

一种基于安全状态跟踪检查的漏洞静态检测方法

梁 彬 侯看看 石文昌 梁朝晖

(中国人民大学信息学院 北京 100872)

(数据工程与知识工程教育部重点试验室 北京 100872)

摘 要 现有的采用基于源代码分析的漏洞静态检测方法中存在的主要问题是误报率和漏报率较高. 主要原因之一是缺乏对数据合法性检查与非可信数据源等程序安全相关元素的精确有效的识别分析. 文中提出了一种基于数据安全状态跟踪和检查的安全漏洞静态检测方法. 该方法对漏洞状态机模型的状态空间进行了扩展, 使用对应多个安全相关属性的向量标识变量安全状态, 细化了状态转换的粒度以提供更为精确的程序安全行为识别; 在漏洞状态机中引入了对合法性检查的识别, 有效降低了误报的发生; 建立了系统化的非可信数据鉴别方法, 可防止由于遗漏非可信数据源而产生的漏报. 基于此方法的原型系统的检测实验表明: 文中方法能够有效检测出软件系统中存在的缓冲区溢出等安全漏洞, 误报率明显降低, 并能避免现有主流静态检测方法中存在的一些严重漏报.

关键词 漏洞检测; 静态分析; 状态机; 漏报; 误报

中图法分类号 TP309

DOI号: 10.3724/SP.J.1016.2009.00899

A Static Vulnerabilities Detection Method Based on Security State Tracing and Checking

LIANG Bin HOU Kan-Kan SHI Wen-Chang LIANG Zhao-Hui

(School of Information, Renmin University of China, Beijing 100872)

(MOE Key Laboratory of Data Engineering and Knowledge Engineering, Beijing 100872)

Abstract The main problem of existing static vulnerability detection methods based source code analysis is their high false positive and false negative rates. One main reason is lack of accurate and effective identification and analysis of security-related program elements, e. g. data validation checking, tainted data source, etc. A static vulnerability detection method based on data security state tracing and checking is proposed. In this method, the state space of state machine model is extended; the security state of a variable is identified by a vector that may correspond to multiple security-related properties rather than by a single property; Fine-grained state transition is provided to support accurate recognition of program security-related behaviors; The recognition of validation checking is introduced in vulnerability state machine to reduce false positives; and a systematic discrimination mechanism for tainted data is constructed to prevent false negatives result from neglecting tainted data sources. The experimental results of a prototype system show that this method can effectively detect buffer overflow and other type's vulnerabilities in software systems, and with obviously lower false positive than existing mainstream static detection methods and avoid some serious false negatives of these methods.

Keywords vulnerabilities detection; static analysis; state machine; false negative; false positive

收稿日期: 2007-11-06; 最终修改稿收到日期: 2009-01-03. 本课题得到国家自然科学基金(60703102, 60873213)、北京市自然科学基金(4082018)和国家“八六三”高技术研究发展计划项目基金(2007AA01Z414)资助. 梁 彬, 男, 1973 年生, 博士, 副教授, 主要研究方向为信息安全与系统软件. E-mail: liangb@ruc.edu.cn. 侯看看, 女, 1986 年生, 硕士研究生, 主要研究方向为静态分析. 石文昌, 男, 1964 年生, 博士, 教授, 博士生导师, 主要研究领域为信息安全、可信计算与系统软件. 梁朝晖, 女, 1968 年生, 博士, 讲师, 主要研究方向为信息安全、网络通信.

1 引 言

大多数信息安全事件的根源在于计算机软件系统中存在有安全漏洞(vulnerability)^[1]. 要杜绝这类安全事件,最根本的办法在于在软件发布前检测出安全漏洞并修正. 为了提高漏洞检测的效率,安全界对自动化的漏洞检测技术进行了研究,主要有动态分析(dynamic analysis)和静态分析(static analysis)两类. 动态检测工具使用较容易^[2],且确实能够发现一些安全漏洞,但测试用例对程序逻辑的覆盖率问题限制了动态检测工具的发掘能力,难于发现隐藏较深的安全漏洞. 静态漏洞分析技术主要通过系统代码进行程序分析(program analysis)来发现各种安全漏洞^[3],例如缓冲区溢出、非法指针引用、竞争条件等等. 与动态分析相比,静态分析技术能够支持更为有效的深度漏洞检测.

漏报(false negative)和误报(false positive)是各种安全漏洞检测技术所要共同面对的问题. 近年来,漏洞静态检测技术已经取得了很大的进展,研究人员已经提出和实现了一些漏洞静态检测方法和工具,目前已在操作系统等复杂系统的漏洞检测中得以应用^[4],但仍然存在误报率和漏报率较高的问题,影响着静态漏洞检测方法进一步推广应用. 这与检测模型和安全相关程序特征的识别有着密切的关系.

首先,现有静态检测方法采用单一的性质标识对程序数据安全性质进行描述,这种一元的描述机制难以进行细粒度的数据状态分析和精确跟踪分析多种平行发展的数据安全相关性质,不能很好地全面覆盖安全漏洞的激发条件. 直接导致的一个严重问题是缺乏对非可信数据合法性检查的有效识别,从而产生大量的误报. 目前的检测方法仅仅集中于跟踪分析数据的可信性,一元的安全性标识往往被完全用于描述数据的可信性,无法精确跟踪对非可信数据所实施的合法性检查,混淆了未经合法性检查的和已经过检查的非可信数据. 现实中,几乎所有的软件系统都要处理各种各样的非可信数据,而大部分非可信数据在引用前都经过了相应的合法性检查,消除了其导致危险操作的可能性. 因此,这种混淆的后果会将大量安全的非可信数据操作识别为安全漏洞,将导致大量的安全程序片段被错误地检测为含有安全漏洞,从而产生大量的误报. 例如,根据文献[5]数据,Johnson 等人使用 CQUAL 系统对

Linux 内核代码进行的用户态/内核态指针引用漏洞检测,由于缺乏对合法性检查的有效识别,误报率在 95%以上. 如此高的误报率需要投入大量的资源进行后期人工分析,大大增加了漏洞静态检测技术推广应用的难度.

其次,虽然个别的漏洞静态检测方法实现了部分的合法性检查识别,但对合法性检查是否有效缺乏必要的深入分析,未能区分出实现上存在缺陷的无效的合法性检查,从而产生漏报. 以斯坦福大学 Engler 等人研发的静态漏洞检测系统 MC 所采用的方法为例^[6],MC 中用于检测内存操作长度越界漏洞的 Range Checker 中引入了对非可信数据的上/下界检查的识别,由于模型表示空间的限制,其仅仅检测长度参数是否经过了相应的算术比较,而并不关心参与比较的上/下界值的类型. 而入侵者能利用一个负值的整数绕开与常量或有符号上界值进行的上界比较,触发安全漏洞. 在现实世界中,最近几年此类安全漏洞已经大量出现在各种平台系统中,例如 Linux 内核中的蓝牙设备驱动有符号缓冲区索引安全漏洞^①. 由于缺乏必要的检查有效性识别,MC 未能检测出此漏洞.

