作后，如果由固件的引入程序点输入的污点数据流经的路径地址上因发生了内存访问操作而触发了堆溢出，固件中就存在能够触发缓冲区溢出漏洞的堆脆弱性。

具体地，就堆相关的原理而言，当固件内的程序在执行堆缓冲区分配操作时，动态内存分配器（Dynamic Memory Allocator）会将控制数据添加到堆缓冲区的首末区域。与此同时，如果在来自固件外部引入的数据在内部流经的敏感内存路径上执行了内存访问操作时，受控数据被写入了相应被分配的缓冲区中，进而触发了堆缓冲区问题，那么，污点数据就会覆盖原本动态内存分配器在堆缓冲区首末地址处存储的控制数据，也就是堆元数据（Heap Meta Data），此时攻击者便可以利用固件中的这一堆溢出类脆弱性通过输入的受控数据来触发执行任意代码的漏洞。

除此之外，与堆脆弱性相关的可以触发的漏洞还包括双重释放（Double Free）以及释放后访问（Use After Free），这两种类型脆弱性的触发机制以及可利用性的识别将在本节的第3.2.5条中作详细分析。

如图3-1所示，给出了一个可利用堆溢出的实例，内存写操作使得缓冲区buf1发生溢出，并且擦除缓冲区buf2的元数据，从而使得堆溢出类脆弱性可以被触发漏洞[39]。

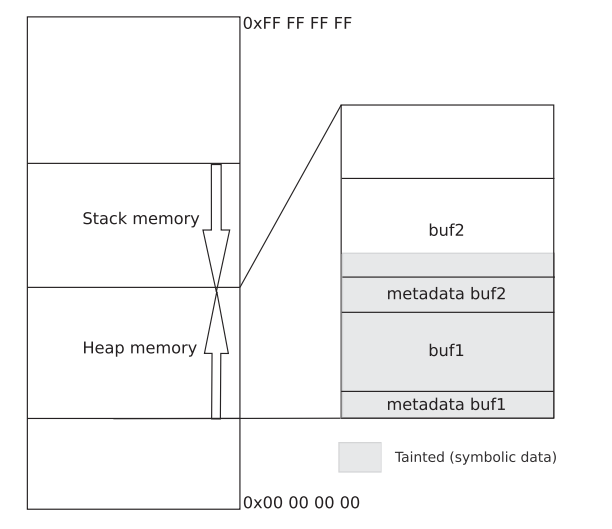


图3-1可利用的堆脆弱性实例

因此，针对以上对固件中堆溢出类脆弱性的漏洞触发机制的简要分析，可以得出，固件中的这一脆弱性点可以被攻击者所利用的几个必要条件如下所述。

（1）在固件的敏感内存上执行了分配堆缓冲区的操作。首先， 源于外部的数据在固件当中的传播路径构成了敏感的内存区域。在这些内存地址中，攻击者可以将其所能够控制的数据引入到固件内部的区域，而如果这些内存地址之中存在脆弱性点，那么在满足特定的漏洞触发条件时便可以使其能够达到预期的攻击目的。与此同时，在敏感区域当中，如果调用了相关的堆缓冲区分配函数（包括malloc、calloc、realloc），那么这些堆就能够被自外部引入的污点数据所影响，所以固件中这一内存地址区域也就存在了堆溢出类脆弱性。

（2）在敏感内存区域中执行了内存访问操作。然后，在固件上的这些敏感内存区域上的内存访问操作（即内存读取与内存写入）可能会涉及到固件当中已经分配的堆缓冲区。当固件中存在堆溢出类脆弱性时，这些内存访问操作可能会使得外部的受控数据写入到这些堆缓冲区中，从而使得固件触发相应的漏洞。

（3）相关敏感操作引入的敏感数据被写入了堆缓冲区，并且发生了溢出。在固件的敏感区域执行了内存访问操作并且分配了堆缓冲区时，如果有数据写入敏感内存地址上的堆缓冲区，同时，写入的数据超出了为堆分配的边界，那么就会触发堆溢出漏洞。

所以，针对以上对堆脆弱性的漏洞触发机制以及必要条件的详细分析，实现对堆溢出类脆弱性可利用性的判别的关键主要包括以下这几个方面。

（1）确定固件的敏感内存地址区域。 首先，通过固件二进制分析技术以及污点分析与符号执行完成对污点数据的追踪，从而确定固件当中由外部引入的污点数据组成的内存地址的路径集合，也就是固件的敏感内存区域。

（2）识别在敏感内存区域上的堆分配操作，记录堆分配函数的相关参数。然后，对于固件当中这些由敏感内存地址组成的集合，通过判断是否包含堆分配函数的内存地址来判断是否分配了堆缓冲区。其中，已知固件程序中堆分配库函数包括malloc、realloc、calloc，从而可以在二进制程序分析框架下确定这些函数的内存地址。在程序流向这些地址之后，需要获取堆脆弱性的信息来为后续的可利用性识别步骤提供判断依据。因此，这里就可以通过二进制分析引擎获取实时的执行状态信息，从中读取堆分配的缓冲区地址以及分配大小，进而附带堆标识一起作为值存入以当前敏感内存地址为键的保存固件脆弱性信息的对象容器中。

（3）识别在敏感区域的内存访问操作，为后续的判断添加断点。进一步地，完成了上述对堆脆弱性信息的获取之后，需要确定相关能够影响固件当中能够影响到已分配的堆缓冲区的元数据从而触发堆溢出的操作。因此，需要通过利用二进制程序分析工具为固件二进制程序的敏感内存路径上添加内存访问操作的断点，从而识别这些敏感操作。同时，需要指定堆脆弱性点的识别函数为回调函数参数，从而完成对固件堆相关漏洞的发现。

（4）判断相关操作是否会触发堆缓冲区溢出漏洞。最后，需要设计固件中存在的堆溢出脆弱性点的可利用性识别模块来完成固件漏洞的发现。具体来说，执行了敏感区域内存访问操作后，程序流将会触发对应的程序断点，并将其引导至漏洞触发判断函数。在漏洞触发判断模块中，通过固件二进制分析技术以及分析引擎获取当前对应的执行状态信息，从中读取到当前内存访问操作的目标堆缓冲区地址、以及堆已经分配的缓冲区长度以及写入堆的数据大小。进而，通过判断这一操作是否会使的对应的堆缓冲区发生溢出来完成对固件中存在的脆弱性点的可利用性的识别。

简而言之，对于受控数据流经的内存地址中分配的堆缓冲区，如果在程序执行了内存访问操作后会触发堆溢出从而覆盖堆的元数据，那么固件中也就存在可利用的堆溢出脆弱性。

### 3.2.5 双重释放和释放后访问

在嵌入式固件系统中，如果在敏感内存区域上调用了堆释放函数或执行了内存访问操作的同时，固件程序中污点数据流经的路径上某一地址分配了堆缓冲，那么固件中就存在释放类脆弱性。进一步地，如果相应的操作是在满足特定条件的情况下执行的，那么这种释放类漏洞也就是可以被攻击者所利用的，固件中也就存在相应的漏洞。

#### 3.2.4.1 释放后访问

在固件中，如果程序访问了已经释放了的缓冲区时，由于创建了悬空指针指向了某一内存地址中已经释放的对象，这时就会出现释放后访问(use after free，即UAF)漏洞[40]。一个能够触发这类漏洞的程序源代码片段实例如下。这些代码若被执行，那么最终指针将指向无效的内存地址区域[34]。

char \* ptr = malloc(SIZE);

…

if(condition){

free(ptr)

}

…

printf("%s",ptr)

根据上文对释放后访问类脆弱性点的漏洞触发机制的原理的简要分析，在固件系统的存在这种是释放类脆弱性点，同时其能够被攻击者利用进而触发漏洞的几个必要条件具体如下。

（1）在敏感的内存地址区域上分配堆缓冲区。首先，在来自固件外部引入的污点数据在固件内部的流向路径上执行了堆分配操作，那么，相应的指针就会指向该内存地址的堆缓冲区。

（2）将上述位于敏感内存地址的堆缓冲区释放。然后，调用堆释放函数将对应的堆缓冲区释放之后，指向该敏感内存地址上的指针依然可用。在这种情况下，固件程序中就存在了能够在特殊情况下触发漏洞的释放后访问脆弱性点。

（3）在敏感区域中执行的内存访问操作访问了已释放的堆缓冲区。最后，如果在污点数据流经的敏感内存路径上执行了内存访问操作，并且此时使用了指针指向了之前释放的堆缓冲区，那么固件程序中就出触发释放后访问漏洞。

