针对 Java 程序的一种脆弱性静态分析技术

A Static Analysis of Vulnerability for Java Program

匡春光　陈　华　张鲁峰　（北京系统工程研究所 北京 100101）

摘　要： 为了提高 Java 软件的安全性，针对 Java 程序中的脆弱性分析问题，提出了一种基于数据流的污染分析技术。其中包括对 Java 程序传播用户输入途径的介绍和基于数据流的污染分析技术的描述。依据此方法实现的分析系统能有效地分析出 Java 字节码程序中存在的 XPath 注入、SQL 注入等脆弱性，结果证明了基于数据流的污染分析技术的正确可行性。

关键词： 软件脆弱性 脆弱性分析 字节码 数据流 污染分析

1　引言

Java 语言由于其平台无关性而得到了越来越广泛的应用。相应地，Java 程序的安全性也越来越受到重视。Java 程序中存在脆弱性是引发 Java 程序安全问题的主要因素，应用脆弱性静态分析技术尽早发现 Java 程序中存在的脆弱性可以避免许多不必要的损失。

目前已有的脆弱性静态分析技术主要有语法模式匹配法[1]、利用类型限定符的污染分析法、模型检测法和自动定理证明法。语法模式匹配法的缺点是会产生大量虚报错误。利用类型限定符的污染分析法的最大缺陷是需要人工在源代码中增加与分析有关的注释，实用性不是很强。模型检测法存在状态爆炸问题，不太适用于大型程序。自动定理证明法的缺点是分析过程需要人工干预，自动化程度不高。

Java 程序中一些特定的脆弱性是由于程序接收了用户输入信息后，又利用这些信息直接或间接访问敏感系统资源（如 XML 文件中的数据、数据库中的数据、特权代码等）而产生的[2]。本文提出了一种基于数据流[3]的污染分析技术，能有效地分析出 Java 程序中存在的特定的脆弱性。

2　Java 程序传播用户输入的途径

Java 程序传播用户输入的途径分为方法内的途径和方法间的途径两类。方法内的传播途径主要包括局部变量赋值、域赋值、类型转换、数组元素赋值、集合（collection）操作。方法间的传播途径主要包括参数、返回值、域赋值。下面分小节介绍各种传播途径。

2.1　局部变量赋值的传播方式

局部变量赋值的传播方式是指用户输入被保存到一变量（如 input）后，应用程序使用此变量参与各种操作，操作结果用于对另一变量（如 media1）赋值，如果 input 中含有恶意信息，即 input 是被污染的，则 media1 也可能被污染，input 被污染的属性传播到了 media1。这种传播过程可能继续，即 media1 被污染的属性还可能传播到 media2，media3...，如果使用被污染的变量控制访问敏感系统资源，程序中就存在脆弱性。

2.2　方法内域赋值的传播方式

方法内域赋值的传播方式和局部变量赋值的传播方式是类似的。

2.3　类型转换的传播方式

类型转换的传播方式和局部变量赋值的传播方式也是类似的。

2.4　数组元素赋值的传播方式

数组元素赋值的传播方式要复杂一些。Java 对数组元素赋值时，要指定数组名和元素位置。如果数组中的某一元素（如 Arr[i]）被污染了，并不意味着其它元素也被污染了。当使用该数组中的元素（如 Arr[j]）时，如何判断 Arr[j] 是否被污染了呢？如果 j = i，则 Arr[j] 是被污染的，否则，Arr[j] 可能是未被污染的。对 Java 程序进行静态分析时，元素位置 i 和 j 的值有时是确定的，有时是不确定的。为了达到既不遗漏又尽量提高分析精度的目的，使用数组中的元素时可以使用如下判断原则判断该元素是否被污染了：

如果元素位置不确定,则数组中任一元素被污染,此元素被判断为被污染的。

如果数组元素位置确定,但数组中有位置不确定的元素被污染了,此元素被判断为被污染的。

如果数组元素位置确定,且能确定数组中该位置的元素被污染了,此元素被判断为被污染的。

如果数组元素位置确定,且能确定数组中该位置的元素未被污染,此元素被判断为未被污染的。

2.5 集合操作的传播方式

Java 的集合包括 List 和 Set。集合操作的传播方式和数组元素赋值的传播方式有相似之处,但其污染属性的传播要更不确定一些。使用集合中的元素时,只要集合中有任一元素被污染,被使用的元素就可能是被污染的。