此外,现有静态检测方法对于数据可信性的鉴别也缺乏清晰的概念模型,在实际应用中容易遗漏一些较隐蔽的非可信数据源,导致对部分非可信数据跟踪分析的缺失而产生漏报. 以 MC 所采用的方法为例,在其对 Linux 内核的漏洞检测中,遗漏了内核对用户文件的装载解析这一非可信数据源,导致了对可执行文件解析所引发的安全漏洞的误报.

针对以上问题,本文提出了一种对程序数据安全状态进行跟踪检查的漏洞静态检测方法. 在此方法中,使用有限状态机模型描述程序数据安全性质的变化,对描述数据安全状态的状态空间进行了扩展,采用了多元的状态描述机制,使用一个对应多个安全相关属性的布尔向量标识变量安全状态,更为清晰地反映了变量安全性质的变化,细化了状态转换的粒度以提供更为精确的程序特征识别. 在此基础上,利用多元的安全状态设置,在跟踪分析数据可信性的同时引入了对合法性检查及其有效性的识别,有效降低了由此引起的误报和漏报的发生. 此外,在本文方法中还引入了可信边界与可信边界入口的概念,以可信边界的界定为线索建立了

① Linux Kernel Bluetooth Signed Buffer Index Vulnerability.
<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-0750>

系统化的非可信数据鉴别方法,防止了由于遗漏非可信数据源而产生的漏报。为验证方法的有效性,作者基于编译技术实现了一个可实际运行的原型检测系统 DELTA (DEtect vuLnerability with sTatic Analysis)。使用 DELTA 对 Linux 内核代码的漏洞检测实验表明,本文方法能够有效检测出系统中存在的安全漏洞,并能避免其他主流静态检测工具中存在的漏报和误报,为提高软件质量及安全性提供了有效可行的方法及工具。

本文第 2 节给出一种基于程序安全状态的安全漏洞检测方法;第 3 节描述 DELTA 的设计及实现;第 4 节描述对 Linux 操作系统内核源代码的静态检测实验,展示方法的应用效果;第 5 节介绍相关工作并进行分析比较;最后是全文的总结。

2 检测方法

2.1 概述

本文检测方法的基本思想是对程序状态的跟踪和检查。如图 1 所示,在本文中,将使用基于有限状态机模型的漏洞状态机,描述程序变量安全状态的转换规则,针对待检测漏洞,设置相应的漏洞状态机,规定相关数据的状态转换;并对程序各可能执行路径进行静态遍历并识别当前操作,对当前操作所涉及的程序变量根据状态机赋予其对应的安全状态;在安全相关操作处设置检查点(checkpoint),以检测操作数据是否具有期望的安全状态,若出现与期望安全状态不符的情况,则表示发现了一个可能的安全漏洞。

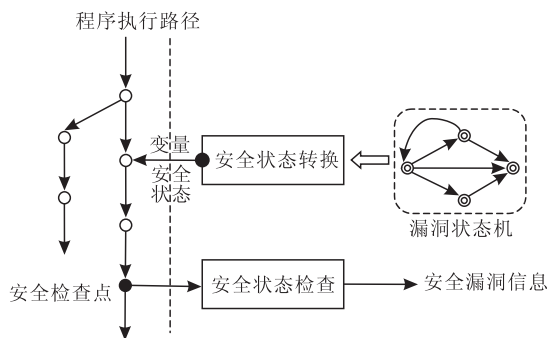


图 1 检测方法示意图

以拷贝超长数据所导致的缓冲区溢出漏洞为例,当使用内存拷贝函数 *memcpy*、根据一个未经上界检查的用户输入的操作长度参数复制超长的数据至目标缓冲区时,将发生缓冲区溢出。此时,安全漏洞的相关要素为安全相关操作——内存拷贝函数;

操作数据状态——用户输入的(非可信的)、未经上界检查的操作长度参数。内存拷贝操作将为一个安全检查点,检测系统所期望的操作长度参数为一个可信数据或经过上界检查的非可信数据。当一个用户输入的数据进入系统处理流程时,将被标识为非可信数据。若经过了有效的上界检查,其状态中将加入通过上界检查的标识;否则,在作为内存拷贝操作长度参数时,检测系统将认为其不具有期望的安全状态,会引发一个缓冲区溢出。

2.2 漏洞状态机模型

在传统的漏洞静态检查中所采用的状态机模型中,数据的安全状态使用单个属性值标识,如 $MC^{[6]}$ 和 $XGCC^{[7]}$ 等。在本文中,为精确跟踪多种平行发展的数据安全性质(主要为数据的可信性和合法性检查状态),对漏洞状态机的状态空间进行了扩展。

首先给出以下模型元素。

SRP: 变量安全相关属性集。SRP 包含各种细化后的安全漏洞相关变量特征,例如数据来源、已经经过的合法性检查(可能为多个)、指针释放与否等。不失一般性,设 SRP 集合元素个数为 n ,属性被赋予 $1 \sim n$ 的编号;

STO: 安全状态转换操作集。STO 包含各种导致变量安全相关属性发生变化的操作,例如赋值、拷贝、数据比较检查等;

$opp: STO \rightarrow P(\{1, 2, \dots, n\})$ 。安全状态转换操作到 SRP 编号集的幂集上的一个映射,标识 STO 中的元素所对应的相关 SRP 属性编号集;

$opv: STO \times \{1, 2, \dots, n\} \rightarrow \{true, false\}$, 标识一个 STO 操作对相关的某个 SRP 属性的状态设置,为 true 表明此 STO 操作导致变量具备了相关的属性或被执行了相关操作。

在漏洞状态机中,将使用一个对应 SRP 集元素的布尔向量描述变量安全状态。基于有限状态机模型,一个漏洞状态机为以下一个五元组:

$$VM = \langle S, \Sigma, f, s_0, Z \rangle.$$

漏洞状态机各元素的定义如下:

S: 安全状态集,为对应 SRP 集的 n 维布尔向量集,即

$$S = \{true, false\}^n \\ = \{(s_1, s_2, \dots, s_n) \mid s_i = true \text{ 或 } false\},$$

其中,分量 s_i 对应编号为 i 的 SRP 属性;

Σ : 状态机控制字符集,为状态机对应安全漏洞的安全相关操作,即 $\Sigma = STO$;

f : $S \times \Sigma \rightarrow S$, 状态转换函数,表示在当前状态下实施一个安全相关操作后的后续安全状态。对于

一个安全状态 $s = (s_1, s_2, \dots, s_n)$ 及一个 STO 操作 op , 设 $f(s, op) = s' = (s'_1, s'_2, \dots, s'_n)$, 其中

$$s'_i = s_i, \text{ 当 } i \notin opp(op);$$

$$s'_i = oppv(op, i), \text{ 当 } i \in opp(op).$$

$s_0 = \{\text{false}, \text{false}, \dots, \text{false}\}$, 初始状态, 其所有元素为 false, 表示变量未具备任何安全相关特性或执行了相关操作;

Z : 终止状态集, 在漏洞状态机中此集为空, 即 $Z = \emptyset$.

