二进制程序的动态符号化污点分析

朱正欣[1] 曾凡平[1,2] 黄心依[1]
(中国科学技术大学计算机科学与技术学院 合肥 230026)[1]
(安徽省计算与通讯软件重点实验室 合肥 230026)[2]

摘 要 污点分析技术常用于跟踪二进制程序的信息流及检测安全漏洞,通过程序的动态执行来检测程序中由测试用例触发的漏洞。它的误报率很低,但是漏报率较高。针对污点分析的这一问题,动态符号化污点分析方法对污点分析进行了改进,通过将污点分析符号化来降低漏报率。根据基于指令的污点传播来获得相关污点数据的信息,同时制定符号化的风险分析规则,通过检测污点信息是否违反风险规则来发现存在的风险。实验结果表明,该方法不仅具有污点分析低误报率的优点,而且克服了污点分析高漏报率的缺点。在污点分析过程中产生的漏洞、风险及相关污点信息还可用于指导测试用例的生成,提高测试效率并降低测试用例的冗余。

关键词 污点分析,符号化,漏洞检测,测试用例,数据跟踪
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2016.2.034

Dynamic Symbolic Taint Analysis of Binary Programs

ZHU Zheng-xin[1] ZENG Fan-ping[1,2] HUANG Xin-yi[1]
(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China)[1]
(Anhui Province Key Lab of Software in Computing and Communication, Hefei 230026, China)[2]

Abstract The dynamic taint analysis (DTA for short) technique is usually applied to track information flow and detect security vulnerabilities. It detects the vulnerabilities of program triggered by some test cases dynamically. Though its false positive rate is very low, its false negative rate is very high. Concerning this issue, the dynamic symbolic taint analysis (DSTA for short) is an enhancement to dynamic symbolic analysis, which symbolizes the taint analysis to reduce false negative rate. The technique collects taint information according to taint propagating based on instructs, and makes symbolic risk rule to find some potential vulnerabilities by detecting whether the taint information breaks some risk rules. The experimental results show that this method not only ensures the advantage of DTA's low false positive rate, but also reduces the disadvantage of DTA's high false negative rate. The information of vulnerabilities, risks and taint data can be applied to generate test cases, which improves the test efficiency and reduces the redundancy of test case.

Keywords Taint analysis, Symbolic, Vulnerability detecting, Test case, Data tracking

1 引言

动态污点分析(DTA)是一种常用的漏洞检测技术,它通过动态执行来实时检测程序是否非法使用了污点数据,从而发现程序中实际被触发的漏洞。然而,大部分测试用例并不一定会触发漏洞,在这种情况下即使存在某些漏洞,DTA也会忽略这些漏洞并且认为程序是安全的。如果这些未被检测出的漏洞被黑客利用,就可能造成无法估量的破坏。当前主要是利用模糊测试方法产生测试用例,以尽可能多地触发漏洞,流行的模糊测试框架有Peach[1]、SPIKE[2]。模糊测试普遍存在的问题是效率较低、测试用例的冗余度较大,并且基本不可能触发程序中所有的漏洞。

为了检测出那些被忽略的潜在漏洞,本文提出动态符号化污点分析方法(DSTA)。该方法将污点分析符号化,首先,动态地将来自外部的数据标识为污点源;然后,通过分析程序执行时的指令和操作数,按照污点传播策略跟踪和记录污点数据的传播路径,并标记该污点数据的相关信息;其次,定制不同类型漏洞的风险检测规则,当污点数据传播到敏感操作点时,根据前一阶段获得的相关信息进行分析,检测是否会违反风险规则,并根据污点变量是否触发漏洞或者违反风险规则将其分类为3个风险等级,分别为触发漏洞变量、风险变量、安全变量;最后给出风险报告,包括相关污点的等级以及漏洞和风险的信息。通过这种方式可以检测出因未被测试用例触发而导致被DTA忽略的漏洞。在程序执行过程中需要分析每个污染点,检测出风险后给出报告,直至程序运行结束。如果在此过程中检测到被实际触发的漏洞,程序将直接停止运行,并且报告该漏洞的相关信息。如何跟踪、收集污点变量的相关信息,并对不同类型漏洞进行不同分析,是本文需

