基于类型的动态污点分析技术研究

陈力波[1,2]　诸葛建伟[2]　田　繁[1,2]　鲍由之[2]　陆　恂[2]
（1. 清华大学 计算机科学与技术系，北京 100084；2. 清华大学 网络科学与网际空间研究院，北京 100084）

摘要：动态污点分析技术是目前一种主要的软件漏洞分析技术，但在二进制代码分析中由于缺少类型支持，目前只能工作在内存地址与寄存器粒度上，为漏洞机理语义分析带来了较大困难。本文提出了基于类型的动态污点分析技术，定义外部输入变量为污点变量，并添加变量的类型信息与符号值作为污点属性，利用函数、指令的类型信息来跟踪污点变量传播过程，进行类型感知的污点传播以及面向类型变量的符号执行，从而获得污点变量传播图与程序执行路径条件，求解出输入变量的约束条件，支持全面了解输入变量在数据流中的传递和对控制流的影响，最终达到更好的安全漏洞语义理解。本文以分析浏览器安全漏洞为例应用该技术，运用动态二进制代码插装技术将分析代码插入目标进程，动态执行污点分析，设定策略规则检测进程是否非法使用污点数据，结合符号执行，解出输入变量约束条件。从数据依赖和控制依赖两方面，获得含有外部输入类型信息的安全漏洞特征。实验结果验证，基于类型的动态污点分析方法可以有效提升安全漏洞机理分析的语义，能够输出更易理解与使用的安全漏洞特征。

关键词：动态污点分析；软件漏洞；类型信息；污点传播；符号执行；漏洞特征
中图分类号：TP311

Research of technology for type-based dynamic taint analysis

CHEN Libo[1,2]　*ZHUGE Jianwei*[2+]　*TIAN Fan*[1,2]　*BAO Youzhi*[2]　*LU Xun*[2]
（1. Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China）
（2. Institute for Network Science and Cyberspace, Tsinghua University, Beijing 100084, China）

Abstract: Dynamic Taint Analysis is a major technique for software vulnerability analysis today. But since it lacks type information during analysis with binary code, it can only work on byte-level granularity, which poses much difficulty for the semantic analysis of vulnerability mechanism. This paper proposes type-based DTA. It marks the input variables as tainted and adds the type of the variable and their symbol value as attributes to the taint. Type Information within functions and instructions are used to track the propagation of tainted variables. Typed-a-

基金项目：国家自然科学基金（61003127）
作者简介：陈力波（1985—），男（汉），江西，硕士研究生。
通讯作者：诸葛建伟，副研究员，Email：zhugejw@cernet.edu.cn

ware taint propagation and type-oriented forward symbolic execution are performed to get the propagation map of tainted variables and the path-conditions, which are used to calculate the desired constraints on the inputs. Thus we know about the propagation of input variables in the data-flow as well as their impact to the control-flow, and eventually leading to a better understanding of vulnerability semantics. In this paper, this technology was applied to the analysis of a browser-based vulnerability by utilizing Dynamic Binary Instrumentation to inject the analysis codes into the target process to perform taint-analysis dynamically. The policy for detection checked whether the process illegally use the tainted data. Combining symbolic execution, the input constraints was calculated. Concerning both data and control flow dependencies, the type information embedded vulnerability signature was revealed. Our experiment showed that Typed DTA can significantly upgrade the semantics for the analysis of vulnerability mechanism and produce more comprehensible and usable vulnerability signature.

Key words: dynamic taint analysis; software vulnerability; type information; taint propagation; symbolic execution; vulnerability signature

引　言

随着互联网的快速发展,针对用户终端的网络渗透攻击层出不穷,形形色色的木马、病毒等恶意代码大肆窃取用户信息。然而,这些攻击的背后起决定因素的则是终端上运行的各类软件所包含的安全漏洞。正是这些漏洞的存在,才使得攻击者得以获取终端的控制权来运行恶意代码,植入后门或木马。因此,软件安全漏洞是威胁互联网用户终端主机系统安全的根本原因。

软件漏洞的根源是现代电子计算机在实现图灵机模型时,没有在内存中将代表规则集合的程序代码和代表状态集合的数据严格地区分开。内存中的数据也能够被程序当作代码执行。针对这个本质原因,攻击者通过各种手段挖掘、利用软件漏洞,实施各种内存攻击,构造畸形的外部输入数据使得程序违背了设计者的初衷而出错,从而改变程序控制流,窃取程序的控制权。面对层出不穷的内存攻击,安全界从各个方面提出了多种技术手段。

近些年,利用动态数据流分析来应对此类安全威胁引起了广泛的关注,其中又以动态污点分析技术最为突出[1]。动态污点分析从软件安全漏洞的本质出发,标记外部输入为污点数据,跟踪污点数据在内存中的扩散过程,检测程序是否非法使用污点数据,譬如覆盖栈中返回地址,作为 EIP 的值等等。由于其抓住了软件安全漏洞的实质,无论安全漏洞是否已经公开,该检测技术在原理都是行之有效的。与此同时,随着硬件计算能力的提高,在程序分析领域,符号执行[2]作为分析程序语义的形式化方法和分析技术得到越来越广泛的应用,尤其是在软件安全漏洞领域。相比于动态污点分析对数据流的关注,符号执行对应于对控制流的分析。结合动态污点分析与符号执行,通过对输入数据的符号化进行符号执行,得到各个程序分支处的谓词而构成程序执行路径条件,约束求解出输入数据满足的

