

# 基于污点分析的二进制程序漏洞检测

董国良<sup>1,2</sup>, 臧 洌<sup>1</sup>, 李 航<sup>1</sup>, 甘 露<sup>1</sup>, 郭咏科<sup>1</sup>

(1. 南京航空航天大学 计算机科学与技术学院, 江苏 南京 211106;

2. 江南计算技术研究所, 江苏 无锡 214083)

**摘 要:**针对现有动态污点分析平台由于欠污染和过污染导致的准确度问题,研究并实现了一种面向二进制程序的动态污点分析方法。从污点标记、污点传播和污点检测三个方面对现有污点分析技术的准确率问题进行改进,扩展了污点标记状态空间与污点传播状态转换的行为实体,根据指令特征对 X86 架构指令进行分析和归类,设计了兼顾数据流传播策略与控制流传播策略的污点传播策略,扩充了关于间接污染、潜在漏洞、污点清除等污点传播规则,定义了新的污点检测安全规则与相应的处理方式,完善了污点检测处理方法。基于上述方法实现了改进的动态污点分析原型系统 ODDTA,对原型系统的实验结果表明,该方法可有效解决现有污点分析平台的漏报和误报问题,提升污点分析的准确率和执行效率。

**关键词:**动态污点分析;漏洞检测;污点标记;污点传播;污点检测

中图分类号:TP311

文献标识码:A

文章编号:1673-629X(2018)03-0137-06

doi:10.3969/j.issn.1673-629X.2018.03.029

## Vulnerability Detection of Binary Program Based on Dynamic Taint Analysis

DONG Guo-liang<sup>1,2</sup>, ZANG Lie<sup>1</sup>, LI Hang<sup>1</sup>, GAN Lu<sup>1</sup>, GUO Yong-ke<sup>1</sup>

(1. School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China;

2. Jiangnan Institute of Computing Technology, Wuxi 214083, China)

**Abstract:** In view of the accuracy caused by over-tainting and under-tainting in existing dynamic taint analysis platform, we study and implement a dynamic taint analysis method for binary program, which improves the accuracy of the existing taint analysis techniques from taint mark, taint propagation and taint detection. The state space of the taint marking and behavior entities of the taint propagation is extended. According to the characteristics of the instruction, the X86 architecture instruction is analyzed and classified. We design a complete taint propagation strategy considering both tainted data-flow propagation strategy and control-flow propagation strategy, increase the taint propagation rule about indirect taint, potential vulnerabilities, taint removal and so on, define the new taint detection rules and their corresponding treatment, and perfect the taint detection method. On the basis of above methods, we implement a binary oriented vulnerability detection prototype system, namely ODDTA. The tests show that the proposed method can effectively solve the problem of false negatives and false positives in the existing dynamic taint analysis platform, and improve the accuracy and efficiency of the taint analysis.

**Key words:** dynamic taint analysis; vulnerability detection; taint mark; taint propagation; taint detection

## 0 引 言

动态污点分析技术(dynamic taint analysis, DTA)是指对非信任来源的数据进行标记,追踪并记录其在程序执行中的传播过程,检测污点数据的非法使用,以达到获取关键位置与输入数据关联信息的分析方法<sup>[1]</sup>。在对攻击进行有效分析的同时获得程序的漏洞

所在,且错误率较低,实用性很强<sup>[2]</sup>,被广泛应用于信息安全验证、恶意代码分析<sup>[3]</sup>、隐私泄露分析、协议格式逆向分析<sup>[4]</sup>等领域<sup>[5]</sup>。国内近年来的研究成果包括李根博士团队研发的 Hunter<sup>[6]</sup>和北大王铁磊博士研究的 TaintScope<sup>[7]</sup>,以及基于类型<sup>[8]</sup>和基于虚拟化<sup>[9]</sup>技术的动态污点分析技术等。

收稿日期:2017-02-23

修回日期:2017-06-28

网络出版时间:2017-12-04

基金项目:国家重点研发计划“云计算和大数据”重点专项(2016YFB1000500)

