基于Mozilla的安全性漏洞再修复经验研究

张 凯 孙小兵 彭 鑫 赵文耘
(复旦大学软件学院 上海 201203) (复旦大学上海市数据科学重点实验室 上海 201203)

摘 要 相较于其他类型的漏洞,安全性漏洞更容易发生再修复,这使得安全性漏洞需要更多的开发资源,从而增加了这些安全性漏洞修复的成本。因此,减少安全性漏洞再修复的发生的重要性不言而喻。对安全性漏洞再修复的经验研究有助于减少再修复的发生。首先,通过对Mozilla工程中一些发生再修复的安全性漏洞的安全性漏洞类型、发生再修复的原因、再修复的次数、修改的提交数、修改的文件数、修改的代码行数的增减、初始修复和再修复的对比等数据进行分析,发现了安全性漏洞发生再修复是普遍存在的,且与漏洞发生原因的识别的复杂程度和漏洞修复的复杂程度这两个因素有关;其次,初始修复涉及的文件、代码的集中程度是影响再修复的原因之一,而使用更复杂、更有效的修复过程可有效避免再修复的发生;最后,总结了几种安全性漏洞发生再修复的原因,使开发人员有效地识别不同类型的安全性漏洞再修复。

关键词 安全性漏洞,再修复,漏洞修复,经验研究
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.11.007

Empirical Study of Reopened Security Bugs on Mozilla

ZHANG Kai SUN Xiao-bing PENG Xin ZHAO Wen-yun
(School of Software, Fudan University, Shanghai 201203, China)
(Shanghai Key Laboratory of Data Science, Fudan University, Shanghai 201203, China)

Abstract Compared to other types of bugs, security bug reopens more often, moreover, they need more development resources to fix it, which adds an extra cost to fix them. Hence, the empirical study of reopened security bugs is important. Our study collected the reopened security bugs from the Mozilla project, and analyzed them from the times of their reopening and commits, files which were modified to fix them, lines of added and deleted code, and comparison of the original fixing and reopened fixing. The empirical results show that security bug reopening often happen and it relates to the complexity of recognizing the reason that a security bug happens and fixing bugs. In addition, the locality of the files and code in the original security bug fixing is one of the causes to influence its re-fixing for bug reopens, and using more complex and effective fixing process can help reduce the security bug reopens. Finally, we summarized several causes for security bug reopens to help developers more easily identify the reopens of different types of security bugs.

Keywords Security bug, Reopens, Bug fixing, Empirical study

1 引言

在软件开发和维护过程中,开发人员常常遇到程序运行状态与预期不符的情况,如输出值不正确、程序死机、数据丢失等,这种现象来源于程序的漏洞(bug)。开发人员如果在程序中发现漏洞的存在,则必定会停止开发工作来修复漏洞,或者投入专门的人力和物力进行漏洞修复。不论是哪种应对方案,都会不可避免地消耗开发资源,使开发成本变得更加昂贵,直接导致开发软件的利润变得更小,软件竞争力下降。

在众多漏洞类型中,安全性漏洞是软件开发者在软件维护与更新的过程中重要的关注点之一[1-3]。安全性漏洞是指能被攻击者利用,以在计算机系统中获取未经授权的访问或进行超出权限的行为的一种软件漏洞,会导致安全性缺陷,使计算机系统陷入危险之中。安全性漏洞不一定会导致程序崩溃或停止运行,但其一旦被攻击者利用,则可能会造成不可挽回的损失。

有时,漏洞的修复不能一蹴而就,开发人员也不可避免地会发生修复漏洞错误的情况。开发人员完成漏洞修复之后会关闭漏洞,但如果发现这个漏洞未被完全修复,为了使之修复正确,需要重新打开(reopen)该漏洞来对其进行再修复,以避免在开发人员自认为修复正确的情况下,安全性漏洞却悄悄造成重大的损失。Zaman等人[4]发现,安全性漏洞相比于其他漏洞更常发生再修复。因此,对安全性漏洞的再修复进行研究,找出再修复与初次修复的隐含关系,对于安全性漏洞再

到稿日期:2016-10-07 返修日期:2016-12-17 本文受国家自然科学基金(61402396,61370079),中国博士后科学基金(2015M571489)资助。
张 凯(1993-),男,硕士生,主要研究方向为软件维护;孙小兵(1985-),男,博士,主要研究方向为软件分析、维护与演化;彭 鑫(1979-),男,博士,教授,CCF高级会员,主要研究方向为需求工程、自适应软件、软件产品线、软件维护、逆向工程,E-mail:pengxin@fudan.edu.cn;赵文耘(1964-),男,硕士,教授,CCF高级会员,主要研究方向为软件复用、软件产品线、软件工程。

修复的预测和提供修复意见十分重要。

本文主要通过对 Mozilla 工程中发生再修复的安全性漏洞的各项数据进行统计分析，找到安全性漏洞在再修复时存在的各种修复规律，如修改文件和代码较为分散的安全性漏洞较容易发生再修复，一些类型的安全性漏洞再修复会包含初始修复的修改模式、修复过程和修改的文件与代码。这些规律可为开发安全性漏洞再修复预测和自动化修复的工具提供经验支持。

本文第 2 节介绍经验研究的一些背景知识；第 3 节介绍研究的设计，包括数据来源、研究的数据和数据分类方式；第 4 节给出对数据分析的结果以及得到的规律；第 5 节介绍相关工作；最后总结全文。

2 背景知识

2.1 漏洞生命周期

一个漏洞通常遵循图 1[5] 所示的生命周期。

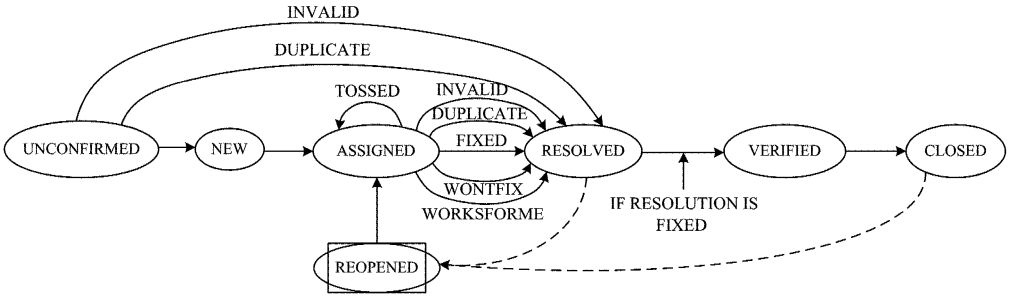


图 1 漏洞生命周期

一个漏洞在被发现后会被发现者报告出来，此时漏洞发生的根本原因、漏洞类型等信息还不确定，这便是未确认(UNCONFIRMED)阶段。下一阶段就是通过错误信息对漏洞发生的原因进行分析，如果是新的漏洞，就进入这个新漏洞的各种信息获取阶段(NEW and ASSIGNED)；如果与已有漏洞相同，就标注为重复(DUPLICATE)。获取完漏洞信息之后，即对漏洞进行修复，修复完成则进入解决(RESOLVED)阶段。解决阶段之后就是对修复进行验证(VERIFIED)的阶段，该阶段主要负责验证修复是否正确，最后关闭(CLOSED)漏洞。