条件；与此同时，内存中污点数据所包含的符号则对应了其与输入数据的对应关系。使得分析人员能够从控制依赖、数据依赖两个方面解析外部输入触发软件安全漏洞的这一程序行为。

然而，威胁用户的安全漏洞大部分存在于闭源的商业软件当中，这使得上述的两种技术手段往往面对的是底层二进制代码，而非源代码。由于在二进制代码中，程序缺乏包含语义和语法的类型信息。对于污点分析来说，污染源是外部输入在底层的比特流，我们只能按字节等内存操作数的粒度来定义污点数据；同样的，对于符号执行来说，对输入数据的符号化也将针对字节粒度等。这不仅增加了技术实现的复杂性，加大了暴露技术本身缺陷的可能性，譬如符号执行存在的路径爆炸，而更为关键的是我们得到的分析结果只能局限于没有类型信息的外部输入中。例如，污点分析的结果是外部输入文件中的某些字节污染了程序的 EIP 寄存器；符号执行得到的符号表达式往往非常庞大、冗余很多，而由此得到的路径条件往往相当复杂或难于求解。

最近的研究表明，在底层的二进制代码中存在揭示类型信息的位置[3]。比如，系统调用、标准库函数调用、含有特定类型信息的指令，例如，MOVS/B/D/W、STOS/B/D/W 等。研究人员将其称之为 Type Sink 点[3]。这些位置的指令、函数的公开类型信息可以应用在底层的内存操作数中，恢复其对应的上层结构变量及其语法、语义信息。有些研究工作利用这些位置的库函数语义进行函数抽象，做语义感知的污点传播，提升污点分析的效率[4]。除此之外，基于组件架构的程序开发模式，诸如微软的 COM 编程以及 Mozilla 平台的 XPCOM，使得获取模块外部输入及类型信息成为可能。这些组件模块暴露的接口由统一标准的系统函数调用，研究人员可以通过 hook 这些函数调用的方式，监控组件对象创建及方法调用的过程，得到外部输入数据的类型信息和内存地址[5]，我们将其称为 Type Source 点。

在上述研究的基础上，我们提出基于类型的污点分析技术。该技术将利用 Type Source 点的类型信息，获得外部输入字节粒度数据流中的输入变量，将其标记为污点变量，添加类型信息和符号值作为其污点属性，构成污点源。在污点传播的过程中，在跟踪指令对字节粒度污染数据传播的基础上，有条件地提升为变量级污点传播，利用指令、公开库函数的语义，结合 Type Sink 点揭示的类型信息作为补充，实现类型感知的污点变量传播，在污点传播的同时实施面向类型变量的符号执行。在污点传播的过程中检测程序是否非法使用污点数据。在程序执行后，回溯污点数据传播过程得出污点变量传播图，结合符号执行得到的程序执行路径条件，解出输入变量对应于程序行为需要满足的约束条件。如果检测到污点数据非法使用，则约束条件对应于安全漏洞特征。

我们的主要创新点有以下两点：

1）分析利用指令、函数的语义传播污点数据，并赋予污点数据类型信息，进行

变量级污点传播,实现类型感知的污点传播;

2)利用污点变量的类型信息,赋予污点变量成员统一的符号值,精简符号运算,实现面向类型变量的符号执行。

基于类型的污点分析利用 Type source 的类型信息,结合底层二进制代码中对变量类型的传递和揭示,利用指令和函数的语义传递、推导污点变量。这不仅有效精简了污点分析和符号执行,提高了效率,增加了准确性,而且分析结果将具有高层语义,有助于分析人员分析、理解软件安全漏洞。

本文将以检测分析浏览器漏洞为应用场景,利用该技术获取具有高层语义的安全漏洞特征。实验结果验证了技术的有效性。

本文分为 4 个部分:第 1 节将介绍相关的研究工作;第 2 节介绍基于类型的动态污点分析技术原理与系统设计;第 3 给出系统的具体实现,以及测试结果;第 4 节进行总结。

1 相关工作

CMU 的 J. Newsome 和 D. Song 等人将动态污点分析技术——Dynamic Taint Analysis[1]运用到网络渗透攻击检测、分析、特征值提取中,并在 X86 模拟器 Valgrind[6]上实现污点分析工具:TaintCheck[1]。DTA 技术专注于外部输入数据的污点属性,抓住了软件安全漏洞的本质,能够检测到未公开安全漏洞的渗透攻击。

我们的研究工作与传统 DTA 技术的不同在于以下两点:

1)与传统 DTA 技术仅仅做字节粒度的污点分析不同,基于类型的动态污点分析在字节粒度基础上复合了变量粒度的污点传播,使得我们的分析结果具有了上层变量的语义和语法信息。

2)传统的 DTA 不关注于污点数据对控制流的影响,有隐式流的问题[7]。基于类型的动态污点分析将会利用符号执行获取控制流的程序执行路径条件,以及对操作污点变量的库函数进行抽象,来处理对隐式流的跟踪。

符号执行是程序分析中用于分析程序语义的一种形式化方法和分析技术。如果将外部输入数据符号化,符号执行能够形式化地表达程序的外部输入与当前程序执行路径的关系以及外部输入与程序中变量的对应关系。如果与污点分析结合可以弥补污点分析的一些原生性缺陷,比如提供路径信息、获得内存中污点数据与输入数据的对应关系等。因此,在安全漏洞领域,结合使用动态污点分析与符号执行已经成为一个新的技术热点,比较有影响力的包括:TaintScope[8]、DsVD[9]。