到稿日期:2014-12-18 返修日期:2015-04-19
朱正欣(1990—),女,硕士生,主要研究领域为软件测试及漏洞检测;曾凡平(1967—),男,博士,副教授,主要研究领域为信息安全与软件测试,E-mail:billzeng@ustc.edu.cn;黄心依(1993—),女,硕士生,主要研究领域为信息安全。

要解决的一个问题。

由于是对二进制程序进行动态分析，而二进制程序缺乏相关的包含语义和语法的类型信息，因此 DTA 按字节的粒度来定义污点数据。而 DSTA 将根据指令的类型，在污点传播的过程中标记污点数据的长度信息(1,2,4,n 字节，其中 n 字节表示字符串的长度)，所以更确切地说，DSTA 标记的是污点变量而不是简单的单字节污点数据。如何在程序执行过程中获得污点变量的长度信息以及相关的堆栈信息，是本文需要解决的另一个问题。

2 相关工作

插装是一种常用的动态分析二进制程序的技术，它在现有的代码中插入新的代码，但是要保持原程序逻辑完整功能不变。它在指令级监视程序的执行过程，用于指令追踪、分析程序的执行信息和运行状态。基于动态二进制程序的插装平台 Pin[3] 和 Valgrind[4] 是目前较为流行的方法。Pin 提供了丰富的 API。API 对很多重要指令集的特质进行了抽象，并且可以将寄存器内容等相关的上下文信息当作参数注入代码中。对于被注入代码覆盖到的寄存器，Pin 会自动保存和恢复相应的数据，以保证原程序不受影响。Pin 采用动态编译技术实现动态插装，Pintool 以动态链接库的形式存在，将分析代码和被分析程序动态编译后放置到代码缓存，然后由 CPU 动态执行代码缓存里的代码。只要在 Pin 中运行应用程序并带上相应动态链接库，就可以完成代码插装注入的工作。因此，对于开发者来说完全不需要修改原程序的二进制代码，这对于原二进制代码来说是完全透明的。本文的系统正是基于 Pin 实现的。

污点分析技术被广泛应用于二进制程序漏洞检测的研究。它对运行程序中的污点数据进行信息流跟踪，从而发现不安全行为，在程序的一次执行过程中只能分析一条路径。比较有名的基于动态污点分析的工具有 taintcheck[5]、LIFT[6] 和 Dytan[7]。为了降低漏报率，可以通过产生畸形测试用例来尽可能多地触发漏洞，基于这个想法出现了结合动态污点分析和模糊测试技术的方法[8]。

符号执行[9,10]技术使用符号值代替具体的数据作为程序输入，在执行程序的过程中以符号计算代替普通真实的数值计算，用符号表达式表示变量，最终输出结果也是符号表达式，通常与约束求解技术结合使用。符号执行最大的问题就是空间爆炸问题，并且当用于二进制程序时，存在变量识别问题。现在越来越多的技术将符号执行与污点分析结合起来使用，以提高路径覆盖率，如 DTA++[11]、TaintScope[12] 以及 DsVD[13]。

针对当前污点分析的不足，本文提出了动态符号化污点分析的思想，并且实现了基于该思想的二进制程序漏洞检测工具。这里将其称为动态符号化污点分析，但这并不表示它是将污点分析与符号执行结合起来的一种技术，而是使用符号化的思想将污点信息及风险规则符号化，来分析污点数据在传播过程中是否违反某些风险规则，进而检测出程序在动态执行过程中的不安全行为。实验证明，该方法在保证污点分析低误报率的前提下，很好地解决了高漏报率的问题。