2.6 参数的传播方式

Java 程序中的一个方法(如 method1)调用另一个方法(如 method2)时,如果 method1 中的某一被污染的变量或元素(如 var)作为 method2 的一个实参使用时,var 被传染的属性会传播给 method2 中相应的形参(如 par)。这就是参数的传播方式。Par 被传染的属性可能在 method2 内继续传播。为了跟踪参数的传播,在分析完 method1 后,还要接着分析 method2。在分析 method2 时,par 被当作最原始的污染源。

2.7 返回值的传播方式

Java 程序中的一个方法(如 method2)调用另一个方法(如 method3)时,如果 method3 具有被污染的返回值(如 retVal),如果 method2 接收了该返回值,并把它保存到一变量或元素(如 receiveVal)中,则 retVal 被污染的属性会传播给 receiveVal。如果从最初的用户输入(如 input)开始跟踪这种传播,即先分析 method3 再分析 method2 会存在一个问题。Method3 被分析完后会得到一个结论,即 method3 的返回值是被污染的,但是谁调用了 method3 呢?即谁是上面提到的 method2 呢?在分析 method3 时这个问题是不好回答的。为了解决这个问题,本文提出了一种反向跟踪的方法。即先分析 method2 再分析 method3,分析时不从最初的用户输入 input 开始,而从应用程序访问敏感系统资源处(如 access)开始,分析对 access 赋值时使用了其它哪些变量或元素(如 media),这些变量或元素是否是被污染的。如果这些变量或元素接收了另一方法(如 method3)的返回值,就接着分析 method3。在分析 method3 时继续使用反向跟踪的方法。这一过程可能继续到 method1,method0...,直到找到最初的用户输入 input,或证明此处的 access 不可能被污染为止。

2.8 方法间域赋值的传播方式

尽管都是域赋值的传播方式,但方法间域赋值的传播方式要比方法内域赋值的传播方式复杂。域是属于整个类的,而不是属于类中的方法的。类中的任何一个方法都可以访问其所属类中的域,也可以访问其它类中的一些公共域。类实际上是一个全局变量。在一个 方法(如 method2)中使用一个域(如 field)时,field 有可能在另一个方法(如 method3)中被赋值。和返回值的传播方式类似,分析方法间域赋值的传播时需要使用反向跟踪的方法,先分析 method2 再分析 method3。和返回值的传播方式不同的是,分析返回值的传播时,被调用的方法 method3 是唯一的,分析方法间域赋值的传播时,对 field 赋值的方法 method3 有可能不唯一。正文内容,正文内容,正文内容,正文正文内容,正文内容,正文内容,正文内容,正文内容,正文。

3 基于数据流的污染分析技术

从上面传播途径的分析可知,方法内的污染分析是整个分析的基础。方法间的污染分析是对多个方法进行方法内污染分析的串接,其串接的依据是污染属性在方法间的传播,这一点在传播途径的分析中已阐述了,因此本节主要阐述方法内的基于数据流的污染分析技术。基于数据流的污染分析技术主要包括以下几部分:(1)构建程序的 CFG;(2)分析 CFG 中各个节点的支配节点集;(3)分析 CFG 中各个节点的最邻近支配节点;(4)分析 CFG 中各个节点的支配前沿;(5)在程序中插入必要的变量赋值;(6)生成程序的静态单赋值表示;(7)分析程序中是否存在特定的脆弱性。下面较详细的介绍各个部分。

3.1 构建程序的 CFG

要构建程序的 CFG,需要先把程序分成若干基本块,每一个基本块是 CFG 中的一个节点,如果控制可以从一个节点 A 流向另一个节点 B,则在 CFG 中增加一条从 A 到 B 的边,A 成为 B 的前驱节点,B 成为 A 的后

继节点[5]。

3.2 分析 CFG 中各个节点的支配节点集

CFG 中节点 n 的支配节点是指从程序的入口到 n 的所有路径都经过的节点,所有支配节点的集合就构成支配节点集。显然,n 是它自身的支配节点,n 的所有前驱节点的支配节点集的交集中的节点也是 n 的支配节点。

3.3 分析 CFG 中各个节点的最邻近支配节点

CFG 中节点 n 的最邻近支配节点 id 是指满足如下条件的节点,① id 是 n 的支配节点,② id≠n,③ 不存在任一节点 m,m≠id,m≠n,id 支配 m,m 支配 n。直观地说,id 就是所有支配节点中级别最低的节点。分析 id 的算法可以借助选择排序算法的思想,先设支配节点集中的第一个节点为临时的最邻近支配节点 t,然后把 t 和支配节点集中的其它节点 r 比较,如果 t 不是 r 的支配节点,则继续往后比较,如果 t 是 r 的支配节点,则使 t = r,继续往后比较,最后得到的 t 就是 n 的最邻近支配节点。