在漏洞状态机中, 安全状态由对应 SRP 集元素的布尔向量表示, 标识程序变量是否具备相关安全属性或执行了什么操作. SRP 集的设置由具体的目标系统与待检测的漏洞类型决定, 主要由 3 部分构成:

(1) 变量值是否为一非可信的数据, 即此数据是否有可能被恶意用户所控制.

对于软件系统而言, 其存在的每一个安全相关编程错误不一定能最终导致对系统安全的危害, 只有当攻击者构造的恶意数据能够触及到有缺陷的代码时, 方可能导致实际的安全威胁. 否则, 其仅仅是一个难以被触发的编程错误而非一个安全漏洞. 因此, 对变量可信性的标识是通过变量状态跟踪以检测安全漏洞的重要基础. 在实际检测中, 主要将对非可信数据进行跟踪, 安全状态布尔向量中将设置一个分量标识变量是否源自一个非可信数据.

(2) 变量通过了的合法性检查.

实际系统中, 在引用外部非可信数据前, 往往需要进行合法性检查, 例如对外部指定的内存操作长度进行上界检查. 在静态检测中对合法性检查的识别较为复杂, 对于不同的待检测漏洞, 所涉及的合法性检测不同. 即使是同一漏洞类型, 各系统中实现的合法性检测的静态表现形式也存在差异. 此外, 合法性检查还可由多个步骤组成, 如首先检查操作长度变量的符号, 再检查其上界.

如图 2 所示, 为了能较准确地识别合法性检查, 拟细化合法性检查的识别粒度, 将合法性检查分解成一序列相关的子步骤进行识别. 在 SRP 中, 将针对具体的漏洞相关的合法性检查构成, 设置与之对应的多个变量安全相关属性, 用于标识目标变量所经过的合法性检查步骤. 而且, 每个步骤可细化为对应着一个检测引擎可单步识别的基本语法单位, 从而保证了精确识别. 安全状态布尔向量中的相应分量用于标识此变量是否通过了相应的合法性检测步骤. 此外, 合法性检查各阶段的检查结果被单独记录在变量的安全状态中, 各阶段的检查结果互不覆盖,

在检查点上再对变量是否具有期望的安全状态进行检查. 从而不必前摄性地规定合法性检查各步骤的排列次序, 符合实际系统程序设计的规律.

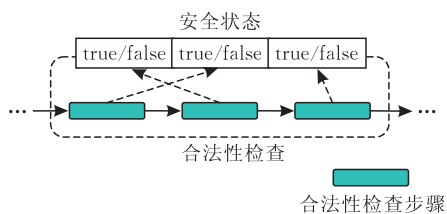


图 2 合法性检查细化识别

如此, 可将合法性检查的识别与变量状态的跟踪统一到漏洞状态机中, 既提高了合法性检查的识别准确性, 又保证了漏洞检测机制的简洁高效.

如若使用一元的安全状态描述机制, 必将需要在较大的粒度上对合法性检查进行分析判断, 这需要相应的程序模式归纳. 而涉及合法性检查的各种可能的构成形式及排列次序很难归纳为相对固定的表示序列, 这必将导致对合法性检查程序模式的部分缺失. 例如, 有符号整数的有效上界检查就可能包含多种构成形式. 每种形式都可能包含多个次序可以任意变换的步骤, 如通过判断整数是否非负和是否小于一个常量上界来进行检查. 多元的安全状态描述机制能够较自然地容纳对这种不定次序的多步骤检查的识别. 而在采用一元安全状态描述机制的 MC 中^[6], 整数上界检查被作为一个整体进行识别判断, 此判断只能在一个不全面的上界检查特征归纳基础上进行, 不可避免地会降低识别分析的精度. 对 Linux 内核蓝牙设备驱动安全漏洞的漏报就是一个直接的后果.

(3) 变量的操作实施状态.

释放后指针引用、多次释放等漏洞类型还涉及到对目标变量操作实施的流程, 特别是对指针型变量的操作流程. 为此, 在 SRP 中还设置了相应的元素标识对目标变量已实施的操作, 例如针对释放后指针引用类型漏洞, 设置了标识指针已被释放的状态元素.

从以上分析描述可见, 对安全状态空间进行的多元化扩展并不是数量上简单的增加, 而是为了更好地符合程序设计自然规律、提高检测精确度所进行的必然选择.

2.3 非可信数据鉴别

漏洞状态机的核心是状态转换的处理, 即状态转换函数 f 的确定. 变量在声明后, 其初始安全状态为 s_0 , 其后续状态根据其上所实施的相关操作 (Σ 中元素) 确定. 当变量被直接或间接赋予一个可能被

恶意用户所控制的数值时,将设置其安全状态中的相关分量,标识其为一非可信的数据.数据可信性的确定与具体目标系统的结构和数据处理机制密切相关,基于软件系统的一般结构特征,引入以下概念支持非可信数据的鉴别.

定义 1. 可信边界(trusted boundary)是目标系统不受外界影响的数据生成处理边界,在可信边界内生成的数据不受外部操作的影响.

定义 2. 可信边界入口(entry point)是外部数据进入可信边界的通道.

界定可信边界最重要的目的在于鉴别非可信数据,即通过明确外部数据输入途径来确定哪些数据源自非可信的外部世界.可信边界上的所有内外数据通道构成了可信边界入口.在界定了可信边界后,以其为线索可确定可信边界入口.现代软件系统,特别是系统软件,一般将其所提供的底层服务集成在一个相对独立的核心组件中,通过接口对外提供服务.以操作系统为例,其内核通过系统调用接口为应用程序提供了底层的计算服务,内核中的数据生成与操作不受用户态应用程序控制.这种共性的体系结构为可信边界的界定提供了良好的基础,可结合待检测漏洞的特性对目标系统体系结构进行安全性分析,界定具体的可信边界.可信边界入口不仅仅是系统服务的调用接口,还包括所有可能的外部数据输入机制,例如操作系统内核协议栈中对远程网络数据包的接收等.