3 动态符号化污点分析技术的原理及系统设计

3.1 基本原理

本文提出了动态符号化污点分析技术，其原理如图 1 所示。

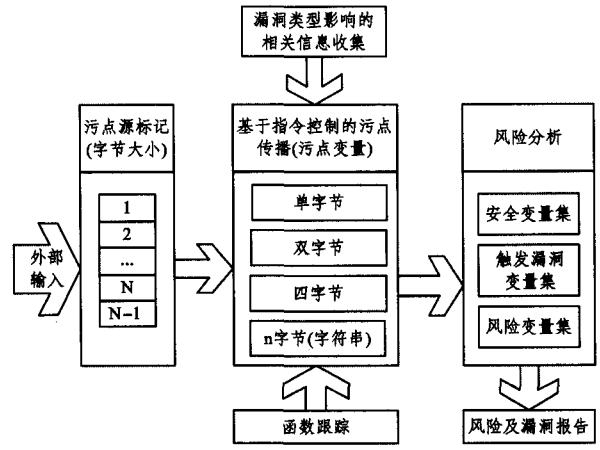

图 1 动态符号化污点分析技术原理

该技术分为以下 3 个部分：

(1)从外部获得输入，并且按照字节大小将其标记为污点源。

(2)一方面，在指令控制下进行污点变量的传播；另一方面，通过跟踪函数栈的变化来跟踪函数的执行，获得相应的函数信息，以及针对不同类型漏洞所需的信息进行收集(如对于堆溢出，需要跟踪堆的变化)，将其加入相关污点变量属性中。

(3)根据污点变量的属性及风险规则进行风险分析，按照污点变量的安全性将其加入相应的污点变量集中，并给出风险报告。

3.2 基于指令控制的污点传播

根据指令功能，可将指令分为：算术运算指令、逻辑运算指令、数据传送指令、移位操作指令、堆栈操作指令、字符串处理指令、输入输出指令及其它指令。指令操作数存于存储器或寄存器中，在 32 位操作系统中，操作数大小可分为 8 位、16 位或 32 位。因为需要对污点进行传播，所以要分析每个有关变量操作的指令，并根据操作数所在的位置及操作数的大小分别进行处理。有如下定义。

定义 1(污染标记图) 污染标记图(Tag)用于标记内存是否受污染，Tag(addr)为 0 表示 addr 位置的字节不受污染，否则表示该字节受污染。

定义 2(直接污染源) 若某污点变量直接决定另一个污点变量的污染情况，则前者称为后者的直接污染源。令污点源的直接污染源是它本身。

定义 3(污点变量属性) 污点变量属性包括基本属性和附加属性。基本属性(TD_Battr)用于标记污点变量的基本信息，包括：变量地址(addr)、变量大小(size)、直接污染源的长度(len)、影响该变量的污点源信息 (src)；附加属性(TD_Aattr)用于标记针对不同类型漏洞分析所需的信息，例如对于栈溢出漏洞，附加信息包括局部变量所在的函数栈的栈底地址(ebp)。

定义 4(污点变量属性链表) 污点变量属性链表用于存储不同大小的变量的属性。内存的污点变量属性链表分为：Mem_taint1、Mem_taint2、Mem_taint4、Mem_taintn(后缀 1、2、4、n 表示变量的大小)，其中 Mem_taintn 用于记录字符串

污点变量的属性；寄存器的污点变量属性存储在数组 Reg_taint 中，假设 32 位 EAX 寄存器的索引号为 n，则 EAX 的 4 个字节属性分别存储于 Reg_taint[$n*4$]、Reg_taint[$n*4+1$]、Reg_taint[$n*4+2$]、Reg_taint[$n*4+3$]。

下面介绍污点变量传播处理策略。

首先根据指令类型判断目的操作数的污染情况由什么决定，包括 3 种情况：清除目的操作数的污染情况、Tag(dst_addr)|=Tag(src_addr)、Tag(dst_addr)=Tag(src_addr)。有些指令可能有不止一个目的操作数，或者没有操作数，但是只要该指令改变了内存或者寄存器，则按照以上 3 种污染传播策略之一进行处理。然后判断 Tag(src_addr)!=0 是否成立，若成立，则修改目的污点变量的相关属性；否则清除目的操作数的污染或者不做处理。接下来，根据指令类型、操作数类型及操作数大小将目的污点变量加入到相应的污点变量属性链表中，具体的基于指令控制的污点变量传播策略流程如图 2 所示。