对于释放后访问漏洞，可以被攻击者获取固件中的敏感信息（机密信息、内存布局信息），也可以执行攻击者提供的代码。执行任意代漏洞，相应的触发步骤如下。

（1）固件程序中分配后释放了缓冲区buffer1。

（2）攻击者分配了缓冲区buffer2，同时将内存地址指定为之前给buffer2分配的相应内存地址。

（3）攻击者将受控数据写入到内存块buffer2中。

（4）固件程序执行内存访问操作访问了内存块buffer1，进而访问了攻击者在此内存地址中存储的数据。

结合以上对释放后访问脆弱性点漏洞触发机制的原理、对应的必要条件以及具体的攻击实例，对这类脆弱性点的可利用性识别的核心主要包括以下几个方面。

（1）识别固件程序在敏感的内存地址中执行调用的堆分配函数的操作。

（2）识别固件程序中针对在上述敏感内存地址上的对应堆缓冲区的释放操作。

（3）识别固件敏感内存区域中，执行的能够涉及到相应内存地址的内存访问操作。

#### 3.2.4.2 双重释放

双重释放漏洞与堆溢出漏洞的原理类似，当固件中针对某一敏感内存地址上分配的堆执行了两次的释放操作后，堆相关地址会被泄露，攻击者进而可以利用这些泄露的堆地址完成在这一地址上写入受控数据，进而劫持程序的控制流，最终使得固件程序满足执行任意代码漏洞的触发条件[41]。在这种情况下，固件中就存在可以被攻击者利用来触发漏洞的双重释放脆弱性。由以上双重释放类漏洞触发机制，可以得出如下这几个脆弱点得以被利用的必要条件。

（1）固件程序在敏感内存上分配了堆缓冲区（malloc）。

（2）程序对这一缓冲区执行了重复的释放操作（free）。

因此，针对此双重释放脆弱点可利用性识别的关键在于以下方面。

（1）通过获取外部引入的污点数据在固件当中的流向路径来确定固件的敏感区域。

（2）识别在敏感区域上的堆分配和释放操作。

（3）通过在执行相关操作时记录并更新下对应内存地址下的堆的分配/释放状态信息来完成对双重释放漏洞的发现。

## 3.3 固件脆弱点定位方法

综合以上对固件中各种类型的脆弱点的分析，实现对它们所会触发的漏洞的发现的具体方法将在下文中以脆弱点信息收集以及漏洞检测两个方面展开介绍。

### 3.3.1 不安全函数和整型溢出

不安全函数整型溢出脆弱点需要收集获取相关不安全函数的内存地址、当前程序执行路径以及路径约束以及污点分析结果等脆弱性信息。

这类脆弱点的定位方法为：如果当前在污点数据路径上调用了不安全函数，同时函数执行参数超出了路径的边界长度，那么认定固件中存在这类漏洞，与其相关的信息将会被计入到日志中。

因此，对于这些能够触发漏洞的此类脆弱点的定位算法如表3-1。

表3-1不安全函数脆弱点可利用性识别算法

|  |
| --- |
| 算法1：不安全函数脆弱点可利用性识别算法 |
| **输入：**当前执行路径current\_path、路径约束bouded\_info、不安全函数地址sink\_addrs  **输出：**固件漏洞信息（路径、通信数据键、触发地址等）  **Begin**  //获取程序跳转目标是否为出发地址的结果  is\_sink ← jump\_in\_sink(current\_path, sink\_addrs)  //判断是否到达触发地址并且没有严格的约束  **if** ((not is\_sink) or (not is\_any\_taint\_var\_bounded (bouded\_info)) **then**  未触发漏洞，则返回False  **else**  触发漏洞，则将不安全函数相关信息记录至日志  **end if**  **End** |

### 3.3.2 受控循环

受控循环类脆弱点需要获取包括程序执行路径以及对应的执行状态信息、污点分析结果等脆弱性信息。受控循环类脆弱点的定位方法具体是：如果当前在程序在某一敏感内存区域存在循环，同时循环的迭代条件完全依赖于污点数据，那么认定固件中存在这类漏洞，与其相关的信息将会被计入到日志中。

因此，对于这类脆弱点的定位算法如表3-2所示。

表3-2可诱发漏洞的受控循环识别算法

|  |
| --- |
| 算法2：可诱发漏洞的受控循环识别算法 |
| **输入：**当前执行路径current\_path、当前执行状态current\_state、当前地址current\_addr  **输出：**固件受控循环类漏洞信息（路径、触发地址、循环迭代条件等）  **Begin**  **//**获取后续执行路径  next\_path ← current\_path.copy(copy\_states=True).step()  //获取后续执行状态  next\_active ← next\_path.active  //判断程序执行路径是否已经到末端  **if** len(next\_active<=0) then  结束，则返回False  **else**  //通过判断是否不存在重复地址以及后跳地址  **if** ( len(same\_addrs) == 0 or len(back\_jumps) == 0 ) then  //若不存在则返回，此时的二进制程序块不存在循环  return  //存在循环则读取相应执行路径的迭代条件cond\_guard  **else**  **for** each g **in** next\_active[0].guards  cond\_guards.add(g)  cond\_guard ←cond\_guards[-1]  **end if**  //判断当前循环的迭代条件取决于污点数据  **if** (cond\_guard depends on taint data) **then**  迭代条件依赖于污点数据，则将受控循环相关信息记录到日志中  **end if**  **End** |

### 3.3.3 栈溢出

栈溢出脆弱性点需要获取程序执行路径以及对应的执行状态信息、程序调用栈的执行状态信息、污点分析信息。同时，如果在某一内存地址识别到调用操作（Ijk\_Call）将三元组（返回地址，返回地址值，栈标识）作为值存到以内存地址作为键的对象容器中；如果在某一内存地址识别到返回操作（Ijk\_Ret），则删除对象容器中某一个返回地址属性值等于此时函数的返回地址的栈脆弱性信息元素。因此，栈溢出脆弱点执行状态信息的获取算法如表3-3所示。

表3-3栈脆弱点执行状态信息获取算法

|  |
| --- |
| 算法3：栈脆弱点执行状态信息获取算法 |
| **输入：**当前执行路径current\_path、当前执行状态state、当前地址addr  **Begin**  获取当前二进制块block  获取函数调用地址call\_func\_addr  获取函数返回地址ret\_func\_addr  获取函数返回值ret\_func\_val  获取程序跳转类型jumpKind  //判断当前是否为函数调用  **if** (jumpKind == 'Ijk\_Call') **then**  将三元组(ret\_func\_val, ret\_func\_addr,'stack')保存到以当前的内存addr为键的对象容器allocated\_buffers中  **end if**  //判断当前是否为函数返回  **if** (jumpKind == 'Ijk\_Ret') **then**  获取返回地址对应的对象容器索引内存地址index  删除对象allocated\_buffers 中键为index的元素  **end if**  **End** |

如表3-3所示，栈类脆弱点的定位方法为：如果当前执行敏感内存访问操作后，对应内存的值的栈脆弱性信息元素中的返回地址值发生了改变，那么固件中就存在这类漏洞，与其相关的信息将会被计入到日志中。

### 3.3.4 堆溢出

堆溢出脆弱点需要获取程序执行路径以及对应的执行状态信息、污点分析信息。在敏感内存区域上，如果调用了堆分配函数（malloc、realloc、calloc），则将当前的内存地址作为键，以三元组（堆分配大小、堆地址、堆标识）存入对象容器中。因此，堆溢出脆弱点执行状态信息的获取算法如表3-4所示。因为涉及到双重释放，因此算法中也包含对这类漏洞的识别。

表3-4 堆脆弱点执行状态信息获取算法

|  |
| --- |
| 算法4：堆脆弱点执行状态信息获取算法 |
| **输入：**当前执行路径current\_path、当前执行状态state、当前地址current\_addr  **Begin**  获取当前二进制块b  获取堆分配、堆释放函数地址addrs\_alloc、addrs\_free  获取程序后续执行地址next\_addr  //判断程序是否将调用堆分配函数  **if** (next\_addr in addrs\_alloc) **then**  获取对应的保存堆相关信息寄存器  访问寄存器获取分配的缓冲区地址  将三元组(buffer\_address, size, 'heap')保存到以当前的内存current\_addr为键的对象容器allocated\_buffers中  将相应堆地址状态更新为已分配（allocated）  **end if**  **//**如果程序将调用堆释放函数  **if** (next\_addr in addrs\_free) **then**  获取对象容器中堆缓冲区地址与其相同的元素的索引index  删除对象容器中对应于索引index的容器  将相应堆地址状态更新为已释放（freed）  **//**判断程序是否重复释放了同一堆缓冲区  **if** (self.heap\_freed\_state[buffer\_address] ==True) **then**  将双重释放漏洞相关信息记入日志  **end if**  **End** |