可信边界入口将被作为非可信数据源.若目标系统中的可信边界入口集为 EP ,其应属于安全状态转换操作集 STO ,即 $EP \subseteq STO$.不失一般性,假设标识非可信数据的 SRP 属性编号为 1,即安全状态向量中的第 1 个分量用于标识数据是否可信.对于变量 v ,对应的漏洞状态机状态转换函数 f_v 为

$$\forall ep \in EP, f_v(s, ep) = (\text{true}, \text{false}, \dots, \text{false}),$$

其中 s 为 v 的当前安全状态, $s \in \mathbf{S}$.需要注意的是,当变量 v 通过可信边界入口接收了外部数据后,其状态向量中的除第 1 个分量外的其它分量都被初始化为 false .这是因为当变量被非可信数据感染后,其以前所通过的合法性检查等安全状态设置需要重新计算.

除了在可信边界入口处标识非可信数据外,还需要考虑变量间的安全状态传播,其核心是非可信数据的转播.变量间的状态转播主要包含以下 2 种情况:

(1) 赋值传播

当一变量作为一赋值表达式的左值(LValue)

时,其安全状态将由表达式右值(RValue)表达式安全状态决定.对于各种形式的右值表达式 S ,其安全状态 $sstate(S)$ 按以下情形计算:

①当 S 为单个变量 x 时, $sstate(S) = state(x)$, 其中 $state(x)$ 为变量 x 的安全状态;

②当 S 为单个常量时, $sstate(S) = s_0$;

③当 S 为对函数 fun 的调用时, $sstate(S) = state(\text{return}(fun))$, 其中 $\text{return}(fun)$ 为函数 fun 当前调用实例的返回值;

④当 S 由多个子表达式组成,如 $S1$ 和 $S2$, 当 $first_component(sstate(S1)) = \text{true}$ 或 $first_component(sstate(S2)) = \text{true}$ 时, $sstate(S) = (\text{true}, \text{false}, \dots, \text{false})$, 否则 $sstate(S) = s_0$, 其中 $first_component(S)$ 为状态向量 \mathbf{S} 的第 1 个分量.

在赋值传递中,考虑到安全漏洞检测的特殊性,对于由多个子表达式组成的右值表达式,本方法采取一种较为激进的方法,表达式安全状态将只考虑各子表达式是否会引入非可信的数据,而忽略其它 SRP 属性的传播.

(2) 内存拷贝传播

除了直接赋值以外,内存的拷贝操作也会引起安全状态的传播,源内存区域数据的安全状态将影响目的内存区域数据的安全状态.以内存拷贝函数 $memcpy$ 为例,进行 $memcpy(d, s, len)$ 调用后,指针 d 指向的数据的安全状态将等于指针 s 指向的数据的安全状态.

2.4 数据合法性检查

为提高识别的准确性,合法性检查被分解成一系列相关的子步骤进行识别,当非可信变量通过了子步骤相应的判断后,变量安全状态向量中将相应的分量设置为 true ,标识其通过了对应的合法性检查步骤.对非可信数据的合法性检查主要有两种方式:逻辑运算判断,例如检查是否大于、小于、等于或不同于某个值;合法性检查例程,通过调用系统中内建的合法性检测例程进行判断.在实际实施中,将针对具体的目标系统,总结各种漏洞类型相关的合法性检查模式,细化为一系列对非可信数据的检查步骤,对应着相应的状态设置,最终基于状态机模型实现合法性检查的识别与变量的状态转换.下面以操作长度越界引发的内存非法操作漏洞为例进行具体的解释.相应漏洞状态机的 SRP 集设置如下:

$$SRP = \{\text{tainted}, \text{unsinged upper-bound checked}, \text{singed upper-bound checked}, \text{non-negative}\}.$$

以上 SRP 集由 4 个元素组成,分别对应数据是否非可信、通过了无符号上界检查、通过了有符号上

界检查、通过了非负判断等,对应编号为 1 至 4. 在以上 SRP 集下,可能的安全状态向量共有 16 个 (2^4),完全的状态转换图较为复杂. 为了描述方便,图 3 给出了一个简化的状态转换示意图. 一个整型变量的初始状态为 s_0 ,表示其未被非可信数据污染并且未经过任何安全检查;当此变量被赋予了一个非可信值后状态转换到 $(\text{true}, \text{false}, \text{false}, \text{false})$,表示其受到非可信数据污染;在处于非可信状态的变量经过了上界检查后,其状态转换到 $(\text{true}, \text{true}, \text{false}, \text{false})$ 或 $(\text{true}, \text{false}, \text{true}, \text{false})$,表示一个非可信整数通过了相应的无符号或带符号上界检查;若此变量为一有符号整型,还可能对其进行非负判断,状态将转换至 $(\text{true}, \text{false}, \text{true}, \text{true})$;在任何状态下,若变量受到非可信数据污染,其状态都将转换到 $(\text{true}, \text{false}, \text{false}, \text{false})$,类似的,若变量被赋予了一个可信值,其状态变为 s_0 .

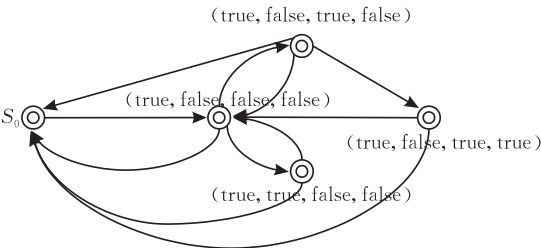


图 3 漏洞状态机状态转换图

上界检查与整型变量的符号类型和密切相关. 对于一个非可信的无符号整型变量,如果通过了一个与其它数据进行的上界比较判断,则其状态转换为 $(\text{true}, \text{true}, \text{true}/\text{false}, \text{true}/\text{false})$;对于非可信的有符号的整型变量,需根据与之进行比较的数据类型决定其状态转换:

- ① 通过了与常量进行的上界比较判断,则状态转换为 $(\text{true}, \text{true}/\text{false}, \text{true}, \text{true}/\text{false})$;
- ② 通过了与有符号变量进行的上界比较判断,则状态转换为 $(\text{true}, \text{true}/\text{false}, \text{true}, \text{true}/\text{false})$;
- ③ 通过了与无符号变量进行的上界比较判断,则状态转换为 $(\text{true}, \text{true}, \text{true}/\text{false}, \text{true}/\text{false})$;

此外,如果有符号整型变量为非负整数(通过了一个非负判断,例如判断是否不小于 0),可以将其作为一个无符号整型变量对待. 因此,需要在安全状态向量中加以标识,通过非负判断的非可信变量状态将转换为 $(\text{true}, \text{true}/\text{false}, \text{true}/\text{false}, \text{true})$.