在解决阶段和关闭阶段之后，都有可能因为修复不正确或不完整需要重新打开(REOPENED)漏洞来重新对其进行修复。这个重新打开阶段承担的工作就是本文要研究的安全性漏洞再修复。

2.2 安全性漏洞分类

OSVDB[1)] (Open Sourced Vulnerability Database)是一个为安全性漏洞提供准确、详细、及时、公正的信息的独立且开源的数据库。OSVDB 为安全性漏洞分类提供了参考。结合对数据的分析及 OSVDB 的参考分类，我们将研究中的安全性漏洞分为 7 类[6]：信息泄露、拒绝服务攻击、跨站脚本攻击(XSS)、缓存溢出、内存泄露、使用非法内存及其他。

信息泄露[2)] 是指在程序运行过程中，一些数据应该被保护或加密，未经授权的用户不能得到这些数据，但这些数据在某些安全性漏洞存在的情况下会被恶意用户获取，从而给系统或正常用户造成损失。

拒绝服务攻击[3)] 是指使目标电脑的网络或资源过载或耗尽，从而使服务暂时中断或停止，导致其对客户不可用。

跨站脚本攻击[4)] 是代码注入的一种，指在网站应用程序中存在安全性漏洞恶意用户通过这些漏洞在网页中插入恶意代码，而当用户进入、使用该网站时恶意代码会被执行。恶意代码包括 JavaScript，Java 等。恶意代码执行完之后，可能会获取更大的权限，窃取隐私信息等，给正常用户造成影响和损失。

程序在运行时都会分配一定大小的缓冲区，而当写入的内容超过缓冲区大小时，就会造成缓存溢出[5)]。若这种缓存溢出未被系统或程序察觉，则攻击者可能破坏程序的运行、获得程序甚至系统的控制权，进而运行恶意代码进行一些破坏性的操作。

内存泄露[6)] 是指在程序中一些已经使用过且不再需要的内存未被释放，使得程序的可用内存越来越少，最后导致程序因缺少内存而使部分功能被迫停止，甚至程序崩溃。这种内存的减少并不是物理上的内存减少，而是可分配使用的内存数量越来越少。

程序中经常使用指针来储存和操作数据，但如果这些指针是空指针、指向地址已经被释放或指向程序堆栈之外即不属于程序的内存地址，则会使程序停止运行或崩溃，这就是使用非法内存[7)]。

通过这些安全性漏洞分类，可针对每类安全性漏洞进行分析，研究各类安全性漏洞自身的特点。

2.3 安全性漏洞的修改模式

安全性问题通常是因为攻击者非法对数据进行访问而引

1) https://blog.osvdb.org

2) https://en.wikipedia.org/wiki/Information_leakage

3) https://en.wikipedia.org/wiki/Denial-of-service_attack

4) https://en.wikipedia.org/wiki/Cross-site_scripting

5) https://en.wikipedia.org/wiki/Buffer_overflow

6) https://en.wikipedia.org/wiki/Memory_leak

7) https://en.wikipedia.org/wiki/Memory_corruption

起的。在对安全性漏洞的分析中,安全性漏洞的修复大多并不复杂,根据程序系统对数据处理的 3 个阶段(数据的预处理、数据处理、数据确保)对安全性漏洞的修复,总结了如表 1 所列的一些修改模式。

表 1　安全性漏洞的修改模式

数据处理阶段	修改模式	详细描述
数据预处理阶段	数据检查	添加 IF 语句;使用 assertion 语句
	数据筛选	添加 IF 语句;添加循环模块
	加强数据限制	增强 IF 条件;将某段程序移入 IF 模块
	减弱数据限制	减弱 IF 条件;将某段程序移出 IF 模块
数据处理阶段	添加判断模块	添加 IF 或 SWITCH 模块
	父类变子类	将变量从父类变为子类
	子类变父类	将变量从子类变为父类
	TryCatch	添加 TryCatch 模块
	弱变强	将变量从弱引用变为强引用
数据确保阶段	数据确保	添加 IF 模块;添加 ENSURE 模块

在数据进入程序阶段,即数据预处理阶段,通常会对数据做一些检测和筛选,来防止外部数据对程序进行攻击。

在数据处理阶段,即程序对数据进行操作的阶段,需要保证数据被正确、安全地处理。

在数据被程序处理完成后,系统通常会根据这些数据输出一些我们想要的结果,而这些结果也可能因为处理过程的不理想而出现一些错误。针对这种情况,需要进行数据确保工作,以防止错误的处理结果对程序造成影响。在数据确保阶段,最常用的修改模式是添加 IF 模块,对结果进行判断,确认其为正确合法的结果。

安全性漏洞的修改模式相较于其他类型的漏洞更为简单,通常是针对数据的修改。安全性漏洞的修改模式不涉及对软件功能性的修改,只是对可能产生安全性问题的数据在程序传播的各个阶段进行检查、确认,针对不同的安全问题与传播阶段添加相应的防护机制,使数据的使用变得更加安全。同时,安全性漏洞的修改模式更具通用性,能够将相同的修改模式应用于不同的安全性漏洞的修复,而非安全性漏洞则难以总结出一些统一的修改模式,只能针对不同的功能模块得出不同的修改方法。

针对以上修改模式,可以观察安全性漏洞再修复与初始修复使用的修改模式之间的区别和联系。

3　研究设计

本文的研究主要围绕以下几个问题展开:

(1)安全性漏洞再修复的发生是否频繁,以及与什么因素有关?