但是这些研究都面临着同样的挑战:在将符号执行移植到底层二进制代码的过程中,由于缺乏类型信息,相比较于源代码中的分析,其复杂性大大增加,导致

符号执行的一些原生性问题[10]更加突出。在基于类型的污点分析中,我们将利用污点变量的类型信息精简符号执行,对污点变量的成员赋予统一的符号,提高符号执行的效率。

由 REWARDS 的研究[3]可知,在程序的执行过程中,底层运行的二进制代码中存在 Type Sink 点,可以揭示内存操作数的上层结构体变量类型信息。相比较于 REWARDS,由于我们从 Type Source 点获得了输入变量类型,采用正向的类型传递,而不是 REWARDS 中从 Type Sink 点的逆向类型推导。在我们的研究中,为了更好地传递污点变量的类型,可以结合这些 Type Sink 的类型信息,比如在库函数的位置提取参数、返回值的类型信息给污点变量。

函数抽象指的是对函数的某一部分功能单独抽取出来进行分析和利用。在 TaintEraser 的工作中[4],研究人员利用库函数、系统函数的公开语义,抽取出函数传播污点内存的功能。研究人员将其称为语义感知的污点传播。本文在这个基础上进行扩充,当污点变量遇到上述函数调用时,不仅利用语义将污点属性扩展到目的操作数,而且还根据函数参数的类型赋予目的内存操作数类型信息,将污点内存提升为污点变量。

2 基于类型的动态污点技术原理及系统设计

2.1 技术原理

污点分析与符号执行这两项技术在应用于安全漏洞分析时拥有很好的互补性,但是由于底层二进制代码缺乏类型信息,制约了这两项技术在二进制代码分析场景中的结合效果。现有的研究往往以文件系统接口作为输入点。程序调用系统函数去读取外部文件到内存中,污点分析将其中的每个字节标记为污点,内存地址加入污点内存列表;符号执行将每个字节赋予一个不同的符号值。这样的污点分析与符号执行技术经常由于实现的复杂性导致系统效率低下。由于缺乏语义信息,其分析结果往往复杂度很高、冗余很大、可读性很差,不利于研究分析。所以,我们希望做以下四点改进:

1)利用组件架构编程的特点,hook 程序对组件接口的调用,获取外部调用模块的函数及其参数,解析出外部输入数据流中的输入变量地址和类型信息,构成 Type Source 点。将这个输入点标记为污点源,添加输入变量为污点变量,使得污点源数据具有了类型信息;

2)针对同一变量的污点内存,有条件地从字节粒度提升到变量粒度上做污点传播,利用函数、指令语义进行变量类型传递与推导,提高效率;

3)对同一污点变量赋予统一的符号,精简符号执行,提高效率;

4)污点分析和符号执行的结果具有输入变量的类型信息,有助于安全漏洞机理的语义分析。

基于以上考虑,我们提出基于类型的动态污点分析技术。具体原理如图 1 所示。

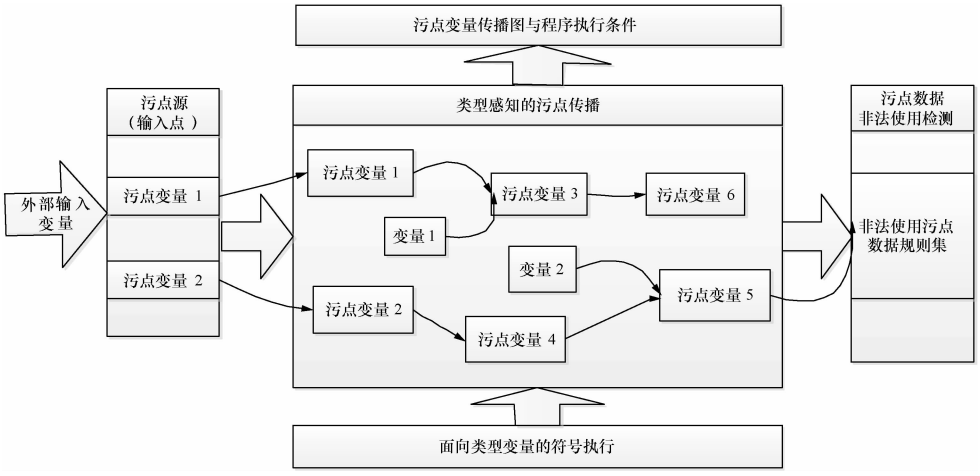

图 1 基于类型的动态污点分析技术原理

在图 1 中,利用输入点的类型信息将输入变量标记为污点,添加变量类型信息和统一的符号值作为污点属性。在程序执行时分析指令与函数对污点变量的操作,依照表 1 所示例的关系对污点变量类型进行传递与推导,实现类型感知的污点传播。在污点传播的同时,对应于指令对污点变量的操作,进行面向类型变量的符号执行。在程序执行过程中检测污点数据是否被非法使用,记录被非法使用的污点数据。程序执行之后,回溯污点传播过程,得到污点变量传播图,将其应用于符号执行得到的程序执行路径条件中,解出输入变量对应于程序行为需要满足的约束条件。