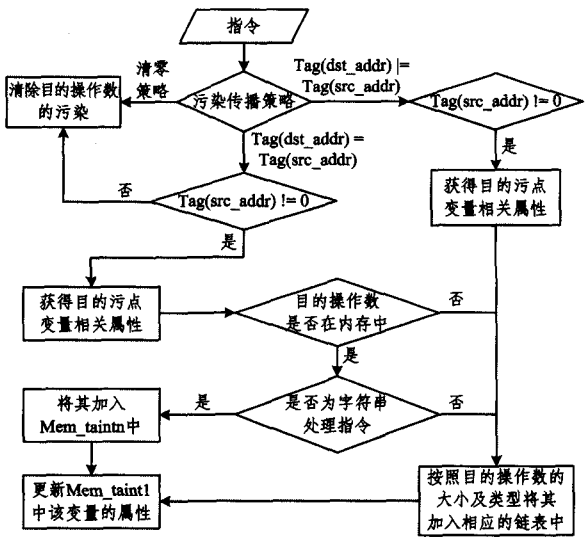

图 2　基于指令控制的污点传播策略

3.3　函数跟踪

为了获得风险函数的信息和堆栈信息，需要跟踪程序在执行过程中函数的执行顺序。通过函数的运行时堆栈之间的关系，实时记录该执行路径下正在运行的函数信息。有如下定义。

定义 5(函数执行链表)　函数执行链表(Rtn_list)用于跟踪记录程序在执行过程中的函数运行情况。函数基本信息包括：函数名(Func)、函数所属的镜像文件(Image)、函数地址(Address)、函数运行时栈基址(Start_stack)、函数发生时的次序(RtnCount)。

下面介绍函数跟踪策略。

在跟踪函数执行的顺序时，用一个全局变量 Count 计录该时刻已经运行的函数总数。每当开始执行一个新的函数时，则 Count++，生成一个新的函数节点指针(Node)，该函数的 Node->RtnCount=Count，通过 RtnCount 可得知函数之间的执行顺序，获得函数的基本信息。但是不能直接将该节点添加到链表头(头插法)，需要判断该函数与表头函数的堆栈关系。如果 Rtnlist == NULL || Node->start_stack<Rtnlist->start_stack，说明该函数是第一个执行的或者是被前面的函数调用的，就直接将其添加到链表头，即：Node->next=Rtnlist，并且 Rtnlist=Node；否则说明该函数的堆栈覆盖了前面函数的堆栈，即表明前面函数的堆栈已经失去作用，则删除链表中被覆盖的函数信息，并保存新的函数信息。通过这种方式产生的函数链表，跟踪记录了程序运行过程中函数的调用信息和堆栈信息。函数跟踪策略的流程如图 3 所示。

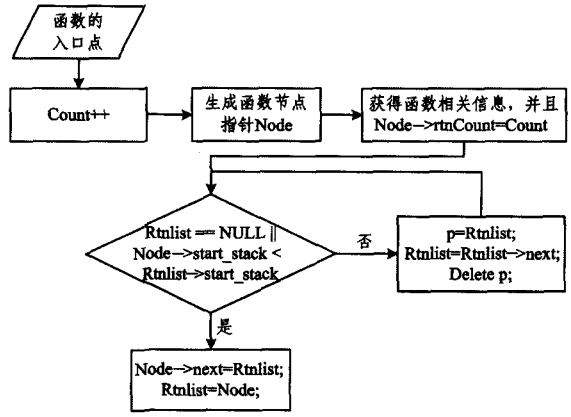

图 3　函数跟踪策略

3.4　漏洞类型影响的相关信息收集

该模块用于获得污点变量的附加属性，例如对于栈溢出漏洞检测，则需要获得该变量所在的函数栈基址 ebp。在标记一个新的污点变量的基本属性后，首先获得该时刻程序的 ESP 寄存器的值 esp，若 addr>esp，则说明该变量在函数栈中，然后通过顺序比较 addr 与此时函数执行链表中 start_stack 的大小，获得第一个比 addr 大的 start_stack，则获得了该变量所在的函数栈，并将函数的信息添加到变量的附加属性中；否则将 ebp 设置为 0，表示该变量不在函数栈中。

3.5　风险分析

在获得上述污点变量相关信息之后，需要对污点变量进行风险分析。下面将简要介绍风险分析的实现。有如下定义。

定义 6(污点变量集)　将污点变量集分为 3 类：安全变量集、触发漏洞变量集、风险变量集。将污点变量分为 3 个等级，便于进行漏洞分析。

下面介绍风险分析策略。不同的漏洞分析策略也会不一样，这里以栈溢出为例。

栈溢出是缓冲区溢出的一种，缓冲区溢出的实质是：如果程序员没有检查复制到缓冲区的输入数据，而当这个数据足够大时，将会溢出缓冲区的范围，从而改写其它的内存区域。如果向这些内存写入的是精心准备好的数据，就可能使得程序流程被劫持，致使不希望的代码被执行，落入攻击者的掌控之中，就会成为漏洞。栈溢出就是通过改写 EBP 和 RET 的内容，当函数返回时，输入的部分数据被加载到 EIP，从而控制程序的执行流程。