(2)哪种类型的安全性漏洞容易发生再修复且修复的难度较大?

(3)初始修复与再修复在修改模式、修复过程、修改内容方面有什么联系?

3.1　数据来源

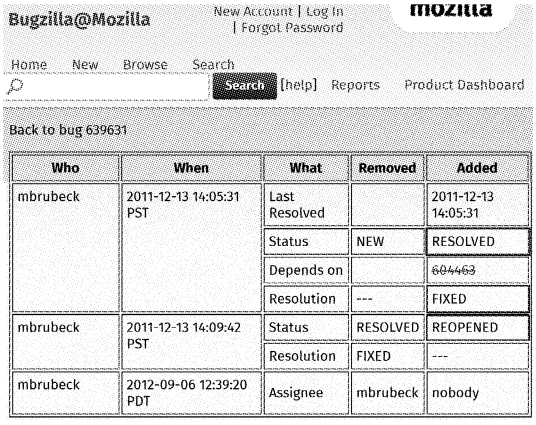
本文选定 Mozilla 作为研究安全性漏洞再修复的数据来源,它维护的全部都是 Mozilla 产品的安全性漏洞,且信息全面、清晰,还能在 git[1)] 上找到对应的提交信息,使得研究数据丰富且全面。

在对安全性漏洞的修复工作中,Mozilla 维护了一个安全性漏洞管理的社区,管理了 2005 年至今 Mozilla 发现的所有安全性漏洞的相关信息。

漏洞报告中包含许多信息,其中对研究有用的是以下几类信息:漏洞 id、漏洞描述、漏洞修改历史记录、漏洞修复评论。

在漏洞修复评论中,可以了解到修复人员的思考过程,从中得出该安全性漏洞产生的原因、修复时对错误信息的分析、使用该种修复方式的原因、对漏洞再修复的原因、修复完成后进行了何种测试等信息。通过对这些信息进行分析,明确了安全性漏洞修复的全过程,从而得出安全性漏洞的类型、漏洞再修复的原因、漏洞每次修复使用的修复方式及原因、每次修复之间的关系等。

漏洞修改历史记录(见图 2)中包含在每个时间点该漏洞处于何种状态的信息,图 2 中表的第 5 列出现“REOPEN”表示该安全性漏洞被再修复,而在接下来的状态中第 5 列出现“FIXED”则表示该再修复完成。


Bugzilla@Mozilla　New Account | Log In | Forgot Password　mozilla

Home　New　Browse　Search　Search　[help]　Reports　Product Dashboard

Back to bug 639631

Who	When	What	Removed	Added
mbrubeck	2011-12-13 14:05:31 PST	Last Resolved		2011-12-13 14:05:31
		Status	NEW	RESOLVED
		Depends on		~~604463~~
		Resolution	---	FIXED
mbrubeck	2011-12-13 14:09:42 PST	Status	RESOLVED	REOPENED
		Resolution	FIXED	---
mbrubeck	2012-09-06 12:39:20 PDT	Assignee	mbrubeck	nobody

图 2　bug 修改历史

对安全性漏洞再修复进行分析时,每次修复时代码的提交信息是必不可少的。Mozilla 将他们的项目提交到 git 上,通过 git 可以得到 Mozilla 工程每次提交的漏洞 id、代码、提交者、提交时间、修改的文件、本次提交与上次提交的代码更改信息、修改相关信息等。但是,git 上的 Mozilla 工程只持续到 2011 年,因此能得到的提交信息年份为 2005－2011。通过 Mozilla 安全性漏洞社区和 git 项目数据,总共获取到 48 个可用的发生过再修复的安全性漏洞的相关数据,这些安全性漏洞的 id 范围为 312278－672485。

3.2　研究内容

对这些安全性漏洞数据的分析主要包括 3 方面:漏洞的整体情况、初始修复和再修复的对比数据、修复过程中用到的数据。

漏洞的整体情况数据包括:漏洞的 id,漏洞的类型,发生再修复的原因,在 git 上这个漏洞的修复提交的总次数,所有

1) https://github.com/ehsan/mozilla-history.git

修改的文件数，所有修改增加和减少的代码行数，发生再修复的次数。这些数据有助于分析安全性漏洞的基本信息中隐含的修复规律。

初始修复与再修复的对比数据包括：两次修复时间的长短对比，两次修复使用的修改模式的对比，修改者的对比，两次修复使用的修复过程的对比，两次修复修改的文件和文件中修改内容的对比。通过这些修复的对比数据可以得到再修复和初始修复之间的关系，可通过初始修复推断再修复的修改模式、修改文件与内容、修复时需要注意的事项，从而提供更方便的再修复方法。

而通过修复过程的数据（如评论、提交的代码）对比，可以得到修复者的思路，并可从中找出安全性漏洞类型、再修复原因、修改模式、修复过程等。以上两方面数据，有些通过对修复过程中的数据进行统计和分析获得。

本文研究通过两种方式进行分类：安全性漏洞的类型和安全性漏洞再修复的原因。

根据安全性漏洞的分类方式，可以统计安全性漏洞再修复发生的次数、再修复的原因、这类安全性漏洞提交的总次数、总的文件修改数、增加的代码行数和删除的代码行数。在这些数据的支持下，对以下问题进行研究：安全性漏洞整体情况的分析；不同类型的安全性漏洞较容易发生哪种再修复；不同类型的安全性漏洞在修改时修改文件、修改的代码数量等有什么规律。

通过对安全性漏洞再修复的原因进行分类，收集初始修复与再修复两方面的数据：使用的修改模式，修复过程，修复所用的时间，修改者，两次修改对文件及代码的影响。通过收集到的数据，研究如下问题：安全性漏洞再修复的原因；再修复和初始修复使用的修复过程有什么联系；两次修复的修改时间和修改者有什么联系；两次修复使用的修改模式有什么联系；两次修复修改的文件和内容有什么联系。

4 数据分析

4.1 基础数据统计

实验共收集到 721 条安全性漏洞信息和 48 条发生再修复的安全性漏洞。在 Mozilla 工程中，安全性漏洞发生再修复的概率约为 6.7%，再修复的概率较高[4]。