作者简介:董国良(1980-),男,工程师,硕士研究生,研究方向为软件测试与网络安全;臧 洌,硕士,副教授,研究方向为网络安全及软件可靠性。

网络出版地址: <http://cnki.net/kcms/detail/61.1450.TP.20171204.1647.008.html>

目前的动态污点分析平台主要存在准确率与性能两方面的问题<sup>[10]</sup>,其中准确率问题主要体现在由于“过污染”(over-tainting)引起的误报,以及由于“欠污染”(under-tainting)造成的漏报问题。另外,早期的粗粒度污点分析<sup>[11]</sup>平台在污点标记属性、污点传播策略及污点检测规则定义过程中存在不够完善的情况,使得漏洞挖掘效率不高,检测结果不够准确。

针对上述问题,文中分别在污点分析的三个阶段对现有问题进行改进和优化。在污点标识过程中扩充污点状态定义,添加除“污染”和“未污染”外的第三种污点状态“间接污染”,细化污点标记过程中定义的污点状态属性;在污点传播过程中,针对 X86 平台汇编指令设计了较为完善的污点传播策略,增加“污点清除”传播状态行为实体,分析造成污点清除的目标指令和操作并将其添加入污点传播策略;在污点检测阶段,进一步扩充安全检测规则,将触发的安全规则按照危险严重性分为漏洞触发、危险操作和安全操作三个级别,并根据不同级别设计了不同的安全响应策略。基于上述方法实现了改进的细粒度动态污点分析<sup>[12]</sup>原型系统 ODDTA,并通过实验对其进行验证。

## 1 面向二进制漏洞挖掘的动态污点分析系统设计

### 1.1 系统框架设计

ODDTA 的框架结构如图 1 所示。

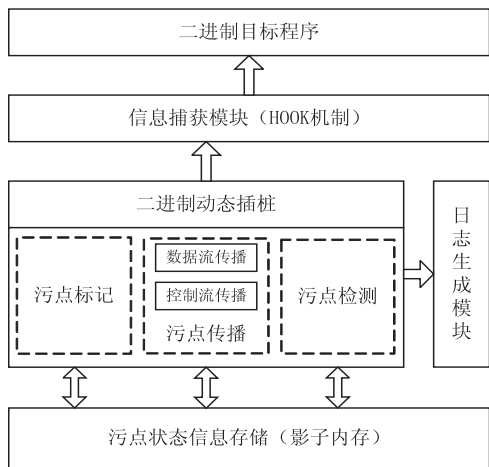


图 1 ODDTA 原型系统框架结构

ODDTA 原型系统主要包括信息捕获、二进制动态插桩、污点标记、污点传播、污点检测、污点信息存储及日志生成等模块。其中信息捕获模块主要使用 HOOK 机制对二进制目标程序的主要输入函数进行监控,获取目标程序的指令流。二进制动态插桩模块负责在程序运行动态编译二进制代码时,将插桩代码置于目标程序中,获取程序运行时的特征数据,其生成的指令为经过二方数据提升的类汇编指令。污点分析的核心

模块包括污点标记、污点传播和污点检测三个部分,所有细粒度的污点状态信息均以影子内存<sup>[13]</sup>形式存放于内存中,在污点分析整个过程中进行维护和存取。在线分析后产生的安全相关报告信息由日志生成模块生成,用于离线的漏洞检测与分析。

### 1.2 污点标记模块设计

污点标记是将外部引入的数据标记为污点数据,并对其设置污点标签,其标记方式极大影响着污点传播处理与污点状态信息存储的效率。但在现有的污点分析工具中,因为污点标记定义和设计的不够完善,不能检测出某些间接污染的情况,如以下代码所示程序:

```
1 void func(char * source,int length)
2 {
3 char buf [10];
4 strncpy(buf, source,length);
5 }
6 void main()
7 {
8 char src[256];
9 scanf("%s",src);
10 intlen=strlen(src);
11 func (src,len);
12 }
```

变量 len 的值由系统函数 strlen() 得出,并非直接通过赋值或算术逻辑运算得出,若 strlen() 中的参数包含污点数据,则 len 与污点数据相关。传统的污点分析只标记变量是被“污染的”或“未被污染的”,且在污点传播过程中,对“污染的”变量只考虑直接由算术运算指令或数据传送指令等直接传递的污点数据和污染链,所以不会将变量 len 标识为“污染的”,这种欠污染的污点标记和传播方式会导致漏报。