对于某个污点变量，首先获得其相关属性，包括它的地址(addr)、大小(size)、直接污染源的长度信息(len)以及它所在的栈基址(ebp)。那么该变量可用的最大安全范围为：addr-ebp，而其实际使用的范围为 size，且该大小由 len 控制。若在 len>size+1 的情况下程序运行正常且没有发生溢出，则说明该位置是安全的；否则不能保证该位置是安全的，说明该位置是存在风险的，通过这种方式可将污点变量分类加入污点变

量集中。风险分析的原理如图 4 所示。

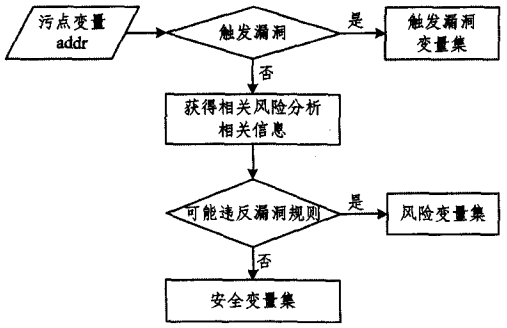

图 4　风险分析原理

对于不同类型的漏洞，其规则也会不同。我们只需要根据不同的规则，在程序执行过程中获得该规则所需要的信息，从而判断是否存在风险。

4　实验结果

基于动态插装平台 Pin[14] 以及 libdft[15]（libdft 是基于 Pin 实现的传统动态污点分析系统）的 API，开发实现了动态符号化污点分析的原型系统 *pintool-DSTA*。下面将以本系统在栈溢出的漏洞检测为例来说明 DSTA 思想的可行性及正确性。实验环境为：32 位 fedora-20 操作系统、gcc-4.8、pin-2.13。

4.1　信息提取分析

为了验证该系统是否获得了正确的污点信息，编写了一个简单的实验程序 test，且该程序存在明显的栈溢出错误，但是只要文件 in.txt 中的字符串长度小于 50（本例的 in.txt 中含有 20 个字符）就不会触发漏洞。实验程序如下：

```
#include<stdio.h>
int main()
{   FILE *fp;
    char s1[50];
    if ((fp=fopen("in.txt","r"))==NULL)
        printf("The file is not exist..\n");
    fgets(s1,100,fp);
    printf("The string is:%s\n",s1);
    return 0;
}
```

表 1 列出了本系统获得的两个字符串污点变量信息。

表 1　污点变量信息

addr	size	len	src	ebp	func
0xbfffef7b	0x14	0x14	0xb5275000	0xbfffefbc	main
0xb5265010	0x14	0x14	0xb5275000		

由表 1 可知，该系统获得了污点字符串的相关信息。其中 0xbfffef7b 是程序 main 中定义的 s1 字符串，它所在的函数为 main，函数栈基址为 0xbfffefbc，它的污点源是 0xb5275000，且字符串的大小和污染源的长度都是 0x14。那么另外一个污点变量 0xb5265010 是什么呢？因为程序中没有定义其它变量，其实它是在调用 printf 函数时产生的，且它的大小为 0x14，污点源也是 0xb5275000，它的污染源长度就是 s1 的大小（0x14）。

表 2 列出了在程序执行某一时刻的函数栈信息。

表 2　函数栈信息

Func	Image	Address	start_stack	RtnCount
_exit	libc.so.6	0xb5616360	0xbfffef6c	1451
-run_exit_handlers	libc.so.6	0xb558ecd0	0xbfffef9c	1426
-x86.get_pc_thunk.bx	libc.so.6	0xb5693bc5	0xbfffefb4	1425
exit	libc.so.6	0xb558edd0	0xbfffefbc	1424
-x86.get_pc_thunk.bx	libc.so.6	0xb5693bc5	0xbffff018	1208
-libc_start_main	libc.so.6	0xb5576870	0xbffff02c	1207
_start	test	0x80483c0	0xbffff050	1190