在合法性检查的识别中进行这样的区分有着非常重要的现实意义. 这是因为在进行带符号的比较判断时,一个负值一定小于正值. 而在 C/C++ 语言

中,内存操作函数的长度参数和数组索引将被作为无符号整型处理. 一个被赋予负值的有符号整型变量能通过使用带符号比较判断的上界合法性检查,但当其被作为内存操作函数的长度实参或数组索引时,将被解释为一个非常大的无符号正值从而导致一个缓冲区溢出或非法地址访问漏洞. 由于历史原因,程序员往往忽视了这种有符号变量和无符号变量混用的危险,导致此类安全漏洞层出不穷. 由此,在内存操作(例如 *memcpy*)和数组下标等检查点上,相关变量的合法安全状态应为以下之一:

- ① s_0 ;
- ② $(\text{true}, \text{true}, \text{true}/\text{false}, \text{true}/\text{false})$,标识其通过了一个无符号上界检查;
- ③ $(\text{true}, \text{false}, \text{true}, \text{true})$,当变量仅仅通过了一个带符号上界检查时,期望其为一非负值. 否则将表明发现了一个可能的安全漏洞,例如将状态为 $(\text{true}, \text{false}, \text{true}, \text{false})$ 的变量作为 *memcpy* 函数的长度实参.

图 3 中仅仅给出了一部分可能的安全状态间的一个不完全的状态转换图,所有 16 个可能状态间的转换远比图 3 中所示的情况复杂. 特别是 SRP 集还可能引入新的元素以识别更多的合法性检查,例如下界检查. 这将导致更为复杂的状态转换. 若在检查系统中直接记录并实现所有的状态转换规则,必定会导致实现及效率上问题,并会大大影响系统的可扩展性. 但由于大部分状态转换仅仅改变安全状态向量中的一个分量,状态转换规则可被大大简化. 如表 1 所示,对于以上状态机,其所有合法性检查相关的状态转换规则可被简化为 4 条. 2.3 节中涉及到的其余状态转换规则也可同理简化.

表 1 合法性检查相关状态转换规则

前状态	STO	后状态
(s_1, s_2, s_3, s_4)	变量为无符号数时:任意上界比较	$(s_1, \text{true}, s_3, s_4)$
	变量为有符号数时:常量上界比较、有符号上界比较	$(s_1, s_2, \text{true}, s_4)$
	变量为有符号数时:无符号上界比较	$(s_1, \text{true}, s_3, s_4)$
	非负判断	$(s_1, s_2, s_3, \text{true})$

2.5 检查点

在使用漏洞状态机对程序数据安全状态进行跟踪的基础上,对安全漏洞检测的实施将通过在一些安全攸关的操作点上对当前操作所涉及的数据变量的安全状态进行检查来进行. 一些在本文实验中所采用的检查点及在 Linux 系统中的部分实例如表 2 所示.

表 2 检查点

检查点	相关参数	状态检查	相关安全漏洞	实例(Linux 内核)
内存分配、拷贝, 设置等操作	操作长度、操作地址	操作长度和操作地址若为非可信数据, 必须经过上界检查或访问合法性检查;	缓冲区溢出、信息泄漏、非法内存访问等	<i>kmalloc</i> , <i>kmem_cache_alloc</i> , <i>memcpy</i> , <i>copy_from_user</i> , <i>copy_to_user</i> , <i>memset</i> , ...
数组索引	索引	非可信的索引数值应经过上界检查	非法内存访问、非法函数调用等	数组元素引用, 如 <i>array[x]</i>
指针引用	操作指针	指针未被释放过	释放后指针引用	指针引用, 如 <i>*p</i> , <i>p->q</i> , ...
循环	循环次数	非可信的索引数值应经过上界检查	死循环导致的拒绝服务	

需要特别指出的是, 在 Linux 内核中, *copy_from_user* 等函数既是可信边界的入口, 又是安全漏洞检测的检查点.

3 原型系统

为检验以上检测方法的效能, 作者实现了一个针对系统软件的静态检测系统原型 DELTA, 能支持对 C/C++ 语言开发实际的目标系统的漏洞检测实验.

3.1 系统结构

如图 4 所示, DELTA 系统首先对待检测的系统源代码进行预处理和代码解析, 并形成中间代码形式输出至一个静态分析引擎. 引擎将在漏洞模式(状态机)支持下进行安全漏洞检测, 主要机制为通过遍历代码中的执行路径驱动漏洞状态机运行来跟踪变量的安全状态, 并在检查点上将相关变量的安全状态与期望安全状态进行比对. 若发现可能的安全漏洞将输出相应的漏洞上下文信息.

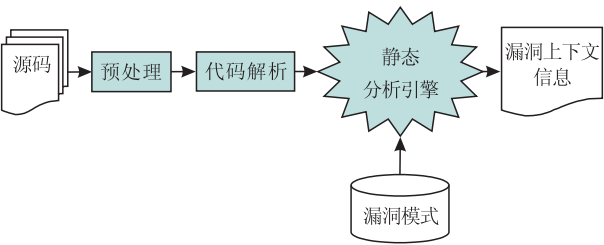


图 4 DELTA 系统结构

DELTA 系统基于编译技术实现:

(1) 预处理器与解析器

预处理器与解析器相当于编译器的前端(Front-end), 负责将原始源代码文件转换成检测引擎能够识别处理的形式. 原始的程序源代码文件往往引用了其他一些说明性的文件, 如 C/C++ 语言中的头文件. 这些文件通过语言的包含机制相互连接起来. 预处理工具将模拟真实编译器编译链接过程, 识别根文件, 以根文件为基础将相关的源文件整合在一起, 形成完整的分析单位. 解析器将对预处理

过的文件进行词法分析与语法分析, 并生成面向静态分析的中间代码.

DELTA 中的预处理器与解析器基于 GCC 编译器的前端实现, 利用了 GCC 中的 C/C++ 语言标准的 lex、yacc 描述脚本为基础生成相应的词法与语法分析器. 在语法分析结束后, 将根据所获得的语法元素生成静态单赋值(Static Single Assignment, SSA)形式的中间代码^[8], 中间代码将以基本块(basic block)的形式组织在一起.

(2) 静态检测引擎

静态检测引擎将首先根据源代码函数间的调用图(Call Graph, CG)确定根函数(root function), 并构建控制流图(Control Flow Graph, CFG). 以根函数为起始点, 用 CFG 驱动漏洞状态机运行, 模拟源代码文件的实际操作, 遍历其相应中间代码的各个可能的逻辑执行路径. 在状态机运行过程中, 引擎将根据当前操作和各个变量的安全状态决定其的后继状态. 若当前程序点为一个漏洞检查点时, 引擎还将对相关变量的当前安全状态进行检查. 若出现不符合当前检查点对变量的安全要求的情况, 则表示发现了一个可能的安全漏洞, 引擎将向用户输出相关漏洞上下文信息(主要包括漏洞位置、数据传播路径等).