安全性漏洞在所有 id 和时间段都有发生再修复，由此可见，安全性漏洞发生再修复是一个普遍存在且发生概率比较高的问题。通过对安全性漏洞再修复的信息进行分析，找到减少再修复、预测再修复、帮助再修复的方法，有助于提高程序的安全性，减少开发、维护的资源消耗，使得程序员在对安全性漏洞进行修复时更加全面、正确，减少对安全性漏洞再次修复的次数。

如表 2 所列，在 48 个发生再修复的安全性漏洞中，37 个发生了 1 次再修复，10 个发生了 2 次再修复，只有 1 个发生了 3 次再修复，未见超过 3 次的再修复。在修复安全性漏洞时，发生 1 次再修复可能是对漏洞的理解有遗漏；发生 2 次再修复就需要程序员特别注意自己的修复以及对漏洞的理解是否正确；唯一一次发生了 3 次再修复的漏洞是对漏洞进行登记和测试，因此通常情况下不会发生超过 2 次的再修复。多次发生再修复不仅会对资源造成浪费，还会加大系统的危险性，给攻击者更多机会。

表 2 安全性漏洞修复基础数据统计

类型	类别数量	再修复1次	再修复2次	再修复3次	git提交数	修改文件数	减少代码行	增加代码行
缓存溢出	12	8	4		3	3.7	45	208
内存泄露	3	2	1		3.7	2.7	59	111
非法内存	10	8	2		3.4	6.7	56.2	119.7
跨站脚本攻击	21	18	2	1	3.3	4.7	57	118
其他	2	1	1		5	1.5	6.5	17
总计	48	37	10	1				

发生 2 次再修复的安全性漏洞大多是缓存溢出，发生 1 次再修复的大多是跨站脚本攻击，其他的比例都比较接近。而这几种类型的安全性漏洞 git 提交的平均次数都比较接近，使用非法内存修改的文件数最多，内存泄露最少。在减少和增加的代码行数方面，内存泄露和缓存溢出分别为减少和增加代码行最多的，缓存溢出和内存泄露分别为减少和增加代码行最少的。

缓存溢出和内存泄露虽然涉及的文件数较少，但是修改的代码行数却较多，说明这两类安全性漏洞问题比较集中，修复时所涉及功能的关联性比较强。而使用非法内存和跨站脚本攻击涉及的文件较多，但是修改的代码行数较少，说明这两种安全性漏洞涉及的功能较为分散，修改的文件范围较广，修复时较容易有遗漏，考虑得不够全面。

4.2 安全性漏洞再修复的原因

在对数据进行分析后，得到以下几个发生再修复的原因：加强漏洞填补（范围加强、强度加强）；减弱漏洞的填补；填补原来没有考虑到的漏洞（线性关系、并列关系）；对修复进行登记或测试；以前修复无效或错误而重新修复；怀疑修复造成了错误，后来发现不是（误诊）。

实验中统计的 48 个发生了再修复的安全性漏洞中，每种类型的再修复数量如表 3 所列。

表 3 再修复类型统计

范围加强	强度加强	减弱填补	并列关系	线性关系	登记测试	无效错误	误诊
4	7	3	13	1	7	11	9

从统计数据可知，除了减弱漏洞的修复数量较少以外，其他再修复的原因数都比较平均，再修复时一般会修复得比较适度或未修复完全，修复过强的情况比较少。修复过强则表示安全性漏洞发生的原因分析正确，且修复方式也正确，只是限制过于强烈，避免这种再修复最好的方法是衡量好限制的强度与影响。而在填补原来未考虑到的漏洞中，线性关系的安全性漏洞也比较少，这种再修复的发生是对数据源头判断得不准确或未考虑数据的来源，修改比较简单，只要找出数据的源头并添加保护机制即可。上面的数据中有些类型再修复数据量较少，对发现普遍性规律有一定的影响，使得偶然性大大增加。

4.2.1　加强漏洞的填补

在修复安全性漏洞时,有时可能是临时性填补,安全性还不够高,还存在着一些安全隐患,后续需要对漏洞重新修复,从而提供更高的安全性防护;又或者可能是修复的范围不够广,同一个问题在不同的地方出现,此时只要将修复延伸至更大的范围即可。这些再修复就是对漏洞填补的加强。加强漏洞的填补可根据加强的再修复和原修复之间的关系分为强度加强和范围加强。

可以对安全性漏洞进行临时性填补是安全性漏洞修复的一个特点。临时填补是为了在没有时间和资源进行完全修复时,尽量减少安全性漏洞的作用力和危害,甚至用一些非常糟糕、后期难以识别维护的修复来修复漏洞。当然,这样做只是暂时使软件能继续运行,不至于因为软件停运造成损失,在有时间和资源时重新对漏洞进行修复,以保证修复的完整性与安全性,这就是加强漏洞修复的强度。

例如,对于安全性漏洞 329385[1],攻击者可以利用这个安全性漏洞进行强制性鼠标操作。初始修复时对 UI 进行修改,使得与这个漏洞相关的 UI 界面暂时停止相关功能。这样修复虽然可以解决问题,但是会造成一些功能无法使用,且真正能够强制操控鼠标操作的原因并没有找到。而在再修复中找到了真正的原因,并对其进行了修复,也使得 UI 的相关功能得以正常运行。

另一种对漏洞修复的加强是范围的加强,即对某个安全性漏洞的修复方式是正确的,但是修复作用的范围未完全包含所有出问题的地方,再修复时只需要将之前的修复作用于剩下的地方即可完成完整的修复。

例如,对于安全性漏洞 370127[2],其传递了某个不安全的方法作为新方法的父方法,而这个新方法如果被调用,则会由于父方法产生安全性问题。一开始的判断是 js 代码会调用到这个新方法,因此初始修复是在 js 调用该新方法时做一些安全性防护,以保证调用时不会传入不安全的父方法;后来却发现不但 js 代码会调用该新方法,C++代码某功能也会调用这个新方法,因此再修复要做的就是将原来的防护方法作用于 C++的调用,使得不论是哪种调用都能安全完成。

4.2.2　减弱漏洞的填补

与增强相反,减弱漏洞的填补也是安全性漏洞再修复的原因之一。在修复安全性漏洞时,如果对漏洞的防护太强,可能会导致一些功能被限制,一些合法数据被阻拦过滤,这样依旧会导致安全性问题。