由表 2 可知该系统跟踪记录了程序在退出前的函数栈信息（最先发生的函数在最下面），最后发生的函数是 exit 退出函数（栈基址最小，次序最大）。

DTA 不能检测出 test 的漏洞，因为测试用例没有触发它，但是本系统则给出了正确的警告信息：[main]:(pid:)751d->(addr:)0xbfffef7b，可见在 main 函数中存在风险，且 0xbfffef7b 的污点变量属于风险变量集。

4.2　有效性测试

C 标准库提供了一些操作字符串的函数，但是如果使用不当便易造成缓冲区溢出。在使用不当时虽然存在溢出漏洞，但是有些测试用例并不一定能够触发它。为了验证本系统能够比传统污点分析技术更正确有效地检测出这些函数存在的潜在漏洞，分别以含有这些危险函数的示例程序在 3 种情况下（存在溢出漏洞，但是未被触发；存在溢出漏洞，并且被触发；不存在溢出漏洞）的 libdft 与 pintool-DSTA 实验对比结果进行说明。表 3 列出了在以上 3 种情况下，DTA（libdft）与 DSTA（pintool-DSTA）的报告情况（其中，“否”表示未发现漏洞，“是”表示发现漏洞并且报告漏洞位置）。

表 3　危险函数溢出漏洞检测

危险函数	存在溢出				不存在溢出	
	未触发漏洞		触发漏洞			
	DTA	DSTA	DTA	DSTA	DTA	DSTA
scanf	否	是	是	是	否	否
fscanf	否	是	是	是	否	否
fgets	否	是	是	是	否	否
sscanf	否	是	是	是	否	否
sprintf	否	是	是	是	否	否
vfsf	否	是	是	是	否	否
vspf	否	是	是	是	否	否
strcpy	否	是	是	是	否	否
strcat	否	是	是	是	否	否

由表 3 可知，DSTA 在保证 DTA 低误报率的前提下，能有效地发现被 DTA 忽略的情况，从而解决 DTA 高漏报率的问题，并且生成的风险报告能帮助我们针对性地生成测试用例以触发漏洞，在提高测试用例生成效率的同时，降低了其冗余度。

接下来，以已公布漏洞的实际程序为例，来验证本系统在实际漏洞检测中的可行性及实用性。分别用 DTA（libdft）和 DSTA（pintool-DSTA）对以下 4 个可执行程序进行漏洞检测（随机选取相同的一个测试用例），表 4 列出了二者的漏洞报告情况。

由表 4 可知，DSTA 在实际应用中具有较好的可用性与高效性，原因在于 DTA 检测漏洞的前提是该测试用例触发了某一个漏洞，而 DSTA 则不需要这个前提。只要该测试用例的执行路径经过了漏洞的位置，DSTA 便可以检测出来，因此相对来说，DSTA 较 DTA 的漏报率更低；并且，针对同样的

（下转第 187 页）

embedding fragile watermarking with flexible watermark payload[J]. Multimedia Tools and Applications, 2014, 721

[8] Wang Chuan-jian, Peng Yu-wei, Zhao Qing-zhan, et al. Fragile marking Scheme for Checking the Tamper of Geographical Data [J]. Mini-micro Systems, 2014, 35(12): 2628-2631(in Chinese)
汪传建,彭煜玮,赵庆展,等. 基于弱水印的地理数据篡改检验方法[J]. 小型微型计算机系统,2014,35(12):2628-2632

[9] Neyman S N, Sitohang B, Sutisna S. Reversible Fragile Watermarking based on Difference Expansion Using Manhattan Distances for 2D Vector Map[C]// The 4th International Conference on Electrical Engineering and Informatics(ICEEI 2013). 2013:614-620

[10] Wang Na-na, Men Chao-guang. Reversible fragile watermarking for 2-D vector map authentication with localization[J]. Computer-Aided Design. 2012, 44(4): 320-330

[11] Wang Na-na, Men Chao-guang. Reversible fragile watermarking for locating tampered blocks in 2D vector maps[J]. Multimedia Tools and Applications, 2013, 67(3): 709-739

(上接第 158 页)