如表3-4所示，堆溢出脆弱点的定位方法是：如果当前执行了敏感内存写操作后相应的堆缓冲区发生了溢出，那么固件中就存在这类漏洞，与其相关的信息将会被计入到日志中。

因此，根据以上的具体方法，以在敏感区域上的内存写操作为例，识别可以触发漏洞的脆弱点算法如表3-5所示。

表3-5堆和栈脆弱点定位算法

|  |
| --- |
| 算法5：堆和栈脆弱点定位算法 |
| **输入：**当前执行路径current\_path、当前执行状态state、当前地址current\_addr  **输出：**固件溢出类漏洞信息（溢出内存地址、内存访问操作前后返回值（栈）、内存访问操作地址和长度（堆）、污点传播路径）或释放类漏洞信息  **Begin**  利用state.inspect方法获取当前写地址write\_addr、长度write\_length  通过读取state.callstack获取当前返回地址ret\_func\_addr，  根据ret\_func\_addr得到返回值ret\_val  **//**遍历保存脆弱点信息的容器  **for** each index **in** self.\_allocated\_buffers  item ← self.\_allocated\_buffers[index]  //如果命中了堆脆弱点，则判断是否会改变返回值  **if** (item[2] == 'stack' and item[0] ==ret\_func\_addr and item[1] != ret\_val) **then**  报告栈溢出漏洞，并将相关信息记录至日志  return 'stack overflow'  **end if**  //如果命中了堆脆弱点，则判断是否会发生溢出  **if (**item[2] == 'heap' and item[0] == write\_addr and item[1] < write\_length**) then**  报告堆溢出漏洞，并将相关信息记录至日志  return 'heap overflow'  **end if**  **end for**  //判断是否访问了已经释放了的缓冲区  **if** (self.heap\_freed\_state[buffer\_addr]) **then**  报告释放后访问漏洞，并将相关信息记录至日志  return 'use after free'  **end if**  **End** |

### 3.3.5 双重释放和释放后访问

释放类脆弱点需要获取程序执行路径以及对应的执行状态信息、污点分析等信息。同时，在敏感内存区域上，如果调用了堆分配函数（malloc、realloc、calloc）以及堆释放函数（free），则需要将当前操作后相应堆的状态信息记录或更新。

这类脆弱点的定位方法则为：如果对同一敏感内存地址上的堆缓冲区执行了重复的释放操作，或者执行内存访问操作的对象是已经被释放了的堆缓冲区，那么固件中就存在释放类漏洞，与其相关的信息将会被计入到日志中。具体地，可触发漏洞的双重释放脆弱点定位如表3-4所示，可触发漏洞的释放后访问脆弱点定位如表3-5所示。

## 3.4 本章小结

本章围绕固件的脆弱性进行研究，从脆弱性的可利用性原理开始，详细分析了各个类型的固件脆弱性的漏洞触发机制以及识别的核心，并给出了对应的固件脆弱性识别的具体方法，并完成了各类型固件脆弱点定位算法的设计。总而言之，本章为后续完成固件漏洞检测系统当中脆弱性点定位以及可利用性识别模块的设计与实现提供理论基础。

# 第4章 固件漏洞检测系统的设计与实现

本文第4章将在第2章中面向二进制的固件静态分析技术的基础之上，结合第3章中所列出的固件脆弱性的可利用性识别原理以及具体地识别算法，完成基于脆弱点定位的固件漏洞检测系统的设计与实现。

其中，本章第1节将根据固件漏洞检测系统的开发目标，完成系统的整体设计并细分出将要实现系统相关功能的特定模块，以结构图的形式来描述系统结构与工作流程。本章第2节将介绍本研究课题的主要工作内容，即以二进制块为单位分析固件脆弱性信息，发现脆弱性点并判断其可利用性，具体包括可利用的不安全函数脆弱性的识别、可诱发漏洞的受控循环的识别、栈和堆的实时执行状态信息的获取栈和堆相关脆弱点的可利用性的识别这四个模块。本章第3节将就设计完成的系统模块，通过实际固件测试的方式验证系统的功能是否能够实现。本章最后一节将对本章关于系统的设计与实现的内容进行小结。

## 4.1 系统总体设计

基于脆弱点定位的固件漏洞检测系统模块结构如图4-1所示，相应的，系统的总体工作流程如图4-2所示。



图4-1系统模块结构图



图4-2基于脆弱点定位的固件检测系统流程图

如图4-2所示，系统各个模块的具体工作如下。

系统首先接收到嵌入式设备的固件样本输入后，经过固件拆包模块，利用固件解析工具binwalk完成对固件样本的初步解析，获取固件的二进制程序集。随后，在固件二进制程序依赖图构建模块中，先是通过边界二进制程序检测算法完成对固件中负责直接与外部用户交互并传递数据的二进制程序后，结合污点分析引擎，确定固件内二进制程序的依赖关系，从而构建二进制程序依赖图。进一步地，在已经完成构建的固件二进制程序依赖图的基础之上，通过对固件中的数据流分析，根据进程间的通信方式确定固件中数据流向的共享通信数据键，从而确定固件中二进制程序之间的数据流向关系以及所使用的通信数据键，最终完成二进制程序流图的构建。最后，系统就进入到了本文课题的研究核心，即固件脆弱性点的发现以及脆弱性点可利用性识别模块。在这一模块中，首先将会通过分析来自外部数据在固件中流向路径中相应的二进制程序块，利用面向二进制的固件分析技术收集固件的脆弱性信息，从而发现固件中存在触发漏洞危险的脆弱性点。随后，针对已经发现的某一特定类型的固件脆弱性，根据其漏洞触发的机制以及必要条件，确定其是否可以被攻击者利用，进而完成对其可利用性的识别。

简而言之，固件漏洞检测系统可以分为两个部分，在上半部分利用面向二进制的固件分析技术完成对输入样本的分析之后，下半部分将以固件的敏感区域上的二进制块为单位分析脆弱性信息、发现脆弱性点并判断其可利用性。

## 4.2 系统详细设计与实现

### 4.2.1 识别可利用的不安全函数相关脆弱点

在系统这一模块中，不安全函数以及整型溢出脆弱性发现以及可利用性识别的具体实现流程如图4-3所示。如图4-3所示，系统这一模块的功能主要包括通过如下步骤实现。

（1）检查固件二进制程序。通过对污点数据在固件中的流向分析，确定流经的内存地址以及相应的二进制程序块，从而以其作为基本单位开始对固件的脆弱性进行分析。

（2）收集危险函数脆弱性的具体触发地址。根据已经确定的具体的不安全函数，收集在二进制程序中这些脆弱性点的触发地址，从而利于在后续中对跳转操作的目标进行判断。

（3）检查当前的二进制块是否包含引导至危险函数类脆弱性触发地址的跳转。如果当前程序流所在二进制基本块中是否会跳转到相应的触发地址，即调用这些不安全的函数，从而发现固件中不安全函数函数相关脆弱性。



图4-3不安全函数相关脆弱性发现与可利用性识别的实现原理图

（4）检查当前固件基本块在调用危险函数时是否使用了污点数据以及污点数据的边界。首先，利用污点分析引擎获取当前的程序执行状态信息，判断执行了不安全函数的二进制程序块是否位于敏感区域中涉及到了外部引入的污点数据，同时，利用符号执行技术收集污点数据在传播路径上的约束条件，从而判断是否会触发漏洞。

（5）将可以触发漏洞的固件脆弱性点相关信息记入日志。根据收集到的固件脆弱性信息，如果其使得脆弱性满足了漏洞触发的条件，那么就将固件脆弱性的信息记入到日志中。

### 4.2.2 识别可诱发漏洞的受控循环

在系统中，可利用的循环类脆弱性识别模块对可诱发漏洞的受控循环识别的具体实现流程如图4-4所示。如图4-4所示，系统的受控循环识别模块的功能主要包括通过如下步骤实现。

（1）分析固件二进制程序，从中获取路径信息path、执行状态信息。其中，路径状态信息的属性以及其含义如表4-1所示[23]。通过完成对其的分析过程，可以获取表中的路径信息，从而为后续的受控循环识别步骤提供判断依据。

（2）判断二进制程序当前的执行路径上是否存在循环。在通过分析二进制程序获取相关路径信息外，需要读取执行状态信息中的history.bbladdr属性，其对应着该状态执行的基本块地址的列表。进而，根据这些信息来获取路径中的重复地址和回跳地址的列表，从而发现路径中可能存在的循环。

表4-1路径状态信息的属性说明

|  |  |
| --- | --- |
| 属性名 | 属性含义 |
| Path.addr\_trace | 到目前为止已执行的基本块的地址，以整数表示 |
| Path.trace | 到目前为止已生成的SimSuccessors对象，以字符串形式 |
| Path.targets | 到目前为止已执行的跳转/后继对象的目标 |
| Path.guards | 执行Path.targets中列出的分支所必须满足的保护条件 |
| Path.jumpkinds | 作为VEX结构字符串，获取的每个基本块的跳转类型 |
| Path.events | 符号执行中发生的事件的日志 |
| Path.actions | 对Path.events的过滤，仅包括执行引擎执行的操作。 |

（3）获取控制循环的迭代条件，判断其与外部引入的污点数据是否存在依赖关系。通过读取循环对应路径内存地址的guard属性，因为其是执行跳转/后继对象的目标中相应的分支所必需满足的条件，也就是循环迭代条件。进而，依据此条件，利用污点分析技术来确定循环迭代条件是否依赖于外部输入的污点数据。