例如,对于安全性漏洞 419848[3],js 通过导入一些脚本作为其内容来使用,这样可能会导入不安全的脚本,在系统权限下对程序造成危害。因此,初始修复就对 js 能导入的脚本进行了限制,使得某些 url 的脚本不能被导入。但这种限制太强,导致一些合法的脚本因含有相应的 url 而不能被导入,程序不能正常运行。再修复就是导入一些程序必须但不能获得系统权限的脚本,以保证程序安全地正常运行。

4.2.3　填补原来没有考虑到的漏洞

一个安全性漏洞的发生可能是因为多个地方有安全性问题,这些不同地方的安全性问题导致了同样的安全性漏洞。而在安全性漏洞的修复过程中,如果只修复了其中一部分安全性问题,另一部分未发现或被忽略,那么这个安全性漏洞就是修复不完整,因此会发生再修复。

因为填补原来没有考虑到的漏洞而发生的安全性漏洞再修复,可以根据具体的安全性问题之间的关系分为两种情况。

1)原修复和再修复是线性关系。这就好比一条河流发生洪水,如果在中流抗洪,并不一定能达到最好的效果,而如果找到发生洪涝的源头,在源头治洪,效果会好很多。

例如,安全性漏洞 312278[4] 发生的原因是某变量在未使用之前就被垃圾收集器释放了它所指向的内存地址,因此在后面使用该变量时就是使用非法内存。第一次修复是在发现内存被垃圾收集器释放的地方管理那个变量,使之不能被释放掉。但是后来发现这个变量被释放的源头不是这里,变量在被管理起来之前就有可能已被垃圾收集器释放。因此,再修复就是找到变量的源头,在源头对变量进行管理,使得变量不会被释放。

2)初始修复与再修复是并列关系。初始修复和再修复解决的安全性问题不是相同的方法或功能,它们的修复方式可能不一样,但是都会导致安全性漏洞。这些并列的安全性问题可能是因为难以全部找到或者经常容易被忽略,从而导致再修复的发生。

例如,安全性漏洞 360529[5] 是因为可以通过某些方法获得浏览器的权限来运行一些恶意代码,是跨站脚本攻击。针对这个安全性漏洞的修复,自然是找出这些方法并对它们进行保护,防止其被攻击者利用。初始修复是对 Array.prototype 和 document.write 进行保护,并认为成功修复了这个漏洞。但是后来发现 appendChild()和 removeAttribute()也会有这种问题,因此第二次修复就是对这些方法进行保护,但其实对每个方法进行保护的方式并不相同。

这种再修复原因与加强漏洞修复的范围的区别在于,用到的修复方式不同。加强漏洞修复范围中,初始修复和再修复运用的修复方法是一样的,只是作用的地方发生了变化;而填补原来没考虑到的漏洞(并列关系)则是初始修复和再修复运用的修复方式可能不同,针对的安全性问题也不同,但是这些安全性问题都会导致同一个安全性漏洞。

4.2.4　对修复进行登记或测试

有时重新打开安全性漏洞并不是因为原来的修复存在问

1) https://bugzilla.mozilla.org/show_bug.cgi?id=329385

2) https://bugzilla.mozilla.org/show_bug.cgi?id=370127

3) https://bugzilla.mozilla.org/show_bug.cgi?id=419848

4) https://bugzilla.mozilla.org/show_bug.cgi?id=312278

5) https://bugzilla.mozilla.org/show_bug.cgi?id=360529

6) https://bugzilla.mozilla.org/show_bug.cgi?id=410156

题，而是对完成的修复进行登记，比如验证、加入实际项目中；又或者是对其进行测试，以保证修复的安全性与正确性。这种再修复不涉及到原修复的改动，而是对原修复做一些操作。

例如，安全性漏洞 410156[6] 的初始修复就已经完全解决了安全隐患，重新打开漏洞是为了添加一些测试，使修复能在更多条件下被检验。

4.2.5 以前修复无效或错误而重新修复

上面发现的一些再修复都是初始修复有一定正确性，能起到一定作用，且再修复时不会完全推翻初始修复的修复方法。但是，有时候对安全性漏洞的修复并不那么简单，只要发生安全性漏洞的原因分析不正确或者修复方式错误，都有可能看似解决了问题，但实际上只是暂时的假象，问题不久又会再次发生或者还可能引起其他错误，因此要推翻初始修复进行重新修复。这种安全性漏洞的再修复一般会伴随着对初始修复的回退，然后用正确的修复替换掉初始修复。

例如，对于安全性漏洞 349527[1)]，初始修复完成之后发现有编译错误，还可能产生数据泄露的危险，因此须对初始修复进行改正，使因修复而产生的安全性问题都得到解决。

4.2.6 怀疑修复造成错误，后来发现不是

最后一种安全性漏洞再修复发生的原因其实是一种误诊。由于安全性漏洞的发生原因复杂且难以找寻，因此如果初始修复后程序产生了某些问题，开发人员可能会对这个修复产生怀疑，认为该修复造成了这些问题。于是，程序员会对这个修复进行回退，回退之后如果发现问题依旧存在，则说明并不是修复带来的问题。因此，这种原因产生的再修复一般是先对初始修复进行回退，然后发现不是修复存在问题后又将初始修复恢复。

4.3 安全性漏洞的类型与再修复的原因

通过统计安全性漏洞类型与再修复的原因，即每种类型对应的再修复原因的数量，可以得到哪种类型的安全性漏洞更容易发生哪种再修复。

通过表 4 所列统计结果得到如下规律：在 7 种安全性漏洞中，数据泄露和拒绝服务攻击没有发生再修复的情况。首先，拒绝服务攻击在安全性漏洞中本就较少，所以可能不会发生再修复；另一方面，也说明这两种安全性漏洞比较简单、易于修复或者修复方式是比较固定的，能够轻易找出所有要修复的问题并进行正确的修复，因此很少发生再修复。由此可见，在修复数据泄露和拒绝服务攻击的安全性漏洞时，如果能找到一个通用的发生漏洞的原因、修复漏洞的方式，就能使得这两种安全性漏洞基本不会发生再修复。

表 4 安全性漏洞类型与再修复原因统计

类型	数量	范围加强	强度加强	减弱填补	并列关系	线性关系	登记测试	无效错误	误诊
缓存溢出	12	1	2		2		2	2	4
内存泄露	3		1					2	
非法内存	10	2	2		4	1	1	1	3
XSS	21		2	3	6		4	6	1
其他	2	1			1				1