3.4 分析 CFG 中各个节点的支配前沿

节点 n 的支配前沿是一个节点集,节点集中的每一个节点 f 至少有一个前驱 p 被 n 支配,但 f 本身不被 n 支配,或者 f = n。实际上,n 的支配前沿可以由两部分组成,第一部分是 n 的部分后继节点,这些节点的最邻近支配节点不是 n,第二部分是 n 的最邻近被支配节点的支配前沿中的一部分节点,这些节点的最邻近支配节点不是 n。

3.5 在程序中插入必要的变量赋值

在进行正向跟踪时,需要进行‘赋值 – 使用’的查找。在进行反向跟踪时,需要进行‘使用 – 赋值’的查找。变量的一次使用可能不只一次赋值和其对应,例如在程序存在分支时,有两个以上的分支都对某一变量赋值了,在分支的汇合点使用了该变量,此处变量的使用就不只一次赋值和其对应。这样,查找过程会比较复杂。如果变量的任何一次使用都只有唯一一次赋值和其对应,查找过程会相对简单一些。为了达到变量的一次使用只有唯一的一次赋值和其对应的目的,对一个变量,应该在给它赋值的所有节点的支配前沿中增加赋值。因为增加赋值后,变量的赋值点可能又增加了,因此这一过程可能要反复多次。使用工作链的方法可以很大地提高算法的效率。工作链的方法如下所述:

① 把对该变量赋值的所有节点都存到工作链中;

② 如果工作链为空,算法结束,否则,从工作链中取出一个节点 n,如果某节点 f 是节点 n 的支配前沿中的节点,而且在节点 f 中还没有增加对该变量的赋值,则

(A)在节点 f 中增加对该变量的赋值,赋值时使用的参数个数与节点 f 的前驱节点的个数相同,参数名暂时与该变量同名;

(B)标记在节点 f 中已增加对该变量的赋值了;

(C)如果在增加赋值前节点 f 中没有对该变量的赋值,则把节点 f 存入工作链中;

③ 转②

3.6 对变量重命名,生成程序的静态单赋值表示

在程序中插入必要的变量赋值后,变量的一次使用只有唯一的一次赋值和其对应,但因变量的多次赋值使用的是同一变量名,这种使用 – 赋值关系不能直观地表示出来,为了直观地表示出这种使用 – 赋值关系,可以对变量重命名,对变量的每一次赋值都赋予一个新的名称,对应的使用点也赋予同样的名称。

3.7 对变量重命名,生成程序的静态单赋值表示

分析程序中是否存在特定的脆弱性是指分析程序中是否有用户的输入信息传播到访问敏感系统资源的代码处的情况。具体的分析方法在上一节中已阐述,在此不再赘述。

4 应用实例

依据此算法实现了一个软件系统 – – – 基于数据流的污染分析系统。应用该分析系统对一些实例进行了分析,这些实例包括 NIST(National Institute of Standards and Technology)的安全工具测试集、开源 Web 应用系统 BookStore 和 OWASP(Open Web Application Security Project)开发的一个开源工具 WebGoat 5.0。其中第一个实例中包括 1 例 SQL 注入,第二个实例中包括 14 例 SQL 注入,第三个实例中包括 20 例 SQL 注入和 1 例 Xpath 注入。基于数据流的污染分析系统能分析出第一个实例中的 1 例 SQL 注入,第二个实例中的 12 例 SQL 注入,第三个实例中的 9 例 SQL 注入和 1 例 Xpath 注入。

(下转第 89 页)

(上接第 96 页)

5 结束语

本文首先分析了 Java 程序传播用户输入的途径及跟踪方法,然后阐述了基于数据流的污染分析技术,即说明如何实现跟踪。依据此算法实现的软件在不需要人工干预的情况下能比较准确地判断被测 Java 字节码程序中是否存在特定的脆弱性,如 XPath 注入、SQL 注入,说明本算法是正确可行的。

参考文献

1 John Viega, J. T. Bloch, Tadayoshi Kohno, Gray McGraw. ITS4: A Static Vulnerability Scanner for C and C + + Code. http://www. cigital. com/its4/. 2000 - 02 - 01.

2 Cyrille Artho, Armin Biere. Applying Static Analysis to Large - scale, Multi - threaded Java Programs. http://fmv. jku. at/papers/ArthoBiere - ASWEC01. pdf. 2003 - 07 - 05.

3 Steven S. Muchnick. Advanced Compiler Design Implementation. . 北京: 机械工业出版社 2003.