漏洞,DSTA 只需生成该漏洞所在路径的某个测试用例,所以 DSTA 所需要的测试用例也相对更少。

表 4 可执行程序漏洞检测

漏洞编号	软件版本	DTA	DSTA	
			漏洞类型	漏洞位置
CNVD-2008-2394	WordNet 2.1	No Vulnerability	Buffer Overflow	searchwn()
CNVD-2009-5871	Xpdf 3.04	No Vulnerability	Buffer Overflow	FoFiType1::parse()
CNVD-2010-1209	Scite 2.26	No Vulnerability	Buffer Overflow	ReadLine()
CNVD-2014-01613	Freetype 2.5	No Vulnerability	Buffer Overflow	cf2_hintmap_build()

4.3 讨论

从以上实验结果可知,该系统不仅获得了正确的信息,同时可以检测出传统 DTA 无法检测出的风险,具有更大的实用性和高效性。但是该系统还存在以下不足:一方面,该系统只跟踪了数据流,而没有跟踪控制流,导致无法检测出由控制流产生的风险;另一方面,该系统建立在插装平台之上,导致检测程序时速度较慢。这些缺陷将在后续的工作中进行研究并加以解决。

结束语 本文介绍了动态符号化污点分析的思想,同时基于 Pin 插装平台实现了一个原型系统。DSTA 是对 DTA 的一种符号化处理及改进,并且取得了更好的效果:一方面它不仅可以直接检测出程序中实际发生的漏洞,同时也可以检测出在程序运行结束前该路径下潜在的风险情况;另一方面,该系统获得的风险变量集可以用于生成测试用例来验证该风险是否真的会触发漏洞,通过针对性地产生测试用例来提高测试效率,降低冗余。

未来的工作包括:完善该系统下的不同类型漏洞的风险检测;跟踪数据流的同时跟踪控制流,并且在跟踪控制流的同时获得该路径分支约束条件,以提高该系统漏洞检测的路径覆盖率;根据风险变量集的报告自动生成测试用例,实现系统的自动化;提高程序检测的速度。

参 考 文 献

[1] Peach [EB/OL]. http://peachfuzzer.com/. 2009 June

[2] SPIKE [OL]. http://www.immunitysec.com/resources-free-software.shtml

[3] Luk C K, Cohn R, Muth R, et al. Pin: building customized program analysis tools with dynamic instrumentation[J]. ACM Sigplan Notices, 2005, 40(6): 190-200

[4] Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation [J]. ACM Sigplan Notices, ACM, 2007, 42(6): 89-100

[5] Newsome D S J. Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software[C]// Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS). 2005

[6] Qin F, Wang C, Li Z, et al. Lift: A low-overhead practical information flow tracking system for detecting security attacks[C]// 39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006(MICRO-39). IEEE, 2006: 135-148

[7] Clause J, Li W, Orso A. Dytan: a generic dynamic taint analysis framework[C]// Proceedings of the 2007 international symposium on Software testing and analysis. ACM, 2007: 196-206

[8] Bekrar S, Bekrar C, Groz R, et al. A taint based approach for smart fuzzing[C]// 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2012: 818-825

[9] King J C. Symbolic execution and program testing[J]. Communications of the ACM, 1976, 19(7): 385-394

[10] Niu Wei-na, Ding Xue-feng, Liu Zhi, et al. Vulnerability Finding Using Symbolic Execution on binary program[J]. Computer Science, 2013, 40(10): 119-121, 138(in Chinese)
牛伟纳,丁雪峰,刘智,等. 基于符号执行的二进制代码漏洞发现[J]. 计算机科学,2013,40(10):119-121,138

[11] Kang M G, McCamant S, Poosankam P, et al. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation [OL]// www.cs.berkeley.edu/~dawnsong/papers/2011%20dat++-ndss11.pdf

[12] Wang T, Wei T, Gu G, et al. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection [C]// 2010 IEEE Symposium on Security and Privacy (SP). IEEE, 2010: 497-512

[13] Wang Z, Tang Z, Zhou K, et al. DsVD: An Effective Low-Overhead Dynamic Software Vulnerability Discoverer [C]// 2011 10th International Symposium on Autonomous Decentralized Systems (ISADS). IEEE, 2011: 372-377

[14] Pin[OL]. https://software.intel.com/en-us/articles/pintool

[15] Kemerlis V P, Portokalidis G, Jee K, et al. libdft: Practical dynamic data flow tracking for commodity systems[J]. ACM SIGPLAN Notices, ACM, 2012, 47(7): 121-132