跨站脚本攻击发生再修复的次数最多，内存泄露发生的次数最少，其他的都比较平均。跨站脚本攻击发生再修复的原因中，剔除 4 个登记或测试和 1 个误诊，基本上每种再修复的原因都有涉及，说明跨站脚本攻击的安全性漏洞发生的原因比较复杂，因此很容易出现遗漏或错误，修复方式也难以做到全面保护，在修复这类安全性漏洞时需要投入更多的精力。特别地，仅有的 3 个减弱漏洞的填补的再修复都是跨站脚本攻击的漏洞，一方面说明减弱漏洞的填补这种再修复本身就很少，另一方面也说明了跨站脚本攻击安全性漏洞的修复方式较其他类型的安全性漏洞更容易发生修复过强的情况。在对跨站脚本攻击漏洞进行修复时，注意修复的强度与范围，防止其影响到其他功能，有助于减少这种再修复的发生。

内存泄露一般是因为内存使用和管理不当而产生的，所以在内存使用时必须进行分配与释放。内存泄露的安全性漏洞发生的原因并不复杂，修复方式也比较简单，同时数量较少。

由各种类型的安全性漏洞修改功能的集中度与各种再修复原因的分布范围可以看出，安全性漏洞发生再修复的可能性与两个因素有关：1）发生原因寻找的难度，找到发生的是哪种类型的安全性漏洞，为什么会发生这种安全性漏洞，发生错误的代码在什么文件、什么位置，这些信息越复杂，就越容易忽略或遗漏一些信息而导致再修复；2）修复方式的复杂程度，比如修改的文件的多少、这些修改文件之间的联系、修改的代码行数、修复的次数，修复方式越复杂，修复时越草率，发生再修复的可能性就越大。

4.4 初始修复与再修复的对比

4.4.1 修复过程的对比

根据表 5 统计的数据，基本上所有发生再修复的安全性漏洞在初始修复时都是一次修改和提交，使用的修复过程是“修复—测试”，而在误诊的初始修复中，使用了“修复—部分修复”和“修复—更好的修复”两种修复过程。这说明，仅仅使用“修复—测试”这种修复过程难以准确、全面地进行安全性防护，导致了再修复的发生。

表 5 修复过程的对比

再修复原因	初始修复		再修复	
	一次修改	多次修改	一次修改	多次修改
范围加强	4		4	
强度加强	7		5	2
减弱填补	3		3	
线性关系	1			1
并列关系	13		12	1
登记测试	7			
无效错误	11		8	3
误诊错误	7	2		

在对漏洞进行登记或测试的再修复中，基本只有一次修改和提交，因为再修复并不涉及代码的修改。再修复中除了使用“修复—测试”外，其他修复过程大多是以前修改错误或无效而重新修改，因为这种再修复是推翻以前的修复进行全

1) https://bugzilla.mozilla.org/show_bug.cgi?id=349527

新的修复,相当于修复一个新的安全性漏洞。通过使用这些修复过程来提高修复的安全性,可避免二次再修复的发生。

使用更复杂、更有效的修复过程有助于全面找到安全性漏洞发生的原因并进行更有效、更正确的修复,减小再修复发生的可能性。

4.4.2　修改时间和修改者的对比

根据表 6 统计的数据可知,除了以前修复无效或错误而重新修复的安全性漏洞,其他类型的再修复时间大多较初始修复时有所减少。这说明,初始修复只要不是完全无用,都对再修复有着参考性的作用,可以帮助寻找被遗漏的安全性问题,提供修复方式的参考。

表 6　修复时间与修复者的对比

再修复原因	再修复时间少	再修复时间多	修复者同一人	修复者非同一人
范围加强	4		4	
强度加强	5	2	4	3
减弱填补	2	1		3
线性关系	1		1	
并列关系	8	5	11	2
登记测试	7		2	5
无效错误	5	6	7	4
误诊错误	9		2	7

并列关系有一定数量的再修复时间更长的情况。并列关系其实是修复一些并列的安全性问题,且修复方式可能不同,导致安全性问题的原因也可能不同,因此需要更多的时间。

在修改者对比方面,除了减弱漏洞的填补、对修复进行登记或测试、误诊这 3 种再修复基本上都是非同一修复者外,其他的再修复是同一修复者的占大部分。这是因为如果同一个修复者进行再修复就不用重新熟悉该安全性漏洞,对漏洞发生的原因、原来的修复方式都有一定的了解,就可以有更高的再修复效率,从而节省时间与资源。

4.4.3　修改模式的对比

针对初始修复和再修复使用的修改模式进行的统计分析如表 7 所列,可以通过得出的规律对再修复进行修改模式的推荐。

表 7　修改模式的对比

再修复原因	范围加强	强度加强	减弱填补	线性关系	并列关系	登记测试	无效错误	误诊错误
包含	3	3	1		2		1	
相同	1	1	1	1	6		5	
不同		3	1		5		5	

对漏洞进行登记或测试和误诊这两种再修复基本上不涉及安全性修改,因此再修复时无修改模式的使用。

加强漏洞的填补中,范围加强再修复使用的修改模式基本包含了初始修复的修改模式,因为再修复只是将初始修复的使用范围变大,但修复方式不变。

在填补原来没有考虑到的漏洞中,线性关系的两次修改模式基本相同,因为两次修复方式完全相同,只是作用于不同的地方。

其他类型的再修复使用的修改模式与初始修复使用的可能相同也可能不同,因为再修复的修复方式与以前的不同,修改模式也可能不同。

4.4.4　文件修改情况的对比

安全性漏洞的修复体现在文件、代码的修改上。通过对比初始修复与再修复修改的文件及其内容之间的区别和关系来找出各种再修复修改时的特点。

根据对数据(见表 8)的分析,主要有 5 种文件修改的情况:相同文件,不同内容;相同文件,相似内容;不同文件,不同内容;不同文件,相似内容;无修改。无修改说明再修复没有对相关代码进行任何改动。相同文件,相似内容,说明初始修复与再修复修改的文件是相同的,对这些文件内容的修改也是相似的。相同文件,不同内容,则是修改文件相同,但是对文件的修改是不同的内容,对修复的作用也不同。剩下两种都修改的是不同的文件,但修改的内容和作用的区别与上面的定义相似。对修复进行登记或测试和误诊两种再修复不涉及代码修改。范围加强和线性关系则全是两次修复修改相同文件、相似内容,因为它们前后的修复方式相同,且安全性问题比较集中。并列关系主要集中在“相同文件,不同内容”和“不同文件,不同内容”两种情况,说明修复方式基本上是不同的。