如果将该技术应用于分析检测安全漏洞攻击中,可以在检测到非法使用污点数据后,结合符号执行得到的路径条件和污点变量传播图,解出外部输入变量满足的约束条件,得出包含高层语义的安全漏洞特征。接下来介绍该技术的两个主要创新点:类型感知的污点传播,和面向类型变量的符号执行。

2.2 类型感知的污点传播

在污点传播过程中,根据指令或函数的语义分以下两类情形来处理污点变量,有条件地提升为变量级污点传播:

1)当前指令或函数不能继承、传递源操作数变量类型时,如果源操作数为污点变量,那么目的操作数将被标记为污点,其内存地址或寄存器将列入污点地址

列表，根据指令的内存操作数粒度标记目的操作数类型为 BYTE、WORD、DWORD 等。并且继承源变量的污点属性中除类型之外的其他属性。

2）当前指令或函数可以继承、传递包含源操作数的变量上的类型信息时，将把污点传播从内存操作数粒度提升为变量级污点传播。即如果源操作数是污点变量，那么将目的操作数标记为污点变量，变量类型根据表 1 进行传递推导，并将变量中各个成员的内存地址加入到污点地址列表中，以及在各个成员的污点属性中加入类型信息，依据变量传递关系得出污点变量的推导公式。

表 1　变量类型传递与推导关系

源操作数变量类型	操作：指令/函数	目的操作数变量类型
整形	算术运算（ADD，ADC，SUB，SBB，INC，…）	整形
	逻辑运算（AND，OR，NOT，TEST，…）	整形
	数据传送（MOV，XCHG，PUSH，POP，LEA，XLAT…）	整形
	浮点运算（FIDIV，FIADD，FISUB，FICMP…）	浮点型
	浮点数据传送（FILD，…）	浮点型
	系统函数、库函数（itoa，itow，…）	根据函数语义得到目的操作数变量类型，包括整形、浮点型、字符串等
浮点型	浮点运算（FADD，FSUB，FCMP…）	浮点型
	浮点数据传送（FST，FSTP…）	浮点型
	浮点数据转化格式（FIST，FISTP）	整形
	系统函数、库函数（gcvt，fcvt，…）	根据函数语义得到目的操作数变量类型，包括整形、浮点型、字符串等
字符串	字符串传送（[REP]MOVSB/MOVSW/MOVSD，[REP]STOSB/STOSW/STOSD…）	字符串
	数据传送（MOV，XCHG，…）	字符
	系统函数、库函数（strcpy，strcat，strlen，atoi，atol…）	根据函数语义得到目的操作数变量类型，包括整形、浮点型、字符串等

类型感知污点传播的关键是根据指令和函数的类型信息来制定出相应的污点传播策略。以库函数为例进行说明。由于库函数的类型信息公开,所以我们将对其进行函数抽象,如下:

1)根据库函数的功能语义,设定由源污点变量推导出目的污点变量的推导公式,如果返回值与污点变量属性相关,那么也将标记为污点变量。

2)根据参数与返回值的类型信息,设定目的污点变量的类型,加入到污点属性中。

以容易造成缓存区溢出的 libC 库函数 strncpy 为例,该库函数公开参数与返回值类型如表 2 所示:

表 2 库函数 strncpy 的类型信息

Copy characters of one string to another char * strncpy(char * strDest, const char * strSource, size_t count); Parameters: strDest Destination string. strSource Source string. count Number of characters to be copied. Libraries All versions of the C run-time libraries. Return Values Each of these functions returns strDest. No return value is reserved to indicate an error.

该库函数的功能语义是拷贝一个 char 型的字符串,参数和返回值的类型如表中所示。根据上述类型信息,进行函数抽象,设定污点传播策略如下:

a) 如果源字符串指针 strSource 指向污点变量,那么目的地址 strDest 指向的长度为 count 的字符串变量标记为污染,将每个字节地址添加到污点地址列表,返回值与 strDest 相同,所以不需再另做处理,污点变量推导公式如式(1)所示:

式 1 污点变量推导公式

$$strDest[n]=strSource[n],\ (n\in[0,\ N-1])$$

b) 每个变量成员的污点属性中添加参数类型,其中,语义为参数名称,语法包括参数类型和布局。前两个字节的污点属性具体变化如表 3 所示:

表 3　污点属性变化

strSource[0]的污点属性	strDest[0]的污点属性
syntax：char ＊/offset：0	syntax：char ＊/offset：0
semantic：strSource	semantic：strDest
symbol：content	symbol：content
strSource[1]的污点属性	strDest[1]的污点属性
syntax：char ＊/offset：1	syntax：char ＊/offset：1
semantic：strSource	semantic：strDest
symbol：content	symbol：content
….	