图4-4 识别可诱发漏洞的受控循环的实现原理图

（4）将可以触发漏洞的固件脆弱性点相关信息记入日志。根据收集到的固件脆弱性信息，如果其使得脆弱性满足了漏洞触发的条件，那么就将固件脆弱性的信息记入到日志中。

### 4.2.3 获取栈和堆的实时执行状态信息

对于可以触发相关溢出类漏洞的栈和堆脆弱性，系统将通过脆弱性信息获取模块完成对相关脆弱性的实时执行状态信息的收集。

针对栈相关脆弱性的收集，系统模块的具体实现如图4-5所示。由图可知，完成对堆执行状态信息的实现步骤具体如下。

（1）检查当前程序流执行的二进制程序块，获取相关的执行状态信息。具体来说，需要获取到执行状态信息后，读取函数调用栈相关的属性值，其属性以及说明如表4-2所示。



图4-5 栈脆弱性信息获取流程

表4-2函数调用栈状态信息的属性说明

|  |  |
| --- | --- |
| 属性名 | 属性含义 |
| callstack.func\_addr | 当前正在执行的函数的地址 |
| callstack.call\_site\_addr | 调用当前函数的基本块的地址 |
| callstack.stack\_ptr | 从当前函数开始的堆栈指针的值 |
| callstack.ret\_addr | 当前函数返回时将返回的位置 |

（2）识别到函数的相关步骤，确定路径的跳转类型。在这一步骤中，需要获取到当前二进制块中的跳转类型，从而决定是读取相关脆弱性信息并存入到对象容器中，还是将对象容器中对应的栈脆弱性信息元素删除。其中，当前二进制程序基本块的跳转属性jumpkind以及说明如表4-3所示。因此，如果当前为Ijk\_Call或Ijk\_Ret时将执行相关操作。

表4-3二进制程序块跳转的属性说明

|  |  |
| --- | --- |
| 属性名 | 属性含义 |
| Ijk\_Boring | 到一个地址的正常跳转 |
| Ijk\_Call。 | 到一个地址的函数调用 |
| Ijk\_Ret | 返回 |
| Ijk\_Sig | 信号 |
| Ijk\_Sys | 系统调用 |

（3）根据当前二进制程序块的跳转类型执行相关操作。如果当前是函数调用操作，则首先将读取状态信息state内的函数调用栈信息，得到函数返回地址，进而根据该地址利用state.mem.load获取返回值，最后以当前内存地址作为键，将(返回地址，返回值，栈标识)存入对象容器中；如果当前是函数返回，则读取函数的返回地址，将对象中的某一个返回地址属性与值相等的元素删去。

而对于堆脆弱性信息的收集的实现方法如图4-6所示。因此，如下图所示，完成对堆脆弱性信息的收集步骤具体如下。

（1）检查二进制程序块执行状态信息，收集堆相关函数操作的内存地址。其中，堆分配操作函数包括malloc、realloc、calloc，堆释放函数为free。

（2）识别堆相关函数操作。通过当前二进制块的执行状态是否包含到堆相关函数的地址跳转判断是否执行了堆相关操作。

（3）保存堆脆弱性信息。如果当前调用了堆分配函数，则从执行状态信息state中获取堆分配的缓冲区地址以及大小，加上堆标识存入到以当前二进制程序块的内存地址为键的对象容器中。



图4-6堆脆弱性信息获取流程

（4）释放堆缓冲区，同时判断是否会触发双重释放漏洞。识别到堆释放操作free时，通过获取当前执行状态信息获取当前释放的缓冲区地址，首先需要判断释放的目标缓冲区地址是否为已释放过的，若是则将双重释放漏洞相关信息记入日志，否则将删除对象容器中缓冲区地址属性值与该地址相等的堆脆弱性信息元素删除。

### 4.2.4 识别可利用的栈和堆溢出类脆弱点

在以上系统相关模块完成对脆弱性信息的收集的基础之上，可利用的栈和堆的脆弱性识别如图4-7所示。可利用的溢出类脆弱性识别的实现步骤具体如下。



图4-7脆弱性信息识别模块的实现图

（1）识别敏感内存区域上的内存访问（内存读取与内存写入）操作。根据固件的二进制分析技术以及污点分析技术，获取自外部引入污点数据在固件内流经的敏感内存地址区域。同时，为这些区域上的内存访问操作添加程序断点，当执行这些操作时将会调用脆弱性点的可利用性识别的回调函数来发现固件漏洞。

（2）获取内存访问操作的相关属性参数。两种类型的内存访问操作的相关属性以及说明如表4-4所示。

表4-4内存读写相关属性的说明

|  |  |  |
| --- | --- | --- |
| 事件类型 | 属性 | 属性说明 |
| mem\_read | mem\_read\_address | 正在被读取的内存的地址 |
| mem\_read | mem\_read\_length | 读取的内存的长度 |
| mem\_read | mem\_read\_expr | 地址中的表达式 |
| mem\_write | mem\_write\_address | 正在被写入的内存地址 |
| mem\_write | mem\_write\_length | 写入内存的长度 |
| mem\_write | mem\_write\_expr | 写入内存的表达式 |

（3）识别整型溢出漏洞。根据固件的架构以及当前内存访问的长度判断是否会触发，若是则将漏洞的相关信息记入到日志中。

（4）识别栈溢出漏洞。对于在对象容器中存储的栈脆弱性信息元素，根据其返回地址属性值读取当前执行状态下的返回值。如果经过了敏感内存访问操作后，该值发生了变化，则将栈溢出漏洞的相关信息记录到日志中。

（5）识别释放后访问和堆溢出漏洞。对于在对象容器中存储的栈脆弱性信息元素，首先根据当前执行状态下访问的堆缓冲区地址以及状态，判断是否会命中已经释放过得堆缓冲区，若是则将释放后访问漏洞相关信息记入日志。否则，通过当前对敏感内存区域写的长度与对应的堆分配缓冲区大小作比较，若会触发缓冲区溢出漏洞，则将漏洞信息计入日志。

## 4.3 系统测试

### 4.3.1系统验证方案

对本文已实现的基于脆弱点定位的固件漏洞检测系统的验证重点主要在本研究课题的主要创新工作。具体来说，输入固件样本并在相应的模块完成分析之后，验证系统是否在敏感的内存地址发现固件二进制程序中存在的栈和堆的溢出类脆弱性，同时完成它们相关脆弱性信息的收集，然后，准确识别出程序中针对这些脆弱性点的敏感内存操作，并提取出这些操作中与脆弱性的漏洞触发机制相关的执行状态信息，最后在根据具体的脆弱性点的可利用性识别算法来判断固件的脆弱性是否会触发漏洞。可以用于本项实验的几种固件样本示例如表4-5所示，表中包含设备名称、指令架构、设备型号等参数信息。

表4-5用于系统测试的固件样本参数

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 型号 | 指令架构 | 类型 |
| Tenda | AC6V1 | ARMEL | 路由器 |
| Tenda | AC9V1 | ARMEL | 路由器 |
| D-Link | DIR890LA1 | ARMEL | 路由器 |
| D-Link | DWR-118\_V1 | MIPS32 | 路由器 |
| NETGEAR | R8900-V1.0.2.40 | ARMEL | 路由器 |

### 4.3.2 实验结果与分析

以下将就Tenda设备中型号为AC6V1的固件样本来对系统的各个模块的进行详细验证实验，展示系统验证的关键指标，并以此方法测试该品牌其他几种型号的固件样本，最终给出这些固件样本在本系统完成检测工作后所识别到的敏感内存地址区域上的内存访问操作、固件脆弱点以及其中可以触发漏洞的数量。

对于该固件样本，经过预处理后，获取到的部分二进制程序如表4-6所示。

表4-6 固件样本Tenda AC6V1获取的部分二进制程序

|  |  |
| --- | --- |
| 固件程序名 | 大小（B） |
| httpd | 972956 |
| envram | 3500 |
| dosfsck | 45352 |
| dnrd | 78572 |
| dhttpd | 212948 |
| dhcps | 179656 |
| dhcpcd | 96780 |
| chkntfs | 382776 |
| chat | 18040 |
| cfmd | 47144 |

首先，对于栈脆弱性的识别与相关信息的收集，如果当前的二进制程序块是位于敏感内存地址上调用了相应的函数，则系统的固件脆弱点信息收集模块在识别到调用后，将完成对栈溢出类脆弱性信息的收集的相关操作，其中，该操作的部分参数信息如图4-8所示。

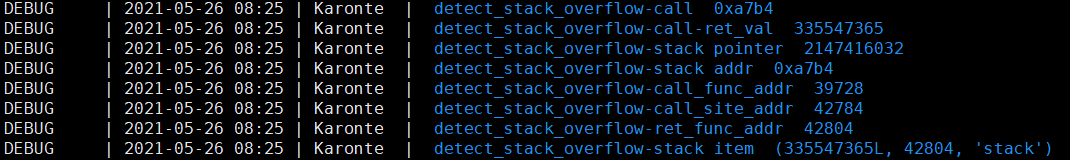


图4-8 栈脆弱性信息获取模块识别到函数调用的操作

如图4-8所示，在内存地址0xa7b4上，识别到函数的调用操作之后，通过当前的执行状态state获取当前函数调用栈实时参数信息callstack后，可以提取到当前调用此函数的基本块地址call\_site\_addr（也就是当前的二进制基本块）、当前调用的函数地址call\_func\_addr以及函数返回地址ret\_func\_addr。进而，根据函数的返回地址，利用执行状态state的内存访问方法state.memory.load(addr)可以获取到返回地址对应的值。最后，将当前的内存地址作为键，将（返回值，返回地址，栈标识）作为值添加到保存固件脆弱性信息的对象容器中。举例来说，对于图4-8所示中的函数调用操作，对应的键是内存地址0xa7b4，值则是元组(335547365L，42804，'stack')。