表 8　文件修改情况的对比

再修复原因	相同文件相似内容	不同文件不同内容	相同文件不同内容	不同文件相似内容	无修改
范围加强	4				
强度加强	3	2	1	1	
减弱填补	2	1			
线性关系	1				
并列关系	1	6	5	1	
登记测试					7
无效错误	7	4			
误诊错误					9

5　相关工作

Pamela Bhattacharya 等人通过对漏洞报告和漏洞修复的经验研究,识别出漏洞报告的优劣性,找出了漏洞生命周期对漏洞修复的影响[7]。Michael Gegick 等人则通过对漏洞报告自然语言的描述进行自动化分析,能够找出其中的安全性漏洞报告,避免了浪费在寻找非安全性漏洞报告上的时间[8]。这些研究给我们漏洞报告的实验分析提供了研究方法的参考。

丁羽等人[9]通过对学术界关于漏洞分类学(Taxonomy)和漏洞分类(Classification)的调研,总结了安全性漏洞分类的几个关键性问题。Li Zhenmin 等人[10]使用自然语言处理分类工具对两个大型开源项目约 29000 个漏洞进行自动化分析,找到了漏洞的一些特性和规律,并发现了安全性漏洞数量正在增加且大部分会造成严重的损失。Yonghee Shin 等人[11]通过 9 种复杂性度量标准找到安全性漏洞与非安全性漏洞之间的区别,并对安全性漏洞进行预测。文章中提供了安全性漏洞分类和漏洞在各修复阶段的特点的参考[11]。Shahed Zaman 等人[4]针对 Firefox 工程中的漏洞,比较了安全性漏洞和功能性漏洞在各方面的不同之处。研究发现,安全性漏洞修复得越快,修复时需要的开发人员越多,涉及到的

文件也越多，再修复发生得越频繁[4]。这些研究都是对安全性漏洞的特征、分类进行了分析总结，而本文主要集中研究安全性漏洞再修复的修复特征，因此这些研究可为我们提供经验参考。

最后，还有一些预测方法的研究，这些预测方法与本文角度不同，他们一般是通过漏洞的各类静态或运行时信息来进行再修复的预测，而本文则主要是通过漏洞修复信息来寻找通用的预测方法。T Zimmermann 等人[12]通过开发人员对再修复原因的分类、漏洞报告、修复人员的关系等方面，利用静态分析模型对漏洞再修复的性质与预测方式进行研究，其中的分类与预测方法给本文研究提供了参考性建议。管铭等人[13]通过静态分析和动态分析两种程序分析技术，研究了在软件开发周期中安全性漏洞检测的技术。

到目前为止，对安全性漏洞的研究主要集中在以下方面：识别安全性漏洞[4,13-14]，预测安全性漏洞[8,15-16]，对安全性漏洞进行分类[9-10,17]，对安全性漏洞报告[8]、性质[18]进行研究与比较，安全模式的应用[19-20]，安全性检测[21-22]等，而对再修复的研究却寥寥无几，基本上是通过漏洞的特征、漏洞报告等信息对漏洞的再修复进行预测[12,23]。但是因为针对的是所有漏洞，安全性这个特点和其特殊性没有重点考虑，因此对安全性漏洞再修复知识的研究十分稀少。

结束语 本文通过对 Mozilla 工程中 2005 年－2011 年共 48 个发生了再修复的安全性漏洞的经验进行研究，分析安全性漏洞发生再修复的原因、再修复的次数、修改的提交数、修改的文件数、修改的代码行数的增减、初始修复和再修复的对比(使用的修改模式、使用的修复过程、修改的时间、修改者、修改文件内容)等数据，首先发现了安全性漏洞发生再修复是普遍存在的，且与漏洞发生原因的识别复杂程度和漏洞修复的复杂程度这两个因素有关；其次，了解了初始修复的文件、代码的集中程度是影响再修复的原因之一，并且使用更复杂、更有效的修复过程可有效避免再修复的发生；最后，总结了几种安全性漏洞发生再修复的原因。

本研究仍然存在一些未解决的问题：未能很好地识别安全性漏洞再修复与初始修复之间在修复方式上的区别，这对自动化再修复的研究有一定的限制。未来可以分析更新、更多的发生了再修复的安全性漏洞的数据，着重研究规律还不明确的几种再修复，尝试找出它们的预测方式，并通过再修复的历史数据找到前后两次修复在修复方式上存在的联系与区别，从而得到自动修复它们的方法，并开发出再修复预测与自动化修复的工具。

参 考 文 献

[1] TAN L, LIU C, LI Z M, et al. Bug characteristics in open source software[J]. Empirical Software Engineering, 2014, 19(6): 1665-1705.

[2] HALEY C B, LANEY R, MOFFETT J D, et al. Security Requirements Engineering: A Framework for Representation and Analysis[J]. IEEE Transactions on Software Engineering, 2008, 34(1): 133-153.

[3] VIEGA J, MCGRAW G. Building secure software: how to avoid security problems the right way[M]. Addison-Wesley, New York, 2001.

[4] ZAMAN S, ADAMS B, HASSAN A E. Security versus performance bugs: a case study on Firefox[C]// Proceedings of the 8th Working Conference on Mining Software Repositories. New York, NY, USA: ACM, 2011: 93-102.

[5] ZELLER A. Why Programs Fail: A Guide to System atic Debugging[M]. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[6] MCGRAW G. Software security: building security in[J]. IEEE Security & Privacy, 2006, 2(3): 6.

[7] BHATTACHARYA P, ULANOVA L, NEAMTIU I, et al. An Empirical Analysis of Bug Reports and Bug Fixing in Open Source Android Apps[C]// Proceedings of 17th European Conference on Software Maintenance & Reengineering. Washington DC, USA: IEEE, 2013: 133-143.

[8] GEGICK M, ROTELLA P, XIE T. Identifying security bug reports via text mining: An industrial case study[C]// Proceedings of the 7th International Working Conference on Mining Software Repositories. Washington DC, USA: IEEE, 2010: 11-20.

[9] DING Y, ZOU W, WEI T. Research summarize of classification of security bugs in software[C]// Proceedings of the 5th Conference on Vulnerability Analysis and Risk Assessment. 2012. (in Chinese)
丁羽，邹维，韦韬. 软件安全漏洞分类研究综述[C]// 信息安全漏洞分析与风险评估大会. 2012.