在很多情况下, 安全漏洞的产生跨越多个函数过程, 甚至是多个源文件. 为了能有效检测出这类安全漏洞, 分析引擎应能实施过程间分析(inter-procedural analysis)和文件间分析(inter-file analysis). 过程间分析需要对函数调用进行跟踪分析. 若当前语句为一个函数调用时, 将跟踪被调用函数(Callee)的中间代码; 在分析完毕后, 返回到调用点继续分析调用函数(Caller). 文件间分析的基础是过程间分析, 即当调用函数与被调用函数位于不同源文件时, 应能跨越源文件进行过程间分析. 文件间分析需要逐个解析待分析的源文件, 然后联立多个文件的解析结果并构建跨文件的全局 CG, 确定根函数后进行过程间分析.

在 DELTA 中, 每一种漏洞类型模式对应着一个

漏洞状态机. 检测引擎中可同时运行多个漏洞状态机,以同时检测多种安全漏洞. 在实际应用中,可根据资源情况选择装载待检测的漏洞状态机. DELTA 系统中的漏洞状态机以检查器(Checker)链接库形式实现. 在静态检测引擎中设置了钩子(Hook)函数,涵盖状态转换、状态传播和检查点等程序节点,检测引擎将在这些点上调用已装载的检查器中的实施函数.

3.2 系统配置

为验证上述方法及原型的有效性,作者选择了 Linux 内核作为实验对象,使用 DELTA 对其进行

了漏洞检测实验.

作为检测的基础,首先需要确定 Linux 系统可信边界和可信边界入口. 根据 Linux 操作系统内核的结构特点,完全源于内核中的数据的安全性不会被用户态操作所影响,可信边界可依据系统内核边界界定. 但需要特别注意的是,虽然基于内核良好的结构化可较容易地确定可信边界,但 Linux 内核中外部数据输入途径并不都是直接明晰的,需要依据可信边界深入进行分析归纳. 通过分析, Linux 内核相应的可信边界入口类型如表 3 所示.

表 3 Linux 内核可信边界入口

可信边界入口	引入的非可信数据	实例
系统调用接口	用户态程序提供的调用参数	<i>sys_open, sys_read, sys_write, sys_mknod, sys_ioctl</i> 等
内核态/用户态数据交换例程	拷贝自用户态内存的数据	<i>copy_from_user, get_user</i> 等
网络包读取	源自网络协议栈传入的远程网络包数据	<i>sk_buff->data</i> 等
可执行程序等用户文件装载解析	载入内核的用户可构造的文件数据	<i>linux_binprm->buf</i> 等

在表 3 中所包括的可信边界入口中,文件的装载解析最容易被遗漏而导致漏报. 事实上, Linux 内核在装载可执行程序等文件时,会从这些文件中引入影响内核操作的数据. 例如,装载执行一个恶意构造的可执行二进制文件,会间接地影响内核 *task_struct* 等关键数据结构中某些域值,对这些域

值的不适当处理可能会导致安全漏洞.

针对最为常见的漏洞类型,作者实现了 5 个基于漏洞状态机的检查器,分别用于检测缓冲区溢出、非法数组下标与非法指针引用、空指针引用、内存泄漏等类型漏洞,相应的漏洞检测设置见表 4.

表 4 漏洞检测设置

漏洞类型	描述	漏洞状态机设置	检查点
操作长度引发的缓冲区溢出	当根据一个未经上界检验的非可信操作长度数进行内存拷贝时,可能会引发内核堆/栈缓冲区溢出	非可信数据源设置见表 2; 非可信上界检查数据状态转换参见 2.4 节	<i>memcpy, copy_from_user, get_user, __copy_from_user_ll</i> 等内核函数,期望状态为 s_0 、 $(true, true, true/false, true/false)$ 或 $(true, false, true, true)$
非法内存操作	当根据一个未经上界检验的非可信操作长度数进行内存分配、设置等操作时,可能会引发非法内存访问和内存耗尽	同上	<i>kmalloc, kmem_cache_alloc, memset</i> 等内核函数,期望状态同上
信息泄露	当根据一个未经上界检验的非可信操作长度数进行内核态至用户态数据交换时,可能会引发关键的内核数据泄露	同上	<i>copy_to_user, put_user, __copy_to_user_ll</i> 等内核函数,期望状态同上
非法数组下标及循环次数	将一个未经上界检验的非可信数据作为数组下标时可能会引发非法内存访问;当将其作为循环次数(上界)时会引发死循环	同上	数组元素引用,如 $array[x]$,期望状态同上;循环次数判断,当循环变量与循环上界比较为无符号比较时,期望状态同上,否则期望状态为 s_0 、 $(true, true, true/false, true/false)$ 或 $(true, true/false, true, true/false)$
引用释放后的指针	通过一个已经释放了的指针访问数据,可能引发非法内存访问	当一指针被释放后,其状态被置为 $(true)$,在重新挂指针后其状态置为 $(false)$	指针引用,如 $*p, p->q$ 等,期望状态为 $(false)$

4 实 验

基于以上设置,作者对 Linux 内核的一些子系统进行了实验检测. 实验结果表明本方法和原型系

统能有效检测出隐藏较深的安全漏洞,其中一些是其他静态检测方法未能发现的. 此外,实验表明 DELTA 还能有效避免缺乏对有效合法性检查的识别所带来的误报. 以缓冲区溢出漏洞检测为例,如表 5 所示, DELTA 检测出了 7 个真实的漏洞,准确

率达到了 31.8%,和 CQUAL 与 Coverity 对 Linux 内核检测结果的比较如表 5 所示。

表 5 缓冲区溢出漏洞检测结果及比较

检测系统	报告漏洞数	真实漏洞数	准确率/%
DELTA	22	7	31.8
CQUAL	264	6	2.2
Coverity	124	15	12.0

DELTA 所检测出的所有漏洞分布情况是:操作长度引发的缓冲区溢出 7 个,非法内存操作 1 个,信息泄露 1 个,非法数组下标及循环次数 2 个,引用释放后的指针 1 个。表 5 中 CQUAL 与 Coverity 对 Linux 的检测结果数据来自其相关文献报道^{[5]①}。由表 5 可见,DELTA 系统对 Linux 内核的检测准确率高于 CQUAL 与 Coverity 系统,即具有较低的误报率。相对于 Coverity 而言,DELTA 检测出的安全漏洞总数较少。原因在于:Coverity 实施了超过 50 个的检查器,在对缓冲区溢出的检测方面,覆盖了各种引发缓冲区溢出的因素。而 DELTA 的主要目标在于检验本文方法在降低漏报与误报方面的效果,仅实现了对操作长度引发的缓冲区溢出漏洞的检测。虽然如此,在同样覆盖的漏洞类型上,DELTA 仍然检测出 Coverity 所未能检测出的漏洞,避免了漏报。DELTA 系统所检测出的 2 个典型漏洞实例如下:

(1) 如下列代码所示,在 2.6.9 版本内核 fs/binfmt_elf.c 文件中,存在一个内核缓冲区溢出漏洞,虽然在 *copy_from_user* 函数引用有符号整型变量 *len* 前对其进行了合法性检查(将其与一个常量 *ELF_PRARGSZ* 进行比较,程序 1228 行),但通过赋予其一个负的长度值就能绕开此检查。因此,当调用 *copy_from_user* 函数时(程序 1230 行)将会引发一个内核栈缓冲区溢出。

```
static void fill_psinfo(struct elf_prpsinfo *psinfo,
                      struct task_struct *p,
                      struct mm_struct *mm)
{
1222 int i, len;
...
1227 len=mm->arg_end-mm->arg_start;
1228 if (len>=ELF_PRARGSZ)
1229     len=ELF_PRARGSZ-1;
1230 copy_from_user(&psinfo->pr_psargs,
                (const char __user *)mm->arg_start,len);
...
1419 fill_psinfo(psinfo,current->group_leader,
                current->mm);
```

使用 DELTA 对上述代码进行检测时,变量 *len* 来自 2 个非可信数据的计算结果(程序 1227 行),这是因为用户能够通过装载一个恶意构造的二进制文件来控制当前进程(*current* 宏)内核数据结构 *task_struct* 中的一些域,变量 *len* 的安全状态将转换为 (true,false,false,false)。当程序 1228 行的判断为假时,表明其通过了一个有符号的上界检查,安全状态转换为 (true,false,true,false)。DELTA 在 *copy_from_user* 函数处设置了检查点,对其第 3 个参数安全状态进行检查,期望对应参数的安全状态为 *s₀*、(true,true,true/false,true/false)或 (true,false,true,true)。参数变量 *len* 的安全状态不符合此要求,DELTA 将输出发现一可能安全漏洞的信息,相应程序执行路径为...→1419→...→1227→1228→1230。

(2) 如下列代码所示,在 2.4.20 版本内核 *drivers/i2c/i2c-dev.c* 文件的 *i2cdev_ioctl* 函数中,存在一个非法内存访问漏洞和一个内核缓冲区溢出漏洞。与上例类似,*rdwr_arg.nmsgs* 为一个非可信数据,其未经检查就被引用为循环上限和数组下标,将导致非法地址访问或死循环;此外,第 279 行使用未经检查的非可信数据作为 *copy_from_user* 的操作长度参数,将导致一个缓冲区溢出。

```
252 if (copy_from_user(&rdwr_arg,
253                   (struct i2c_rdwr_ioctl_data *)arg,
254                   sizeof(rdwr_arg)))
255     return -EFAULT;
...
264 for( i=0; i<rdwr_arg.nmsgs; i++)
265 {
...
279 if(copy_from_user(rdwr_pa[i].buf,
280                  rdwr_arg.msgs[i].buf,
281                  rdwr_pa[i].len))
282     {...
```

特别重要的是,实验还发现了新的未知漏洞,检测出内核 ATM Over IP 模块中对释放后指针引用的未知安全漏洞。此漏洞已经提交国际漏洞权威数据库 CVE 和 Linux 内核开发组织,并得到认可,获得的 CVE 编号为 CVE-2006-4997^②。

此外,作者还使用 DELTA 对已经修正了安全漏洞的内核版本代码进行了检测,以验证其避免误

① Coverity Inc. Report. Analysis of the Linux Kernel. Dec. 2004
② Linux Kernel ATM SkBuff Dereference Remote Denial of Service Vulnerability. <http://cve.mitre.org/cgi-bin/cve-name.cgi?name=CVE-2006-4997>

报的效果. 以(2)中所示漏洞为例, 在对已修正相关漏洞的最新的 2. 6. 20 版内核检测时, DELTA 未再报告检测出相应漏洞.

5 相关工作与比较

斯坦福大学 Engler 的研究团队研发了多个相关的静态漏洞检测工具 MC、XGCC 和 MECA^[6-7, 9], 后发展成为一个商业产品 Coverity, 检测出数以百计的 Linux 内核安全漏洞, 代表了目前静态分析领域的最高水平. 与之比较, 本文工作对状态机模型的状态空间进行了扩展以提高漏洞特征和合法性检查识别的精度, 并建立系统化的非可信数据鉴别方法, 能够更为有效地对隐藏较深的安全漏洞进行检测. 在实验中, 使用本文方法对已经被 Coverity 等工具全面检测过的 Linux 内核版本代码进行检测, 发现了其未能检测出的安全漏洞, 如第 4 节实例(1). 出现这种情况的根本原因在于这些工具未能全面系统地对非可信数据进行鉴别. 就检测模型而言, 与 Coverity 等工具相比, 本文采用了对应多个安全相关属性的布尔向量多元化地标识变量安全状态, 能为高精度的漏洞特征识别和合法性检查识别提供基础, 而非简单地增加数量. 特别是通过多元化的安全状态标识机制, 具备了识别检测各种排列次序的多步骤合法性检查的能力, 能较全面地覆盖各个合法性检查相关要素, 如整数上界比较符号问题, 避免了 Coverity 对于 Linux 内核蓝牙设备驱动中符号相关安全漏洞的漏报.

另外一种主流的漏洞静态检测技术是类型检测, Johnson 等人使用类型限定词技术来检测 Linux 内核中的用户/内核态指针 (User/Kernel Pointer) 漏洞^[8], 扩展了 CQUAL 的基本类型推理能力以支持上下文敏感性和更高的结构数据分析精度. 与之类似, Shankar 等人使用 CQUAL 来检测格式化字符串 (Format String) 漏洞^[10]. 此方法存在的最大的问题在于缺乏对合法性检查的识别机制, 且遗漏了关键的检查点, 导致较多的漏报和误报. 以第 4 节实例(2)为例, 依据文献[5]报告, Johnson 等人能够在 2. 4. 20 版内核中检测出 i2cdev_ioctl 函数中缓冲区溢出漏洞; 但对已经修正了的 2. 6. 20 版内核, 实验表明其仍然将修正过的代码识别为存在漏洞. 此外, 对于 i2cdev_ioctl 函数中非法地址访问漏洞, 无论代码是否进行了修正, 均未能检测出相关漏洞. 在这个实例上, 本文方法能提供精确的结果.