对此,文中提出了第三种污点标记状态“间接污染”,当检测到变量由某些参数包含污点数据的函数间接所得,则标记该变量为“间接污染”。采用统一的标记模型集中存储污点信息,对内存中的污点数据采用<Addr,Length>结构表示,其中 Addr 表示污点引入位置,Length 表示污点数据大小;寄存器中数据采用<REG,TaintTag>结构表示,其中 REG 表示寄存器名,TaintTag 表示是否被污染;EFLAGS 采用一个字节的长度表示,8 位分别代表 EFLAGS 的主要 8 个标志位。在程序中,系统将识别函数 strlen(),判断其包含参数为污点数据,进一步将变量 len 对应指令标记为“间接污染”,同时在污点传播策略和污点检测中增加对此类污点数据的处理。

### 1.3 污点传播模块设计

污点传播指根据二进制程序指令的特点,为不同类型的指令制定不同的传播策略,根据此策略追踪污

点数据的运行过程。污点传播分析不仅要关注污点数据的产生与引用位置,更重要的是找出污点传播路径与影响范围<sup>[14]</sup>。文中的动态污点分析过程即从信息流传播入手,研究执行时的二进制类汇编代码,对污点相关的数据流和控制流传播过程加以分析,制定针对不同类别指令类型的污点传播策略,确定污点传播行为实体(对污点属性的具体操作),并实时更新和维护影子内存中的污点状态属性。

1.3.1 污点传播指令分析

分析了 X86 体系架构各类指令的执行特性,结合污点数据信息流传播方式,在动态污点传播模块将指令归纳为以下几类:

数据转移类:包含 X86 汇编指令中的数据传送指令、算术指令、位操作指令、标志处理指令、串操作指令。数据转移类指令包含数据信息流传播,是数据流流入、数据流流出与参数地址的作用之集。

控制跳转类:包含控制信息流传播,其主要对象为

条件跳转指令,如控制转移指令的 Jcc 类。在程序运行中,运行路径的选择是由条件跳转指令判断转移的。

传播无关类:指执行不会引发数据信息流与控制信息流的传播的无关指令,例如 NOP、HLT、JMP、CALL 等。

1.3.2 污点传播策略设计

在污点传播策略设计过程中,除了针对上述几类指令进行分析外,还应考虑控制流污点传播、污点清除行为、间接污染与检测盲点<sup>[15]</sup>等情况下的污点传播过程。其中检测盲点问题是指由于无效用例导致漏洞未被触发时的潜在漏洞情况。如包含潜在缓冲区溢出漏洞的程序,只有当漏洞触发,且溢出的字符串覆盖并修改了返回值时(即改变了数据,又影响了控制流),传统的污点分析系统才将其确定为漏洞触发,而当输入字符串长度不足以造成缓冲区溢出,或者已经造成溢出但未修改到返回值,则此时漏洞被忽略,存在检测的盲点,造成漏报。污点分析传播策略如表 1 所示。

表 1 指令类别与对应的污点传播策略

指令类别	示例	传播行为	传播策略
数据转移	MOV EAX,ECX DIV EBX(隐式操作)	污点添加 污点传播 不传播	若源操作数包含污点数据,则执行后源与目的操作数均被污染
条件跳转	JZ label	污点传播 不传播	根据跳转条件间接影响污点传播,结合数据流分析
传播无关类	NOP、HLT 等	不传播	无污点信息流操作,不传播
间接污染(函数级)	strcpy( buf, source, len) 对应的 CALL 指令	污点传播 不传播	函数参数为污点数据,则函数结果受其间接污染
敏感函数(函数级)	scanf、sprintf、strcpy 等对应的 CALL 指令	污点传播 不传播	若参数中包含污点数据,则监控和记录数据状态信息和数据流传播信息
污点清除相关类	MOV EAX, IMM SUB EAX,EAX XOR EAX,EAX	污点清除	对转移类指令,若源操作数类型为立即数,目的操作数中包含污点数据,则执行污点清除 对 SUB、XOR 等指令,若源操作数与目的操作数包含污点数据且两者相同,则执行污点清除