相应的，如果当前二进制程序执行函数的返回操作，则将会通过读取当前的返回地址，在遍历保存固件脆弱性信息的对象容器后，将其中返回地址属性与当前函数的返回地址相同的元素删去，相应操作如图4-9所示。

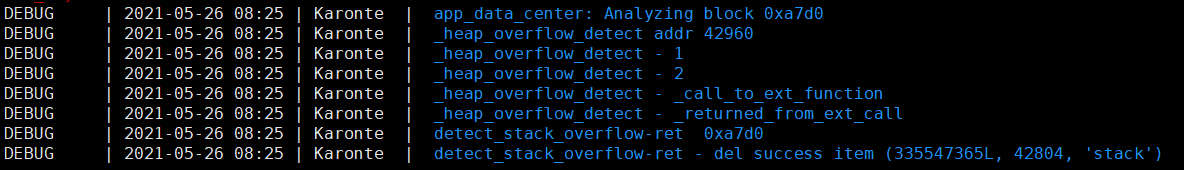


图4-9 栈脆弱性信息获取模块识别到函数返回的操作

随后，对于堆脆弱性信息的收集，如果当前调用了堆释放函数（malloc/realloc/calloc）之后，系统的固件脆弱点信息收集模块获取的相关参数如图4-10所示。

如图4-10所示，当前的二进制程序块在内存地址66488上调用了堆分配函数，随后，通过获取当前的执行状态state，结合当前调用的函数（malloc、realloc、calloc）可以获取到相应的执行参数，包括分配的对缓冲区地址（符号地址）以及分配的大小，最后将当前的内存地址作为键，将元组（堆缓冲区地址，堆分配大小，堆标识）作为值添加到保存固件脆弱性信息的对象容器中，同时将此堆缓冲区地址的状态更新为已分配。

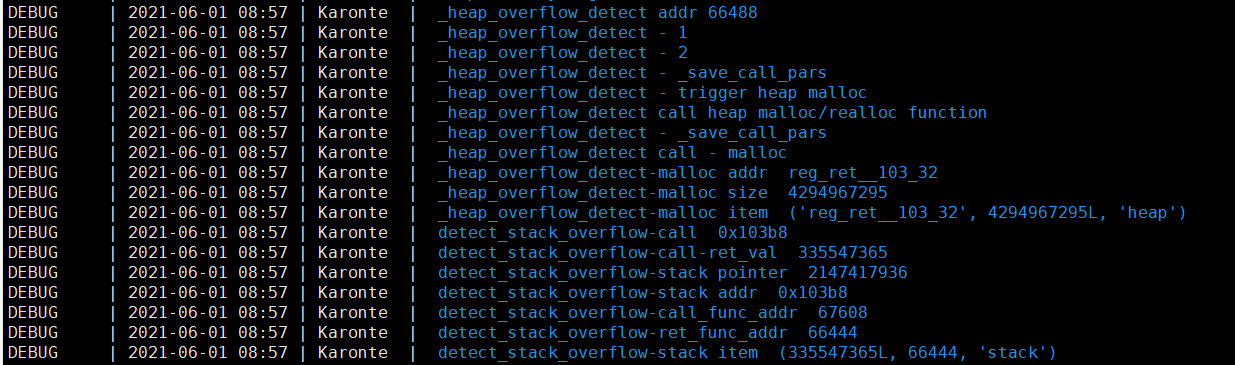


图4-10堆脆弱性信息获取模块识别到堆分配的操作

对应的，如果识别到堆分配函数（free）之后，模块将会将对应于堆释放的缓冲区地址的对象容器中的堆脆弱性信息元素删除，同时将此堆缓冲区地址的状态更新为已释放。如图4-11所示，二进制程序块识别到堆释放函数。

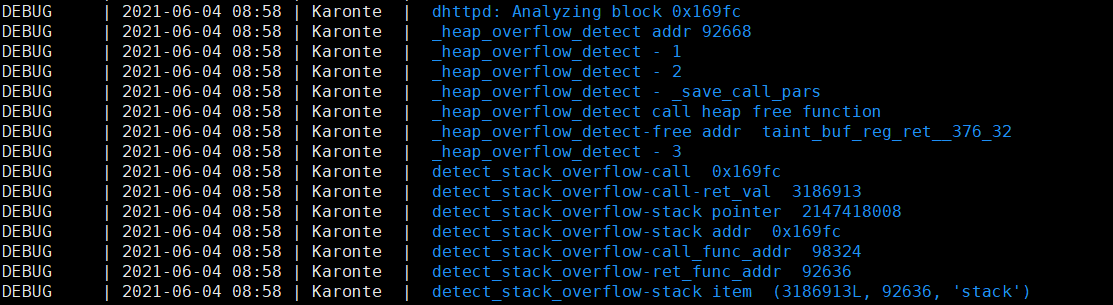


图4-11堆脆弱性点信息获取模块识别到堆释放操作

在系统的脆弱点可利用性识别模块中，如果在污点传播路径的内存地址上如果识别到了到敏感内存访问操作，则会命中程序中设置的断点，同时执行回调函数。在某一内存地址上触发了内存读取操作断点的示例如图4-12所示。

如图4-12所示，通过提取当前执行引擎触发的内存写事件中的相关属性，可以获取到当前内存写入的地址、长度、表达式。此外，由当前的函数调用栈中可以获取当前的函数返回地址等。

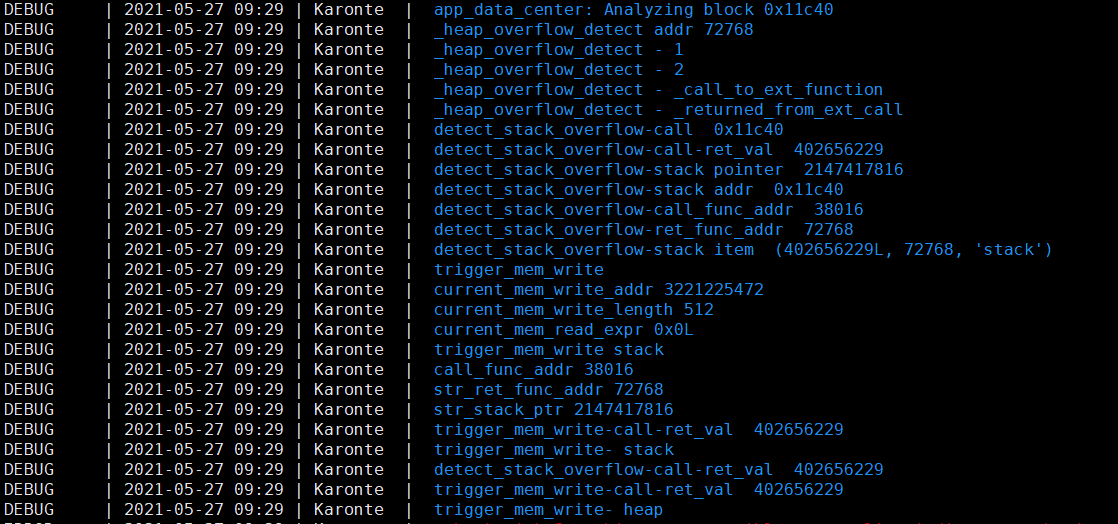


图4-12 脆弱性点可利用性识别模块检测到敏感内存写操作

此外，在当前程序的敏感区域中上，对于上述的识别到内存访问操作的二进制块的反编译结果如图4-13所示。

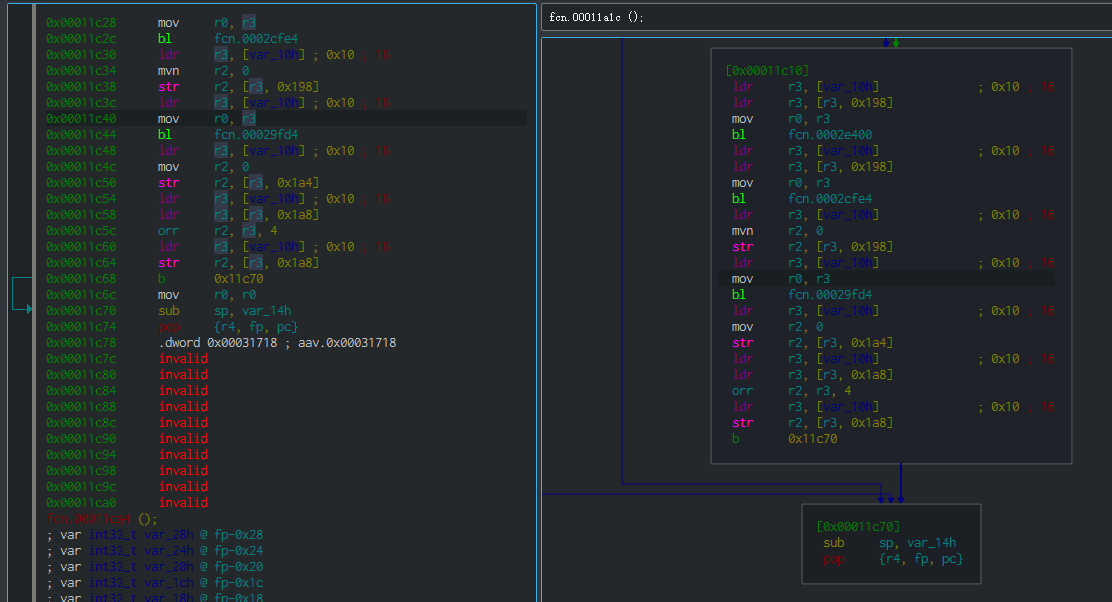


图4-13发生敏感操作的程序二进制块反编译示例

进一步地，完成以上这些执行引擎断点状态信息的获取之后，将进入到脆弱性点的可利用性识别步骤中。具体来说，对于栈溢出类脆弱点，需要确定在敏感内存访问操作之后，根据函数返回地址，利用内存地址读取方法获取函数返回值，随后，通过遍历保存脆弱性信息的容器，读取其中带有栈标识以及返回地址为当前函数返回地址的元素，进而比较两个返回值在经过敏感区域上的内存访问操作后是否发生改变，从而发现栈溢出漏洞。对栈溢出脆弱的一个识别实例如图4-14所示。