[10] LI Z M, TAN L, WANG X H, et al. Have things changed now? an empirical study of bug characteristics in modern open source software[C]// Proceedings of The Workshop on Architectural and System Support for Improving Software Dependability. Washington DC, USA: IEEE, 2010: 11-20.

[11] SHIN Y, WILLIAMS L. An Empirical Model to Predict Security Vulnerabilities using Code Complexity Metrics[C]// Proceedings of International Symposium on Empirical Software Engineering and Measurement. New York, NY, USA: ACM, 2008: 315-317.

[12] ZIMMERMANN T, NAGAPPAN N, GUO P, et al. Characterizing and predicting which bugs get reopened[C]// Proceedings of the 34th International Conference on Software Engineering. Washington DC, USA: IEEE, 2012: 1074-1083.

[13] GUAN M. The research of software security bug detection technology based on the analysis of application[D]. Xi'an: North-Western Polytechnical University, 2007. (in Chinese)
管铭. 基于程序分析的软件安全漏洞检测技术研究[D]. 西安：西北工业大学，2007.

[14] ZHANG L, ZENG Q K. The static detection technology of software security bug[J]. Software Engineering, 2008, 34(12): 157-159. (in Chinese)
张林，曾庆凯. 软件安全漏洞的静态检测技术[J]. 计算机工程，2008，34(12)：157-159.

[15] THOME J, SHAR L K, BRIAND L. Security slicing for auditing XML, XPath, and SQL injection vulnerabilities[C]// Proce-

dings of the 26th IEEE International Symposium on Software Reliability Engineering. Washington DC, USA: IEEE, 2015: 553-564.

[16] SHAR L K, TAN H B K, BRIAND L. Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis [C]// Proceedings of the 35th International Conference on Software Engineering. Washington DC, USA: IEEE, 2013, 8104: 642-651.

[17] LV W M, LIU J. The classification and analysis of the security bugs in C/C++ programs[J]. Computer Engineering and Applications, 2005, 41(5): 123-125. (in Chinese)
吕维梅,刘坚. C/C++程序安全漏洞的分类与分析[J]. 计算机工程与应用,2005,41(5):123-125.

[18] MA H T. The principles and defense methods of security bug in computer software[J]. Science & Technology Association Forum, 2009(6): 49. (in Chinese)
马海涛. 计算机软件安全漏洞原理及防范方法[J]. 科协论坛,2009(6):49.

[19] NGUYEN P H, YSKOUT K, HEYMAN T, et al. SoSPa: A system of Security design Patterns for systematically engineering secure systems[C]// Proceedings of the 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. Washington DC, USA: IEEE, 2015.

[20] YSKOUT K, SCANDARIATO R, JOOSEN W. Do Security Patterns Really Help Designers? [C] // Proceedings of the 37th IEEE/ACM International Conference on Software Engineering. Washington DC, USA: IEEE, 2015: 292-302.

[21] FELDERER M, ZEZH P, BREU R, et al. Model-based security testing: a taxonomy and systematic classification[J]. Software Testing Verification & Reliability, 2016, 26(2): 119-148.

[22] FELDERER M, BÜCHLER M, JOHNS M, et al. Security Testing: A Survey[M]// Advances in Computers. 2016: 1-51.

[23] XIA X, LO D, SHIHAB E, et al. Automatic, high accuracy prediction of reopened bugs[J]. Automated Software Engineering, 2015, 22(1): 75-109.

(上接第 26 页)

参 考 文 献

[1] GUARNIERI S, LIVSHITS V B. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code[C]// Proceedings of the 18th Conference on USENIX Security Symposium. New York, USA: ACM, 2009: 78-85.

[2] GUARNIERI S, PISTOIA M, TRIPP O, et al. Saving the world wide web from vulnerable JavaScript[C]// Proceedings of the 2011 International Symposium on Software Testing and Analysis. New York, USA: ACM, 2011: 177-187.

[3] GUHA A, KRISHAMURTHI S, JIM T. Using static analysis for Ajax intrusion detection[C]// Proceedings of the 18th International Conference on World Wide Web. New York, USA: ACM, 2009: 561-570.

[4] XU W, ZHANG F F, ZHU S C. The power of obfuscation techniques in malicious JavaScript code: A measurement study[C]// Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software. Washington DC, USA: IEEE, 2012: 9-16.

[5] RATANAWORABHAN P, LIVSHITS B, ZORN B G. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications[C]// Usenix Conference on Web Application Development. 2010.

[6] RICHARDS G, HAMMER C, BURG B, et al. The eval that men do[M]// ECOOP 2011-Object-Oriented Programming. Springer Berlin Heidelberg, 2011: 52-78.

[7] RICHARDS G, LEBRESNE S, BURG B, et al. An analysis of the dynamic behavior of JavaScript programs[J]. ACM SIGPLAN Notices, 2010, 45(6): 1-12.

[8] WEI S, RYDER B G. Practical blended taint analysis for JavaScript[C]// Proceedings of the 2013 International Symposium on Software Testing and Analysis. New York, USA: ACM, 2013: 336-346.

[9] CHUGH R, MEISTER J A, JHALA R, et al. Staged information flow for JavaScript[C]// Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, USA: ACM, 2009: 50-62.

[10] VOGT P, NENTWICH F, JOVANOVIC N, et al. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis[C]// The 14th Annual Network & Distributed System Security Symposium. Reston, USA: ISOC, 2007: 12.

[11] SCHÄFER M, SRIDHARAN M, DOLBY J, et al. Dynamic determinacy analysis[C]// Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, USA: ACM, 2013: 165-174.

[12] Google. ChromeV8[EB/OL]. [2016-07-07]. https://developers.google.com/v8.

[13] Adobe. Adobe PhoneGap[EB/OL]. [2016-07-07]. http://phonegap.com.

[14] CHUDNOV A, NAUMANN D A. Inlined information flow monitoring for JavaScript[C]// Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. New York, USA: ACM, 2015: 629-643.

[15] JANG D, JHALA R, LERNER S, et al. An empirical study of privacy-violating information flows in JavaScript Web applications[C]// Proceedings of the 17th ACM Conference on Computer and Communication Security. New York, USA: ACM, 2010: 270-283.