整个过程如图 2 所示：

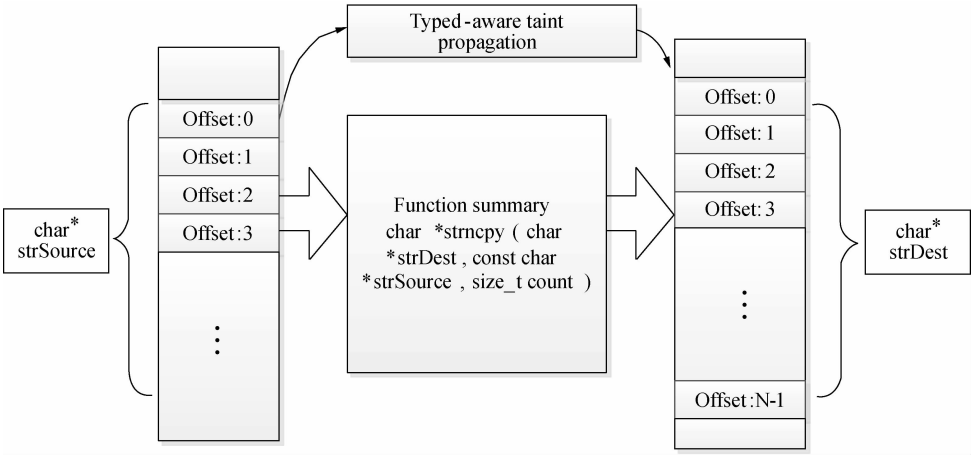

图 2　类型感知的污点传播示例图

2.3　面向类型变量的符号执行

在污点传播的同时，对污点变量的每个成员赋予统一的符号值，将其作为污点属性的一部分参与到污点传播中，进行符号执行。具体分为以下三步：

1）符号初始化部分

在输入点和污点传播过程中产生新的污点变量时，在其成员污点属性中加入统一的符号值；

2）符号执行

在污点传播的过程中，符号作为污点属性一起传播，当遇到指令修改成员变量时，符号作相应的修改，我们的策略是将指令的操作符与操作数的值加入到原符号中，形成新的符号；

3)提取分支谓词

在程序分支处,如果 EFLAGS 寄存器被污染,那么提取其属性中的符号值以及分支跳转条件,组成程序分支谓词;

4)程序执行路径条件

组合所有的分支谓词,得到程序执行路径条件。

以字符串为例,将每个字节都赋予代表内容的符号值 content。可以这样做的理由是,由于每个字节的语法信息中包含了当前字节在变量中的布局,即前述例子中的 offset。所以,结合这个偏移值就可以唯一的表示这个字节。而 content 字符将用于指令执行时做符号执行,具体策略如下:

1)对拷贝、传递等不修改内容的指令,符号在指令执行或函数抽象中随污点数据一起进行传递,符号值不变;

2)对算术运算、布尔操作等修改内容的指令,将操作符和操作数的值或符号加入到原符号中,产生新的符号值;

以表 4 中的一个小程序的二进制代码为例来展示每条指令对应的符号执行。假设 esi 寄存器指向外部输入的参数 char * userdata。每条语句的符号执行在语句的左侧,分支处输出谓词。

表 4 面向类型变量的符号执行

mov cl, byte ptr[esi]	byte ptr[esi].symbol: content byte ptr[esi].syntax: char * /offset 0 byte ptr[esi].semantic: userdata byte ptr[esi] Taint cl cl.symbol: content; cl.syntax: char cl.semantic:
dec cl	cl.symbol:(dec content) cl taint EFLAGS EFLAGS.symbol:(dec content)
test cl, cl	cl.symbol:(test (dec content, dec content)) cl Taint EFLAGS EFLAGS.symbol:(test (dec content, dec content))
jnz short 10019236	分支谓词:(test (dec content, dec content))&& EFLAGS.zf==0

续表

push esi	[esi]. symbol：content [esi]. syntax：char * /offset 0 [esi]. semantic：userdata
call strlen	执行函数抽象，根据函数语义推导出新污点变量 eax. symbol：size eax. syntax：unsigned int eax. semantic：userdata. size
dec eax	eax. symbol：(dec size) eax taint EFLAGS EFLAGS. symbol：(dec size)
test eax，eax	eax. symbol：(test (dec size)，(dec size)) eax Taint EFLAGS EFLAGS. symbol：(test (dec size)，(dec size))
jnz short 10019236xor eax，eax ret … sub 10019236 …	分支谓词：(test (dec size，dec size))&& EFLAGS. zf==0

上述代码两个分支处谓词组成的程序执行路径条件如式(2)所示。

式 2 程序执行路径条件

$$(test(dec\ content,\ dec\ content))\&\&EFLAGS.zf==0$$
$$(test(dec\ size,\ dec\ size))\&\&EFLAGS.zf==0$$

回溯污点传播过程得出，符号 content 对应于输入变量 userdata[0]，size 对应于变量 userdata. size。替换式 2 中的符号，求解出输入变量的约束条件：userdata[0]！=1&&userdata. size>1，即输入字符串的首字符 ANSII 值不等于 1 且大小大于 1，可以看到这个约束条件具有了输入数据的类型信息。

2.4 系统设计

总体设计思路是利用动态代码插装技术[10]，在目标程序运行时动态插入实施污点分析的代码，跟踪污点变量传播的同时做符号执行，在程序运行之后，利用记录的信息得出污点变量传播和程序执行路径条件，求解输入变量满足的约束条件。