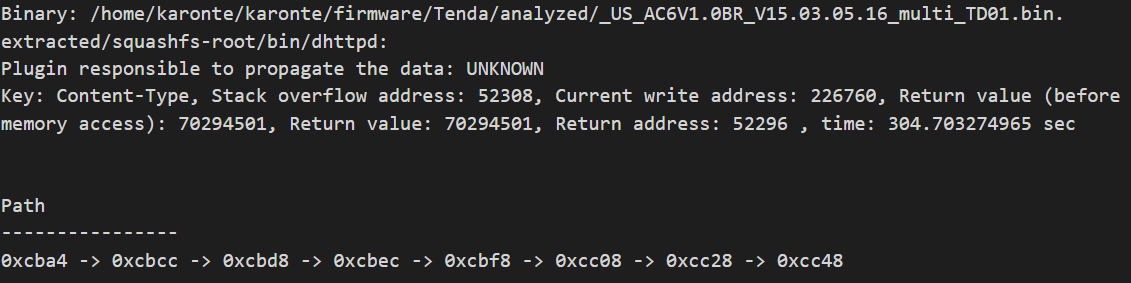


图4-14栈溢出脆弱点识别示例

如图4-14所示，通过追踪数据键content-type在固件二进制程序中的传播路径并识别到在相关脆弱点中发生了内存访问操作后，日志将会记录下对应的脆弱性信息，包括漏洞触发的关键判断信息，从而检测在固件中可以被攻击者利用而劫持程序控制流，或是读取机密与内存布局信息的固件漏洞。

同理，利用该系统识别到可利用的堆脆弱点，会将当前的二进制程序块地址、堆溢出缓冲区地址、当前敏感内存操作读写大小以及污点传播路径等信息记入到日志中。

而系统已实现的对不安全函数以及受控循环漏洞的识别，在日志中需要将其对应的触发地址以及循环迭代条件关键信息记录下来，对应的识别实例如图4-15和图4-16所示。对于不安全函数，因为对应的块会包含漏洞触发函数的调用且缺少约束等，所以其可以被触发漏洞。而对于受控函数，因为其循环迭代条件可以被攻击者所控制，所以可被触发拒绝服务等漏洞。

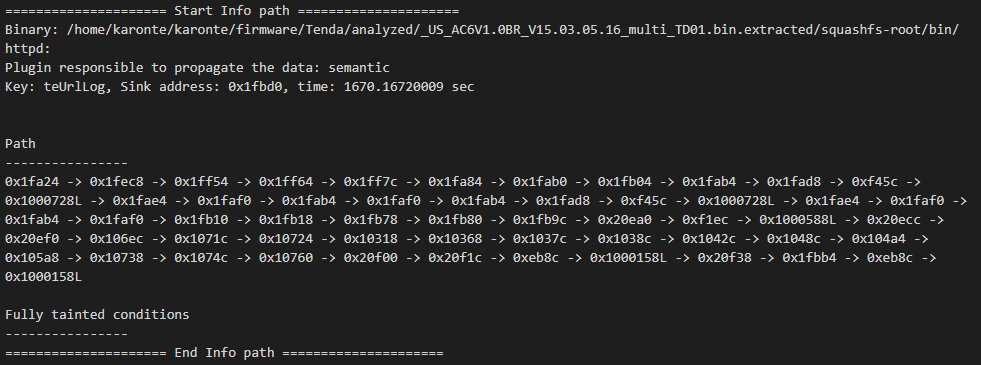


图4-15不安全函数漏洞识别实例

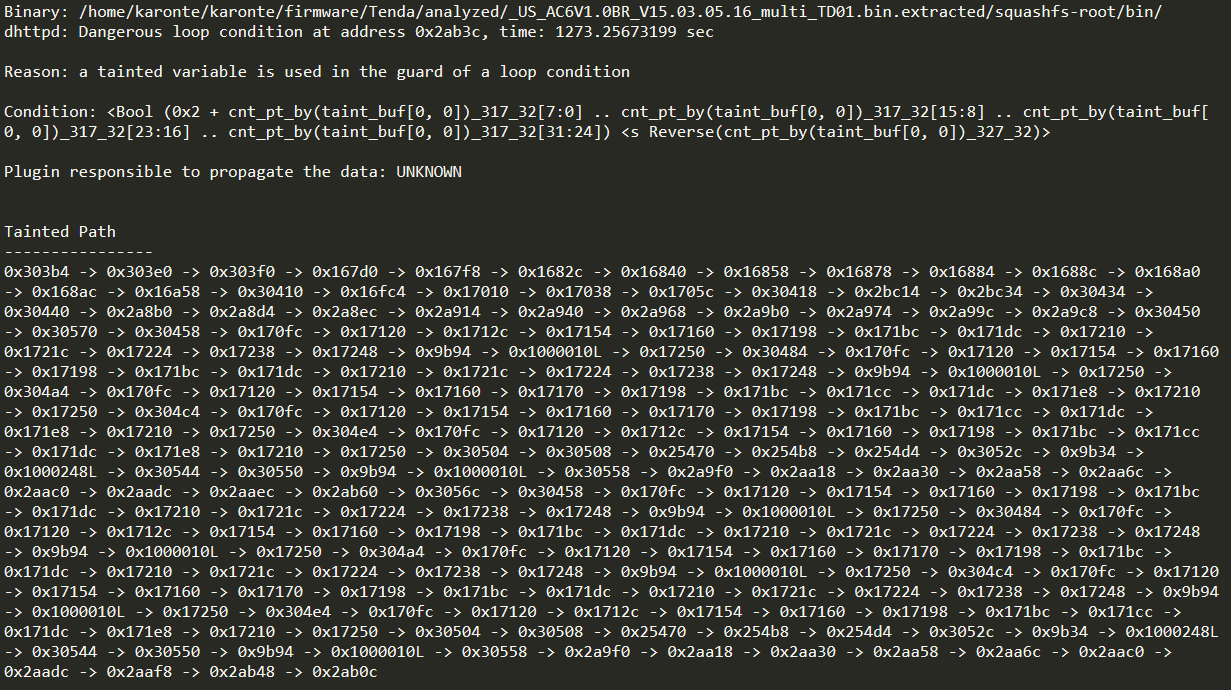


图4-16受控循环漏洞识别实例

因此，结合以上用Tenda AC6V1固件样本完成对系统各个模块的验证步骤，系统发现了1个可以触发漏洞的受控循环类脆弱点，如图4-16所示，同时总共发现了3个可以触发漏洞的不安全函数类脆弱点，如图4-17所示。