(1)数据转移类。

数据转移类指令的传播行为包括污点添加、污点传递和不传播。针对包含显式操作数的指令,如 MOV、ADD 等指令,若源操作数包含污点数据,则执行后源与目的操作数均被污染。对包含隐式操作数(implicit operands)的指令,如指令 DIV EBX,其显式操作数(explicit operands)只有 EBX,但根据指令的执行语义,除数为 EBX,隐含的被除数为 EDX,商保存在 EAX 中,其余数保存在 EDX 中,所以该算术指令的执行同样引发数据信息流传播。

(2)条件跳转类(控制流传播分析)。

控制流操纵了程序的运行路径,确定了执行流程,其为主方向选择上间接影响了数据流的传播。为保证污点分析的精确性,将数据流与控制流结合分析,从路径转移和数据传播两方面确定污点关联信息。

在动态污点传播过程中,一次实际执行对应一条

固定的运行路径。文中进行的控制流传播分析基于程序控制流图(CFG),分析由指令跳转引发的控制转移过程,研究分支路径的跳转条件,获取执行路径的污点约束针对跳转类指令。在汇编指令中,跳转指令分为直接跳转指令(JMP)和条件跳转指令(JCC)。条件跳转分为两类,其中多数以标志位为判断,包括 JZ、JS、JC 等,也有部分非判断标志位的指令,包括 JCXZ、JECXZ 等,其以 CX、ECX 是否为零作为判断条件。文中提出的控制流分析的主要对象为标志位污点信息。

(3)传播无关类。

传播行为仅包含不传播,无污点信息流操作。

(4)函数级污点传播分析。

针对间接污染和潜在漏洞问题,在指令级污点分析外提出了函数级的污点传播分析,对相关函数和其对应的指令(已标记为“间接污染”)进行监控,记录和分析数据状态信息和数据流传播信息,在污点检测模

块误用检测时进行安全规则匹配,根据污点检测处理方式对分析结果进行处理。

### (5) 污点清除类。

传统的污点分析系统,在污点传播阶段只考虑污点信息的增加与传递,即污点传播状态转换的行为实体只包括污点添加、污点传播和不传播三类,未考虑污点清除的情况,会导致“过污染”情况产生,造成误报。如表 1 中的示例,指令中将常量赋给变量,或是 XOR、SUB、SBB 等指令结果为常量时,此时原操作数中包含的污点变量将不应再进行传播。基于此,提出了第四类污点传播行为实体,即污点清除,对符合上述条件的指令执行污点清除操作,不再进行污点传播。

### 1.4 污点检测模块设计

现有的污点分析平台,污点检测时一旦发现污点数据违背安全规则,通常只有一种处理方式,即终止运行并记录相关信息,当违背的安全规则仅仅是触发了某个危害较轻的安全风险,而非发现危害较重的某个漏洞,直接终止运行会造成检测效率低下。

为此,将触发的安全规则按照危险严重性分为三个级别,即“漏洞触发”、“潜在危险”和“安全操作”,根据不同级别对污点数据进行不同处理。当某污点数据在污点传播过程中触发漏洞时,则记录漏洞触发相关信息并终止运行;当某污点数据仅执行的操作危害较轻,仅为“潜在危险”时,则记录该污点数据风险操作的相关信息并继续运行;当污点数据执行的操作为“安全操作”,未违反任何安全规则,则执行继续。

## 2 系统实现

### 2.1 二进制动态插桩

原型系统 ODDTA 基于二进制分析平台 Pin<sup>[16]</sup>实现,Pin 借鉴了 ATOM<sup>[17]</sup>工具的两个概念:Instrumentation Routine(简称 IR,插桩例程)和 Analysis Routine(简称 AR,分析例程)。前者定义插桩的位置,后者定义插桩时需要执行的分析工作。Pin 框架提供指令集插桩、轨迹级插桩和函数级插桩等三种插桩粒度。文中选用轨迹级插桩。

### 2.2 污点标记模块实现

污点标记模块算法实现中变量和函数定义包括:

INST:汇编指令类型;

void SetTaintMark():设定污点源数据标记;

char \* GetInstAddr(INST):获取指令源地址;

int GetInstLen(INST):获取引入污点指令长度;

char \* GetInstDest(INST):获取指令目的地址;