具体分为四个部分：污点源初始化部分、跟踪记录污点变量传播部分、检测污

点数据非法使用部分、污点传播信息处理部分。

1)污点源初始化部分

a) 截获程序的外部输入变量地址和类型信息

在程序获取外部输入数据时,通过截获接口的函数调用,提取函数参数中与外部输入相关的输入变量类型和地址信息;

b) 定义外部输入变量为污染源

将输入变量成员的内存地址加入污点内存地址列表,且列表中每项存有一个结构体指针,指向一个代表污点属性的结构;

c) 构造每个变量成员对应的污点属性,包括语法、语义、符号。

2)跟踪污点数据的传播部分

a) 分析函数、指令的语义,根据表 1 所列的变量类型推导关系设定污点变量的传播策略,执行类型感知的污点传播,记录污点变量传递、推导过程;

b) 在推导出新的污点变量时,将每个成员的污点属性中加入统一的符号值,符号值作为污点属性的一项参与污点传播的同时做面向类型变量的符号执行,记录符号执行的过程。

3) 检测污点数据的非法使用

设置程序非法使用污点数据的规则。比如,污点数据覆盖栈中返回地址等。当程序出现上述情形使用污点数据时,记录这个策略违背点。

4)污点数据传播信息处理部分

a) 污点数据流传播图

根据污点传播过程中记录的信息,回溯污点数据的传播过程,得到污点变量传播图,提取污点变量传递过程中的推导公式;

b) 提取程序执行路径条件

利用成对出现的 callret 指令,剔除程序切片中位于已经结束调用的子函数中的分支,剩余的分支将是影响程序执行的关键分支谓词,将其组合为当前程序执行路径条件;

c) 求解输入变量约束条件

结合污点变量传播图中的变量推导公式,将输入变量成员代入程序执行路径条件中,求解出输入变量约束条件。

3 系统实现与测试

3.1 系统实现

以检测分析 IE 浏览器安全漏洞作为应用场景,分四个模块实现系统:

1)监控外部输入模块

以 COM 组件中的一类 ActiveX 为例,利用开源项目 AxMock[①] 截取 ActiveX 的外部输入。AxMock 项目利用了 ActiveX 的特点, Hook 自动化接口的统一方法调用——Invoke 函数,可以监控到 IE 对 ActiveX 组件方法的调用以及参数值和类型信息,记录下参数值和类型;

2)污点变量跟踪及符号执行模块

这个模块利用动态插装平台 PIN[②] 开发。利用 Pin. exe 程序将该模块插入到已经附加了 AxMock 的 IE 浏览器中。当浏览器打开网页时,分析代码根据第一个模块提供的参数值和类型定义输入变量为污点变量,赋予污点属性,接下来进行污点分析和符号执行;

3)检测污点数据非法使用模块

根据应用场景设定程序非法使用污点数据的规则,在程序运行时,如果发现出现规则中描述的情形,则记录该策略违背点的指令地址和污点数据地址和属性;

4)污点传播信息处理模块

a)根据污点传播记录的信息,回溯污点数据传播过程,得出污点变量传播图;

b)将符号执行得到的分支谓词组合成程序执行路径条件,结合污点分析得到的污点变量推导公式,利用求解器 Z3[③] 消除中间变量,解出外部输入变量满足的约束条件。

整个系统实现如图 3 所示:

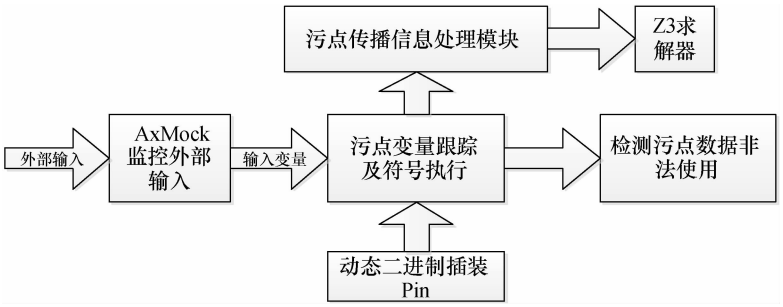

图 3 系统实现概图

① http://code.google.com/p/axmock/

② http://www.pintool.org/

③ http://research.microsoft.com/en-us/um/redmond/projects/z3/index.html

3.2 漏洞实例测试过程与结果

以一个 IE 浏览器的 ActiveX 插件漏洞为例进行测试。该安全漏洞编号为 CVE-2011-3142,利用公开的渗透测试网页样本[①]进行测试。网页源代码如下表 5 所示:

表 5 测试网页源代码

```
<html>
<object classid='clsid: F31C42E3-CBF9-4E5C-BB95-521B4E85060D' id='target' />
></object>
<script language='javascript'>
nse="\xEB\x06\x90\x90";
seh="\x4E\x20\xD1\x72";
nops="\x90";
while (nops.length<10){ nops+="\x90";}
/* Calc.exe alpha_upper badchars -->  "\x8b\x93\x83\x8a\x8c\x8d\x8f\x8e\x87
\x81\x84\x86\x88\x89\x90\x91\x92\x94\x95\x96\x97\x98\x99\x82\x85\x9f\x9a\x9e\
x9d\x9b\x9f\x76 */
shell="\x54\x5f\xda\xdf\xd9\x77\xf4\x5e\x56\x59\x49\x49\x49\x49\x43\x43\
x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58\x34\x41\x50\x30\x41\x33\
x48\x48\x30\x41\x30\x30\x41\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\
x42\x42\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4c\x4b\x5a\x4c\x50\x55\
x4c\x4b\x5a\x4c\x43\x58\x51\x30\x51\x30\x51\x30\x56\x4f\x52\x48\x52\x43\x45\
x31\x52\x4c\x43\x53\x4c\x4d\x51\x55\x5a\x58\x56\x30\x58\x38\x49\x57\x4d\x43\
x49\x52\x54\x37\x4b\x4f\x58\x50\x41\x41";
junk1="A";
junk2="A";
while (junk1.length<624){ junk1+=junk1;}
junk1=junk1.substring(0, 624);
junk2=junk1;
while (junk2.length<8073){ junk2+=junk2;}
arg2=junk1+nse+seh+nops+shell+junk2;
arg1="Anything";
target.ValidateUser(arg1, arg2);
</script>
```