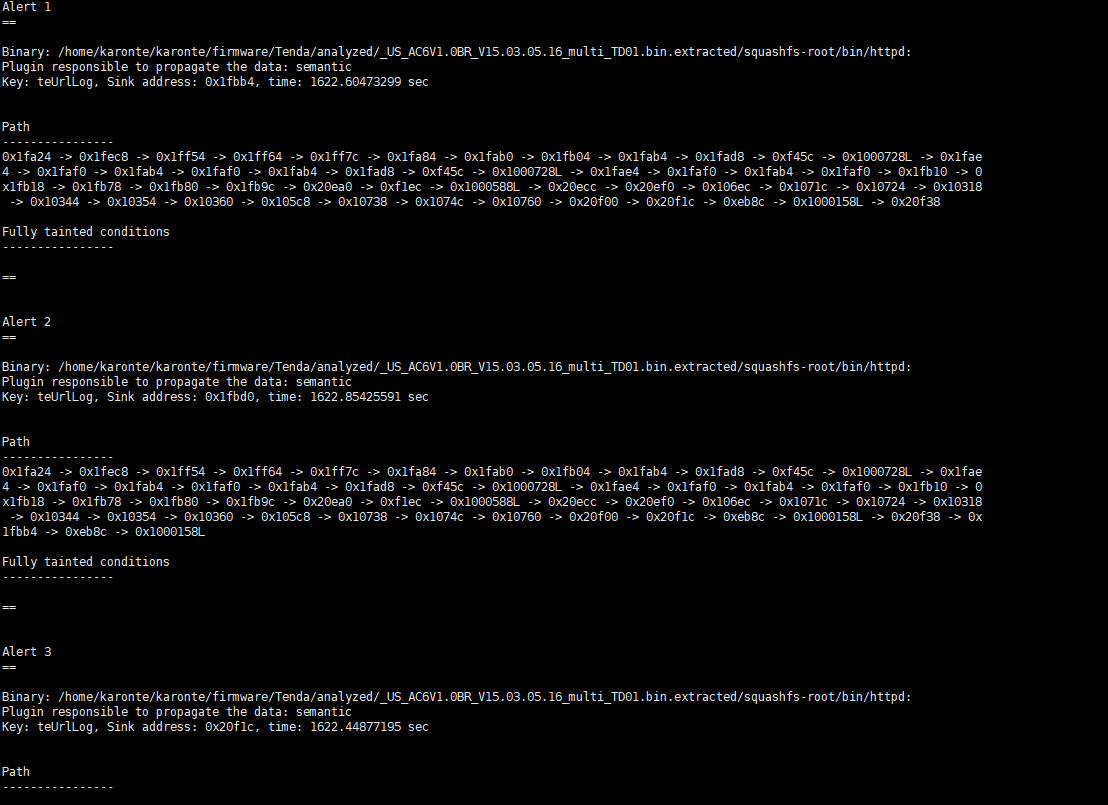


图4-17所有可以触发漏洞的不安全函数脆弱点

通过分析系统日志，系统触发了104次程序断点。也就是说，系统识别到了104次在敏感区域上的内存访问操作。如图4-18所示，系统发现这些敏感内存访问操作命中了4个栈溢出类脆弱点。

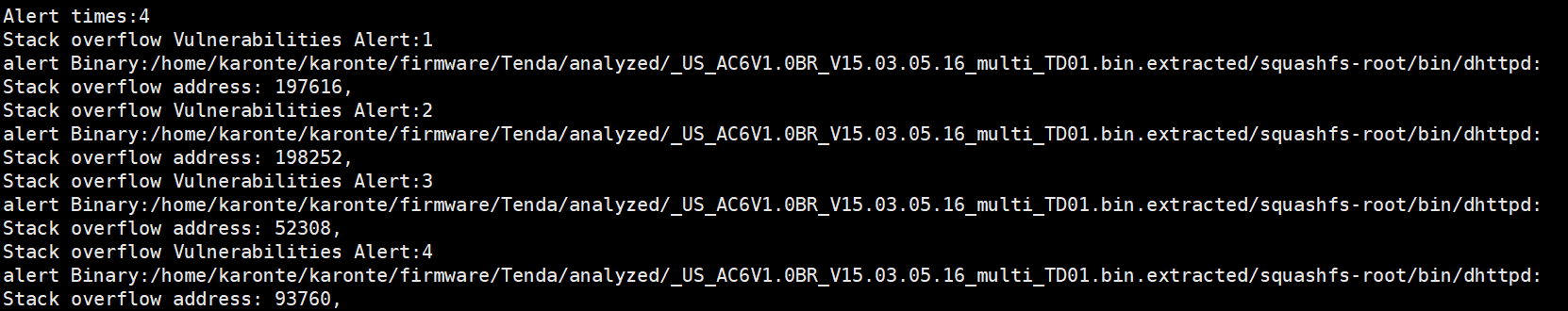


图4-18敏感内存操作命中的固件脆弱性点

系统日志中记录了类似图4-19的脆弱性信息，包括内存块地址、当前内存写地址、内存访问前后的函数返回值等。通过分析如图4-19所示的系统日志，根据其脆弱性信息，发现在内存地址52308上的栈脆弱点不可被触发漏洞。

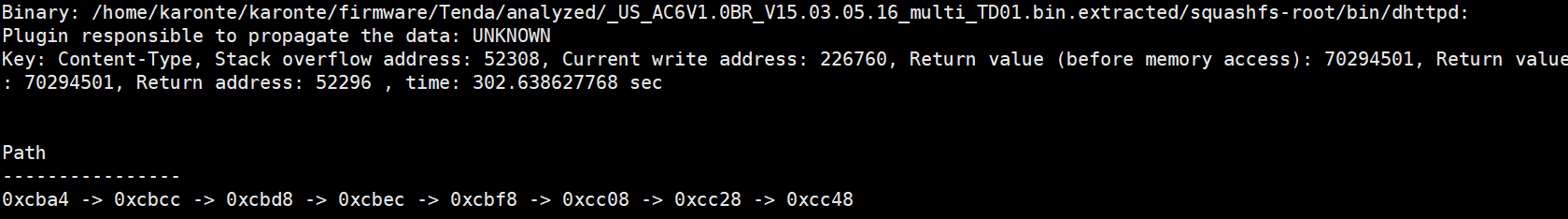


图4-19 系统日志中记录的栈脆弱点信息实例

因此，结合以上的验证实验的各个步骤，系统对固件样本Tenda AC6V1的整体分析结果如表4-7所示。

表4-7 Tenda AC6V1固件样本实验结果

|  |  |  |
| --- | --- | --- |
| 验证系统功能 | 实验指标 | 实验结果 |
| 敏感内存访问操作识别 | 在敏感区域上执行的内存访问操作数量 | 检测到104次敏感内存操作 |
| 脆弱点信息收集 | 检测到的固件脆弱点数 | 在敏感内存区域上发现12个不安全函数类脆弱点、1个受控循环类脆弱点、4个栈溢出类脆弱点 |
| 漏洞发现 | 判断脆弱点是否可以被触发漏洞 | 3个不安全函数类漏洞、1个受控循环类漏洞 |

进一步地，利用本系统完成对Tenda的其他型号的固件样本的实验后，得到如表4-8所示的结果。其中，实验结果的关键指标包括识别到在敏感内存地址区域上的内存访问操作、固件脆弱点以及其中可以触发漏洞的数量。

表4-8 Tenda品牌固件样本实验结果

|  |  |  |  |
| --- | --- | --- | --- |
| 型号 | 敏感内存访问操作 | 脆弱点 | 可触发漏洞的脆弱点 |
| AC6V1 | 104 | 17 | 4 |
| AC9V1 | 489 | 46 | 5 |
| AV15V1 | 301 | 30 | 4 |
| AV18V1 | 811 | 30 | 4 |

## 4.4 本章小结

本章聚焦于本课题的研究核心，即基于脆弱点定位的固件漏洞检测系统的设计与实现。具体来说，在系统总体设计小节中介绍系统的各个模块的工作流程；在系统详细设计中则以模块实现原理图展开对各个模块具体实现的介绍；最后，在系统测试小节中完成了对系统各个模块的功能验证，即系统能够以脆弱点定位的方式发现固件中的漏洞。

# 结 论

本文将物联网设备中的嵌入式固件作为研究核心，通过对国内外在固件漏洞检测领域当中相关的研究现状进行分析后，将基于二进制的脆弱性分析完成基于脆弱点定位的固件漏洞检测作为课题的研究方向，完成固件漏洞检测系统的设计与实现，并在固件样本中对系统的各个模块进行了有效性验证。

本文已完成的工作包括以下两个方面。

（1）完成了可利用的固件脆弱点的分析与定位。首先，对固件脆弱点分析工作从固件中可利用的脆弱点进行概述，分析固件中的脆弱点能够被外部攻击者所利用的条件和识别方法，并介绍可利用的固件脆弱点的影响。进而，针对固件中特定类型的脆弱点，根据其漏洞的触发机制以及条件来提出具体的可利用性识别方法，从而发现可利用的固件脆弱点。进一步地，在这些相关原理的基础之上，完成对固件脆弱点定位算法的设计。

（2）完成了基于脆弱点定位的固件漏洞的检测系统的设计与实现。系统首先将以面向二进制的固件分析技术作为基础，完成对固件中二进制程序的依赖关系和数据流的分析。进而，以固件二进制块为单位分析脆弱性信息，发现脆弱点并判断其可利用性。系统通过在固件样本上完成了对各个模块功能的验证，即可以在二进制分析的基础上，根据第一项工作中设计的算法，完成对固件相关脆弱性信息的收集并发现固件中的脆弱点，最后通过识别敏感内存区域操作是否会满足脆弱点的漏洞触发条件来实现固件漏洞的检测。在实验部分，通过对固件样本Tenda AC6V1详细展示了系统各个模块的功能，并在该品牌的其他型号的固件下完成了系统的测试，发现了这些固件样本当中的敏感内存访问操作、固件所有的脆弱点以及可以触发漏洞的固件脆弱点。

就本文的工作来说，需要进一步深入研究主要是在对系统在大规模固件数据集的适用性以及漏洞检测效率两个方面。在对本课题中设计的系统进行验证时，方法是通过在目前研究者所常用的漏洞检测固件数据集中证明了系统可以在其中收集脆弱点信息，并以此对脆弱点是否满足漏洞触发条件进行判断从而发现固件漏洞。在这一基础之上，可以考虑的更深入的研究方向是通过更大规模的固件样本数据集，在从实验的过程中收集相关漏洞信息来确定更为有效的固件脆弱点识别方法，从而提高固件漏洞发现的效率，并使得系统具有更广的适用范围。

# 参考文献

[1] Miao Y , Jianwei Z , Ming C ,etc.A Survey of Security Vulnerability Analysis, Discovery, Detection, and Mitigation on IoT Devices[J].Future Internet,2020,12(2):27.