SetMEMLabel(char \* DestAddr, int count, Taint-DataTaintLabel):设置内存单元污点标签;

SetREGLabel(char \* DestAddr, int count, Taint-

DataTaintLabel):设置寄存器单元污点标签;

bddAddList(SouceTaintList, DestAddr, count):将污点信息加入到影子内存中。

其中 bdd 为影子内存中基于规约有序二元决策图<sup>[18]</sup>(roBDD)方法设计的存储结构,该方法可实现对集合运算效率的优化,具体实现时调用开源的 BuDDY 库,支持几乎所有的 BDD 运算。

### 2.3 污点传播模块实现

污点传播模块以执行指令为依据,通过分析指令信息获取当前数据流和控制流的传播。算法实现中主要变量和函数定义说明如下:

INSTINFO:指令信息流类型;

AddrInfo:包含 char \* 格式的地址 addr,和污点传播模式 mode;

AddrInfoGetInstDest(INSTINFO):获取指令信息流传播目的地址信息集合;

int GetAddrCount(AddrInfo):信息流目的地址个数;

char \* \* GetRelateAddr(AddrInfo \*, int):获取信息流传播目的地址的相关源地址集合;

GetTaintInfo(char \* \*):通过 bdd 库进行目标目的地址的污点信息计算;

void SetDestTaintInfo(AddrInfo \*, int, bdd):设置目标地址污点信息;

void AddTaintInfo(AddrInfo \*, int, bdd):将污点信息添加至目标地址;

void ClearTaintInfo(AddrInfo \*, int):清除目标地址污点信息;

DestAddr[count].mode:三个取值 {in, add, clear} 对应污点传播状态的三种转换模式:污点传播、污点添加和污点清除。污点控制流传播算法实现如下:

1. TaintControlPropagate(INSTINFO inst, bddCurrentControlInfo)

2. char \* ControlSource = GetJCCSource(inst);

3. TaintInfo = GetControlTaintInfo(RelateSource);

4. AddControlInfo(CurrentControlInfo, TaintInfo);

char \* GetJCCSource(INSTINFO):获取条件跳转的受控源集合;

GetControlTaintInfo(char \* \*):通过 bdd 库获取受控源集合污点信息;

AddControlInfo(bdd, bdd):添加新控制源至当前污点控制信息。

### 2.4 污点检测模块实现

污点检测模块算法实现中主要包含的函数如下:

ModeGetInstMode(INST):获取指令模式(如 MOV\_REG32\_REG32);

boolSusceptibleModeMatch( Mode, Mode \* ): 匹配敏感指令模式;

InstStructGetInstStruct( INST, TaintInfo ): 获取指令结构;

boolSusceptibleStructMatch ( InstStruct, InstStruct \* ): 匹配敏感指令结构;

voidSensitiveOperate( ): 敏感点操作。

在实现污点检测时,依据污点误用检测的规则,设定了几种敏感模式,包括 MOV 类、JMP 类以及敏感函数对应的指令,当当前指令为上述指令类型时,则进入敏感函数匹配。当系统监测到敏感指令执行时,即有可能触发安全规则,进入危险状态。

3 实验分析

3.1 测试环境

实验测试环境即为原型系统的开发环境,底层硬件为曙光 i840-G25 和 DELL R510 服务器,基于 VMware vSphere4.1.0 实现硬件资源虚拟化,虚拟机虚拟硬件配置为 Intel Core2 Duo CPU @ 3.00 GHz x2/4 G/SATA 20 GB,操作系统为 Linux Ubuntu 12.04 (内核 3.8.0-32-generic)。

3.2 有效性测试

为验证污点分析原型系统漏洞挖掘的有效性,采用自编译的包含有缓冲区溢出漏洞的实例程序作为测试目标程序,该程序关键溢出代码如下:

```
intfunc( char * str)
{
    char buf[ 10 ];
    strcpy( buf, str );
    printf( “ % s \n ”, buf );
}

int main( intargc, char * * argv)
{
    char str[ 50 ];
    scanf( “ % s ”, & str );
    if( str[ 0 ] != ‘ Z ’ )
    func( str );
    return 0;
}
```