Volanschi 等人对 GCC 进行了扩展, 形成了一

个用于安全漏洞静态检测工具 MYGCC^[11]. DELTA 也借助了 GCC 编译器的前端, 其体系结构与 MYGCC 类似. 但 MYGCC 的检测方式限于时序类安全漏洞的检测, 如内存泄漏等, 难以支持较复杂的程序特性分析, 无法覆盖危害最大的缓冲区溢出类的安全漏洞检测. 而使用状态机模型对漏洞特征进行识别检测具有较好的扩展性, 能支持多种类型的漏洞检测. 此外, MYGCC 还缺乏文件间分析的能力. 与之类似, Chen 等人研发的基于模型检测 (Model Checking) 的静态检测工具 MOPS, 也同样仅限于对时序类安全漏洞的检测^[12].

6 结 语

针对目前安全漏洞静态检测方法中所存在的漏报和误报问题, 本文从提高检测模型描述能力以支持数据合法性检查的识别分析角度出发, 提出了一种对程序数据安全状态进行跟踪检查的漏洞静态检测方法. 在此方法中, 使用有限状态机模型描述程序数据安全性质的变化, 对数据安全状态的状态空间进行了扩展, 使用对应多个安全相关属性的布尔向量标识变量安全状态, 以跟踪分析多种平行发展的数据安全性质, 并细化了状态转换的粒度以提供更为精确的程序特征识别. 在此基础上, 在漏洞状态机中引入了对合法性检查及其有效性的识别, 有效降低了由此引起的误报和漏报的发生. 此外, 在本文中還引入了可信边界与不可信边界入口的概念, 以可信边界的界定为线索建立了系统化的非可信数据鉴别方法, 防止了由于遗漏非可信数据源而产生的漏报. 基于以上方法, 还实现了一个能支持过程间和文件间分析的原型检测系统 DELTA. 检测实验结果表明: 本文方法能够有效检测出系统中存在的缓冲区溢出等类安全漏洞, 能有效避免其他主流静态检测工具中在合法性检查识别和非可信数据鉴别方面的不足所导致的漏报和误报. 实验还检测出了一些 Linux 内核中存在的未知漏洞, 相关结果已经得到了国际漏洞权威数据库 CVE 和 Linux 内核开发组织的承认.

本文方法对现有漏洞静态检测方法的不足进行了有益而重要的补充, 提供了有效可行的降低漏报与误报的技术手段, 有助于漏洞静态检测技术的进一步推广应用.

参 考 文 献

[1] Howard M, LeBlanc D, Viega J. 19 Deadly Sins of Software

- Security: Programming Flaws and How to Fix Them. USA: McGraw-Hill Osborne Media, 2005
- [2] Sutton M, Greene A, Amini P. Fuzzing: Brute Force Vulnerability Discovery. USA: Addison-Wesley Professional, 2007
- [3] Chess B, West J. Secure Programming with Static Analysis. USA: Addison Wesley Professional, 2007
- [4] Chess B, McGraw G. Static analysis for security. IEEE Security & Privacy Magazine, 2004, 2(6): 76-79
- [5] Johnson R, Wagner D. Finding user/kernel pointer bugs with type inference//Proceedings of the 2004 USENIX Security Symposium. San Diego, CA, USA, 2004: 119-134
- [6] Ashcraft K, Engler D. Using programmer-written compiler extensions to catch security holes//Proceedings of the 2002 IEEE Symposium on Security and Privacy. Oakland, CA, USA, 2002: 143-159
- [7] Hallem S, Chelf B, Xie Y, Engler D. A system and language for building system-specific, static analyses//Proceedings of the 2002 ACM Conference on Programming Language Design and Implementation. Berlin, Germany, 2002: 69-82
- [8] Cytron R, Ferrante J, Rosen B, Wegman M, Zadeck K. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, 1991, 13(4): 451-490
- [9] Yang J, Kremenek T, Xie Y, Engler D. MECA: An extensible, expressive system and language for statically checking security properties//Proceedings of the 2003 ACM Conference on Computer and Communication Security. Washington, DC, USA, 2003: 321-334
- [10] Shankar U, Talwar K, Foster J, Wagner D. Detecting format string vulnerabilities with type qualifiers//Proceedings of the 2004 USENIX Security Symposium. Washington, DC, USA, 2004: 201-220
- [11] Volanschi N. A portable compiler-integrated approach to permanent checking//Proceedings of the 2006 IEEE International Conference on Automated Software Engineering. Washington, DC, USA, 2006: 103-112
- [12] Chen H, Wagner D. MOPS: An infrastructure for examining security properties of software//Proceedings of the 2002 ACM conference on Computer and Communications Security. Washington, DC, USA, 2002: 235-244



LIANG Bin, born in 1973, Ph. D., associate professor. His research interests include information security and system software.

HOU Kan-Kan, born in 1986, M. S. candidate. Her research interests focus on static analysis.

SHI Wen-Chang, born in 1964, Ph. D., professor, Ph.D. supervisor. His research interests include information security, system software, and trusted computing.

LIANG Zhao-Hui, born in 1968, Ph. D., lecturer. Her research interests include information security and network communication.

Background

In recent years, researchers have applied static analysis techniques to security vulnerabilities detection. Taint-based analysis is the mainstream static detection approach. Based on the approach, static detectors traverse the control flows of target program and apply program variables state transition according to some vulnerability state machines. Some static vulnerability detection systems adopt above approach have been developed and applied to complex software systems. Many exploitable vulnerabilities have been found by these systems. However, all of them are reported to have comparatively high false positive and false negative rates. Users must refine detection results by manual code auditing. Even worse, some serious vulnerabilities may be neglected due to false negative. One main reason is lack of accurate and effective identification and analysis of security-related program elements in existing methods and systems, e. g. data validation checking, tainted data source, etc.

In this paper, a static vulnerability detection method based on data security state tracing and checking is proposed.

The state space of state machine model is extended to support accurate recognition of program security-related behaviors to reduce false positives. A systematic discrimination mechanism for tainted data is constructed to prevent false negatives result from neglecting tainted data sources. The experimental results of a prototype system show that this method can effectively detect buffer overflow and other type's vulnerabilities in software systems, and with obviously lower false positive than existing mainstream static detection methods and avoid some serious false negative of these methods.

The research is supported by the National Natural Science Foundation of China under grant No.60703102 and No.60873213, Beijing Natural Science Foundation under grant No.4082018 and National High Technology Research and Development Program (863 Program) of China under grant No.2007AA01Z414. All the granted projects aim to improve the security of software with static analysis and trusted computing techniques.