[2] Wei X , Jiang Y , Yong T , et al. Vulnerability Detection in IoT Firmware: A Survey. 2017.

[3] 蒋轶焜.大规模自动化固件安全分析技术研究[D].国防科学技术大学,2018.

[4] 郑尧文, 文辉, 程凯,等. 物联网设备漏洞挖掘技术研究综述[J]. 信息安全学报, 2019, 4(5).

[5] Davidson D , Moench B , Jha S , et al. FIE on Firmware: Finding Vulnerabilities in Embedded Systems using Symbolic Execution[C]// Proceedings of the 22nd USENIX conference on Security. USENIX Association, 2013.

[6] Yamaguchi F , Golde N , Arp D , et al. [IEEE 2014 IEEE Symposium on Security and Privacy (SP) - San Jose, CA (2014.5.18-2014.5.21)] 2014 IEEE Symposium on Security and Privacy - Modeling and Discovering Vulnerabilities with Code Property Graphs[J]. 2014:590-604.

[7] Eschweiler S , Yakdan K , Gerhards-Padilla E . discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code[C]// The Network and Distributed System Security Symposium (NDSS 2016). 2016.

[8] Chen D D , Egele M , Woo M , et al. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware[C]// Network and Distributed System Security Symposium. 2016.

[9] David Y , Partush N , Yahav E . FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware[J]. ACM SIGPLAN Notices, 2018, 53(2):392-404.

[10] Wang Y , Shen J , Lin J , et al. Staged Method of Code Similarity Analysis for Firmware Vulnerability Detection[J].IEEE Access, 2019.

[11] 李登,尹青,林键,等.基于同源性分析的嵌入式设备固件漏洞检测[J].计算机工程,2016(4):1-7.

[12] 戴忠华, 费永康, 赵波,等. 基于污点跟踪的固件漏洞定位研究[J]. 山东大学学报(理学版), 2016, 51(9):41-46.

[13] 李朝君.二进制代码安全性分析[D].中国科学技术大学.

[14] Ran L , Lu L , Lin H , et al. An Experimental Study of Four Methods for Homology Analysis of Firmware Vulnerability[C]// 2017 International Conference on Dependable Systems and Their Applications (DSA). 2017.

[15] Tue L, Tuan Ng, Trung L, et al.Maximal Divergence Sequential Autoencoder for Binary Software Vulnerability Detection[C]//ICLR 2019

[16] Shoshitaishvili, Y. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. [C] // NDSS 2015.

[17] Costin A . Security of CCTV and Video Surveillance Systems: Threats, Vulnerabilities, Attacks, and Mitigations[C]// 6th International Workshop on Trustworthy Embedded Devices (TrustED 2016). ACM, 2016.

[18] Michel, Sebastian, P. Triantafillou, and G. Weikum. "KLEE: a framework for distributed top-k query algorithms."[C]// International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September DBLP, 2005.

[19] Patton M , Gross E , Chinn R , et al. Uninvited Connections: A Study of Vulnerable Devices on the Internet of Things (IoT)[C]// IEEE Joint Intelligence & Security Informatics Conference. IEEE, 2014.

[20] 朱怀东, 蒋烈辉, 董卫宇. 基于内存模糊测试的嵌入式固件漏洞检测[J]. 计算机工程与设计, 2018, 39(09):300-304.

[21] Redini N , Machiry A , et al. Karonte: Detecting Insecure Multi-binary Interactions in Embedded Firmware[C]// 2020 IEEE Symposium on Security and Privacy (SP). IEEE, 2020.

[22] Wang T , Tao W , Lin Z , et al. IntScope: Automatically Detecting Integer Overflow Vulnerability In X86 Binary Using Symbolic Execution[C]// Network & Distributed System Security Symposium. 2009.

[23] Yan, S. , Wang, R. , Salls, C. , Stephens, N. , & Vigna, G. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis[C]// Security & Privacy. IEEE, 2016.

[24] Redini N , Machiry A , Das D , et al. BootStomp: On the Security of Bootloaders in Mobile Devices[C]// Usenix Security Symposium 2017. 2021.

[25] 陈千. 基于静态污点分析的固件脆弱性分析技术研究与实现[D]. 中国科学院大学.

[26] Heffner C, “binwalk - firmware analysis tool designed to assist in the analysis, extraction, and reverse engineering of firmware images,” https://github:com /ReFirmLabs/ binwalk, 2014.

[27] Cojocar L , Zaddach J , Verdult R , et al. PIE: Parser Identification in Embedded Systems[C]// the 31st Annual Computer Security Applications Conference. ACM, 2015.

[28] Ester M . A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise[J]. Proc.int.conf.knowledg Discovery & Data Mining, 1996.

[29] Cristian, Cadar, Koushik, et al. Symbolic execution for software testing: three decades later[J]. Communications of the ACM, 2013, 56(2):82-90.

[30] Apt, K. R . The essence of constraint propagation[J]. THEORETICAL COMPUTER SCIENCE -AMSTERDAM-, 1999.

[31] Gyung Kang M, McCamant S., Poosankam P., and Song D. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation.[C]// NDSS 2011.

[32] 吴世忠，郭涛，董国伟. 软件漏洞分析技术[M]. 科学出版社，2014.

[33] Schwartz E J , Avgerinos T , Brumley D . All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)[C] // 31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA. IEEE, 2010

[34] 韩心慧, 魏爽, 叶佳奕,等. 二进制程序中的use-after-free漏洞检测技术[J]. 清华大学学报(自然科学版), 2017, 057(010):1022-1029.

[35] Zheng Y, Cheng K, Li Z, et al. A Lightweight Method for Accelerating Discovery of TaintStyle Vulnerabilities in Embedded Systems[C] // International Conference on Information and Communications Security. Springer, Cham, 2016: 27-36.

[36] Chen, J. , Diao, W. , Zhao, Q. , Zuo, C. , & Zhang, K. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. [C] // NDSS 2018.

[37] Corina J, Machiry A, Salls C, Shoshitaishvili Y, Hao S., C. Kruegel, and G. Vigna, DIFUZE: Interface Aware Fuzzing for Kernel Drivers[C] // ACM SIGSAC Conference on Computer and Communications Security (CCS), 2017.

[38] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. VUzzer: Application-aware Evolutionary Fuzzing[C] // NDSS. 2017.

[39] Boudjema E H , Verlan S , Mokdad L , et al. VYPER: Vulnerability detection in binary code[J]. Security and Privacy, 2020.

[40] Cédric B., Florent S., Dimitri K.. An introduction of Use-After-Free detection in binary code by static analysis. https://blog.amossys.fr /intro-to-use-after-free-detection.html

[41] Caballero J , Grieco G , Marron M , et al. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. ACM, 2012.

# 哈尔滨工业大学学位论文原创性声明和使用权限

学位论文原创性声明

本人郑重声明：此处所提交的学位论文《基于脆弱点定位的固件漏洞检测技术研究》，是本人在导师指导下，在哈尔滨工业大学攻读学位期间独立进行研究工作所取得的成果，且学位论文中除已标注引用文献的部分外不包含他人完成或已发表的研究成果。对本学位论文的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明。

作者签名： 日期： 年 月 日

学位论文使用权限

学位论文是研究生在哈尔滨工业大学攻读学位期间完成的成果，知识产权归属哈尔滨工业大学。学位论文的使用权限如下：

（1）学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文，并向国家图书馆报送学位论文；（2）学校可以将学位论文部分或全部内容编入有关数据库进行检索和提供相应阅览服务；（3）研究生毕业后发表与此学位论文研究成果相关的学术论文和其他成果时，应征得导师同意，且第一署名单位为哈尔滨工业大学。

保密论文在保密期内遵守有关保密规定，解密后适用于此使用权限规定。

本人知悉学位论文的使用权限，并将遵守有关规定。

作者签名： 日期： 年 月 日

导师签名： 日期： 年 月 日

# 致 谢

两年的硕士研究生阶段的学习已经进入了尾声，这也是我本身的求学生活的一个阶段性终点。回首过去，有太多的回忆难以忘怀。

首先，衷心感谢导师王莘副教授对本人的指导，他在研究生阶段中对我的指导将是求学生涯中收获到的宝贵的精神财富。也要感谢实验室全体老师、师兄师姐以及同学们在研究生阶段中给予我的热情帮助和支持。

同时，也要感谢家人和朋友们在生活上对我的支持，这是我的学习生涯得以走到今天的重要保障。

最后，展望未来，研究生阶段的结束并不代表我个人的学习生涯就完全结束了。在之后即将步入的社会生活中，我将继续铭记求学生涯中收获的精神财富，不断提升自己，争取使自己迈上更高的台阶。