该程序中,执行 func( ) 中的函数 strcpy 时未对数组边界进行检查,当源数组长度大于目标数组长度时,会发生缓冲区溢出。对该程序进行编译,对生成的可执行文件输入不同的测试用例,分别进行漏洞检测有效性测试、潜在漏洞测试以及危险函数检测。

3.3 漏洞检测

以目标程序 test 作为 ODDTA 输入并执行,输入字符串为“abcdefghijklmnopqrstuvw”,共 24 个字符,

则可覆盖 func 函数调用后的返回地址,此时缓冲区漏洞被触发,程序异常退出。系统终止污点跟踪,输出安全分析日志中的相关信息,如下所示:

```
Program: test
Trace ID: test_55
Crash location: 0x004012b6
Crash instruction: RET
Crash case: themem can't execute
Detection rule: controlled execute
Target instruction address: 0x0022ff11
Target address taint:
0x0022ff11: { 22 }; 0x0022ff12: { 23 }
Taint link:
3027: 004012b6: RET M@ 0x0022ff11 [ 0x00007877 ] $ 4 T
2208: 77c160c1: MOV [ EDI ], EDX M @ 0x0022ff0e [ 0x13f00022 ] $ 4 UT R@ EDX [ 0x78777675 ] $ 4 T
.....
```

污点传播过程为:系统对输入数据进行污点标记和编号,污点数据按照指令执行,进行数据传输,当源缓冲区大于目的缓冲区时,发生溢出,并覆盖与缓冲区相邻的其他地址空间数据,当函数执行完毕后,此时返回地址 0x0022ff11 被覆盖,程序崩溃。

3.4 潜在漏洞测试

以目标程序 test 作为 ODDTA 输入并执行,输入字符串为“abcdefghijk”,共 12 个字符,结合实例程序执行 strcpy 时的栈状态可知,此时发生了缓冲区溢出,但并未覆盖返回地址,并不会造成程序崩溃。此时,早期的动态污点分析系统并不会给出存在潜在漏洞的安全警告,在 ODDTA 原型系统中,由于发生了外部引入污点数据的误用,此时在污点检测中将对应的指令操作标记为“Latent Danger”(潜在危险),输出如下信息:

```
Program: test
Detect target risklevel: Latent Danger
Misuse location: 0x004017c0
Misused mode: execute tainted instruction
Detail instruction: 004012b6: RET M@ 0x0022ff11 [ 0x00007877 ] $ 4 T
```

3.5 可疑点检测

ODDTA 原型系统记录控制流污点信息的传播过程,对目标程序执行进行细粒度污点分析后,可结合离线日志文件中的轨迹信息对每条指令进行细粒度分析,同时可结合控制流污点信息对分析出的漏洞可疑点进行辅助分析。以目标程序 test 作为原型系统输入并执行,输入字符串为“abcdef”,共 7 个字符,此时未发生缓冲区溢出,不会触发潜在漏洞。运行结束后对安全日志文件中的轨迹信息进行离线分析,发现当前存在脆弱性可疑点的匹配项 REP MOVSL ES: [ EDI ], DS: [ ESI ] M@ 0x003e37c0 [ 0x61656c70 ] \$ 4 UT M@

0x0022fef6[0x33323130] \$ 4 T R@ ecx[0x00000002]  
\$ 4 T,判定该位置存在缓冲区溢出可疑点,此时输出如下信息:

```
Program:test  
Trace ID: test_28  
Suspicious location:184  
Suspicious mode: buffer overflow  
Suspicious address:0x77c1168d  
Suspicious instruction: REPMOVSLE:[EDI],DS:[ESI]  
M@ 0x003e37c0[0x61656c70] $ 4 UT  
M@ 0x0022fef6[0x33323130] $ 4 T  
R@ ecx[0x00000002] $ 4 T  
.....
```

通过结合分析污点指令,发现原因在于在字符转移时,汇编程序会对其有效性进行判断,检测是否等于 0x09,0x0d,0x20 等。

程序结束时刻对应的控制流转移指令信息如下:

```
.....  
1311:4013e8;CMP 0x5a,AL,0x5a R@ AL[0x00000030] $ 1  
T I@ 0x00000000[0x0000005a] $ 1  
1312:4013ea;JE J@ 0x00000000[004013f8] $ 4 E@ 0x000000  
00[0x0000000e] $ 4 T  
.....
```