测试分为以下几步:

1)启动 IE 浏览器加载 AxMock,获取插件的获取外部输入的方法及参数和类型等相关信息,写入日志文件。其中与污点源相关的部分如表 6 所示:

① http://www.exploit-db.com/exploits/16936/

表 6 AxMock 日志

```
clsid：{F31C42E3-CBF9-4E5C-BB95-521B4E85060D}
Invoke address：02A08060
method：ValidateUser
parameterCount：2

arg type：8(BSTR)
arg address：001F044C
arg end address：00205B24
arg：AAAAAAAAAAAAA...
arg Length：10971

arg type：8(BSTR)
arg address：02D90448
arg end address：02D90488
arg：Anything
arg Length：8
------------
Type Information：
Func：ValidateUser
[Helpstring：方法 ValidateUser]
arg1：Name
vartype：BSTR
arg2：Password
vartype：BSTR
```

2）利用 Pin 将分析代码插入 IE 进程，读取 AXMock 日志文件，确定 type source 点，定义污点源变量，跟踪污点变量传播过程，写入日志文件。检测模块发现污点数据覆盖了栈中的返回地址，记录该策略违背点；

3）根据日志中的记录的污点变量传播信息，回溯污点数据传播过程，得出污点变量传播图，其示意图如图 4 所示；

4）在污点变量传播图中，回溯覆盖返回地址和程序分支处污点数据的传播过程（图 4 中带箭头的虚线代表），得到包含这些数据的污点变量传播过程及推导公式，如下：

a）覆盖栈中返回地址的字节来源于变量 lpMultiByteStr2[264]，由符号 content 未变化可知，内容没有改变；

b）回溯污染程序分支的污点数据，得到第一个分支的 size 符号来源于变量 int size1；第二个分支的 content 来源于 char lpMultiByteStr2[n]，（$n \in [0, 10971]$）。

c）提取回溯过程中的污点变量推导公式，如式(3)：

式 3 污点变量推导公式

$$Name.size = size1;$$
$$Password[n] = lpMultiByteStr2[n],\ (n \in [0,\ 10743])$$

5) 提取被污染的程序分支谓词,组成程序执行路径条件,如下式(4)所示:

式 4 程序执行路径条件

$$Test(DEC\ size1,\ DEC\ size1)\&\&EFLAGS.zf==0;$$
$$Test(lpMultiByteStr2[0],\ lpMultiByteStr2[0])\&\&EFLAGS.zf==0;$$
$$Test(lpMultiByteStr2[1],\ lpMultiBytestr2[1])\&\&EFLAGS.zf==0;$$
$$\cdots.$$
$$Test(lpMultiByteStr2[263]lpMultiBytestr2[263])\&\&EFLAGS.zf==0$$

6) 从两方面得到触发安全漏洞的程序行为特征:

a) 控制流方面

结合式(3)的推导公式和式(4)的路径条件,用求解器消除中间变量得到输入变量满足的约束条件,如式(5):

式 5 输入变量约束条件

$$Name.size!=0\&\&Password[n]!=0,\ (n \in [0,\ 263])$$

b) 数据流方面

结合污点变量传播图和栈中污点数据符号值未变可知,输入变量 Password 将被不改变内容的拷入到栈中,其中 Password[264]将覆盖返回地址。

至此,该安全漏洞的特征可以完整的描述为:clsid 为 F31C42E3-CBF9-4E5C-BB95-521B4E85060D 的插件调用方法 ValidateUser 时,参数 Name 的长度大于 0 且参数 PassWord 的长度大于 264 个字符时,将触发缓存区溢出漏洞;参数 Password 的内容将会不经改变的复制到栈中,且从第 265 个字符起开始覆盖栈中的返回地址。

需要指出的是,如果恢复溢出前的栈信息,可以看到程序在栈中预留 260 个字节的缓存区,加上预留给结束字符的位置,可以推断出源代码中的局部变量应该类似 char temp[256]。那么,漏洞的本质原因就是将参数 Password 的数据拷入到这个局部变量时,没有检查拷贝长度是否超过临时变量大小。可见,我们得到的特征中的语义对于推断处漏洞的根本原因很有帮助。