该指令中包含的污点变量与 char 型的 Z 的 ASCII 码 90(0x5a) 进行比较,只有相等情况下才执行跳转。

## 4 结束语

针对现有动态污点分析系统存在的准确率方面的问题,提出了基于动态污点分析的二进制程序漏洞挖掘与分析技术,从动态污点分析的三个主要阶段逐一准确度问题进行改进,有效解决了“过污染”造成的误报以及由于“欠污染”造成的漏报问题,增加了针对控制流的污点分析,并基于此方法实现了原型系统 ODDTA。实验结果表明,该方法能够有效提升漏洞挖掘的精准度和执行效率。下一步将结合符号执行和模糊测试技术,进一步细化控制流污点分析,实现测试路径的自动生成,提高对目标程序的分析效能。

## 参考文献:

- [1] LAM M S, MARTIN M, LIVSHITS B, et al. Securing web applications with static and dynamic information flow tracking[C]//Proceedings of the 2008 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation. New York, NY, USA: ACM, 2008:3-12.
- [2] 宋奕青. 基于动态二进制探测框架的缓冲区溢出检测研究[D]. 上海:上海交通大学, 2010.
- [3] SHARIF M, LANZI A, GIFFIN J, et al. Automatic reverse

engineering of malware emulators[C]//30th IEEE symposium on security and privacy. Washington, DC, USA: IEEE Computer Society, 2009:94-109.

- [4] COMPARETTI P M, WONDRACEK G, KRUEGEL C, et al. Prospex: protocol specification extraction[C]//30th IEEE symposium on security and privacy. Washington, DC, USA: IEEE Computer Society, 2009:110-125.
- [5] 史大伟, 袁天伟. 一种粗细粒度结合的动态污点分析方法[J]. 计算机工程, 2014, 40(3):12-17.
- [6] 李根. 基于动态测试用例生成的二进制软件缺陷自动发掘技术研究[D]. 长沙:国防科学技术大学, 2010.
- [7] WANG T, WEI T, GU G, et al. TaintScope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection[C]//IEEE symposium on security and privacy. Washington, DC, USA: IEEE Computer Society, 2010:497-512.
- [8] 诸葛建伟, 陈力波, 田繁, 等. 基于类型的动态污点分析技术[J]. 清华大学学报:自然科学版, 2012, 52(10):1320-1328.
- [9] 陈衍铃, 赵静. 基于虚拟化技术的动态污点分析[J]. 计算机应用, 2011, 31(9):2367-2372.
- [10] 宋铮, 王永剑, 金波, 等. 二进制程序动态污点分析技术研究综述[J]. 信息安全学报, 2016(3):77-83.
- [11] KOHLI P. Coarse-grained dynamic taint analysis for defeating control and non-control data attacks[EB/OL]. (2017-06-12). <https://arxiv.org/abs/0906.4481>.
- [12] YIN H, SONG D, EGELE M, et al. Panorama: capturing system-wide information flow for malware detection and analysis[C]//Proceedings of the 14th ACM conference on computer and communications security. New York, NY, USA: ACM, 2007:116-127.
- [13] NETHERCOTE N, SEWARD J. How to shadow every byte of memory used by a program[C]//Proceedings of the 3rd international conference on virtual execution environments. New York, NY, USA: ACM, 2007:65-74.
- [14] 黄强, 曾庆凯. 基于信息流策略的污点传播分析及动态验证[J]. 软件学报, 2011, 22(9):2036-2048.
- [15] 王卓. 基于符号执行的二进制代码动态污点分析[D]. 上海:上海交通大学, 2011.
- [16] 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.
- [17] SRIVASTAVA A, EUSTACE A. ATOM: a system for building customized program analysis tools[J]. ACM SIGPLAN Notices, 2004, 39(4):528-539.
- [18] 王铁磊, 韦韬, 邹维. 基于 roBDD 的细粒度动态污点分析[J]. 北京大学学报:自然科学版, 2011, 47(6):1003-1008.