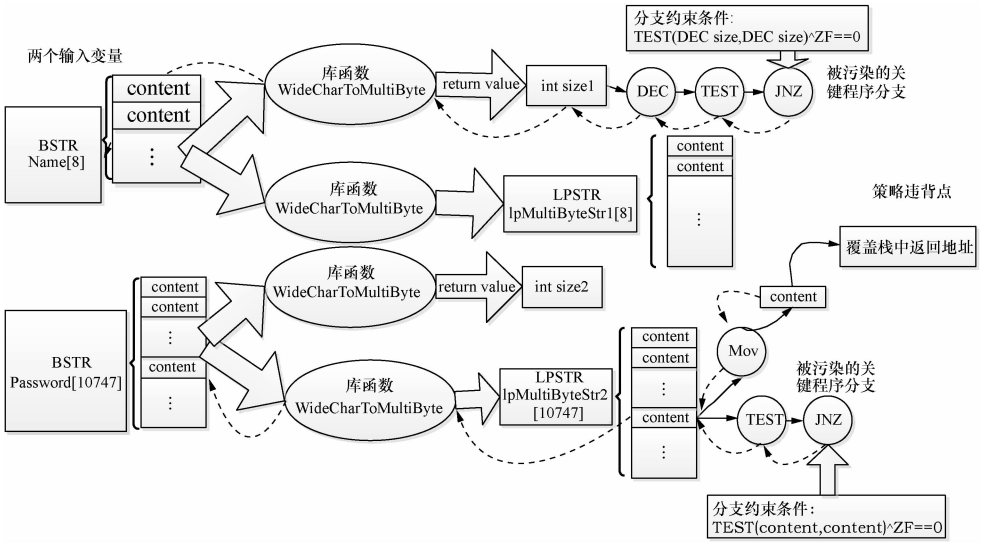

图 4 污点变量传播图

3.3 公开样本集测试结果

我们从公开披露漏洞的著名网站 exploit-db 上选取部分 IE 浏览器的 ActiveX 安全漏洞进行测试，结果如表 7 所示，其中包含漏洞编号和对应的安全漏洞特征。

表 7 样本集测试结果

漏洞编号	插件 ClassID 号及漏洞所在的函数	漏洞特征	漏洞特征说明
CVE-2007-0018	clsid：77829F14-D911-40FF-A2F0-D11DB8D6D0BC func：SetFormatLikeSample	$fileName[n]!=0$，($n\in[0,4095]$)	插件在调用函数 SetFormatLikeSample 时，若参数 fileName 长度大于 4096 个字符时，触发漏洞，参数值被复制到栈中覆盖返回地址
CVE-2007-2919	clsid：BA83FD38-CE14-4DA3-BEF5-96050D55F78A func：LoadOpf	$SubURL[n]!=0$，($n\in[0,511]$)	插件在调用函数 LoadOpf 时，前七个参数为任意字符串，若第八个参数 SubURL 长度超过 512 个字符，触发漏洞，该参数值被复制到栈中，覆盖返回地址

续表

漏洞编号	插件 ClassID 号及漏洞所在的函数	漏洞特征	漏洞特征说明
CVE-2008-3008/MS08-053	clsid：A8D3AD02-7508-4004-B2E9-AD33F087F43C func：GetDetailsString	*bstrProfileName*[*n*]!=0，(*n*∈[0，79])	插件在调用函数 GetDetailsString 时，参数 lcid 为 Long 类型，参数 bstrProfileName 长度大于 80 时，该参数被复制到栈中，覆盖返回地址
EDB-ID：16242	clsid：7677E74E-5831-4C9E-A2DD-9B1EF9DF2DB4 func：HttpPost	*WebUrl*[0]=0*x*68&& *WebUrl*[1]=0*x*74&& *WebUrl*[2]=0*x*74&& *WebUrl*[3]=0*x*70&& *WebUrl*[4]=0*x*3*A*&& *WebUrl*.*size*>2052	插件调用函数 HttpPost，如果参数 WebUrl 以“http:”开头并且后面的字符串长度大于 2052，则触发漏洞，参数被复制到栈中，覆盖返回地址
CVE-2010-3106	clsid：36723F97-7AA0-11D4-8919-FF2D71D0D32C func：ExecuteRequest	*operation*=”*op-client-interface-version*”&& *options*[0～5]=”*debug*=”&& *options*.*size*>526&& *options*.*size*<1018	插件在调用函数 ExecuteRequest 时，参数 operation 为“op-client-interface-version”，参数 options 以“debug=”开头，并且后面部分的长度大于 526 字符小于 1018 字符，触发漏洞，该参数值会被复制至栈中，覆盖返回地址
EDB-ID：14605	clsid：3C88113F-8CEC-48DC-A0E5-983EF9458687 func：OpenFile	*Inputfile*.*size*>216	插件调用函数 OpenFile 时，参数 Inputfile 长度超过 216 个字符时，漏洞触发，该参数值被复制至栈中，覆盖返回地址
CVE-2010-4321	clsid：36723F97-7AA0-11D4-8919-FF2D71D0D32C func：GetDriverSettings	*printerUri*[*n*]!=0，(*n*∈[0，75])	插件在调用函数 GetDriverSettings 时，参数 printerUri 长度超过 76 个字符，另外三个参数为任意字符串情况下，该参数值会被复制到栈中，覆盖返回地址

从测试结果看到，通过该系统的分析，可以得到测试样本触发安全漏洞所要具备的输入变量约束条件，该条件较之传统的符号执行结果大为精简，且具有上层的类型信息，从而结合前端输入点记录的方法调用和样本触发相应的非法使用污点变量规则，得出相应的具有高层语义的完整漏洞特征。

4 结　论

针对传统动态污点分析技术在二进制程序分析中的弱点，本文提出了基于类型的动态污点分析技术，充分利用输入点 Type Source 的类型信息进行类型传递与推导，同时结合 Type Sink 点的类型信息，将二进制程序分析中的动态污点分析提升到变量粒度，并与变量级别的符号执行进行有效结合，使得安全漏洞分析与特征提取具有更好的语义支持。

技术创新点体现在以下四个方面：

1)将输入点 Type Source 的类型信息结合到污点分析中，划分、定义污点源变量；

2)利用公开库函数、指令类型信息提升了污点分析的粒度，从而提高了污点分析的效率，得到了污点变量级传播图；且利用库函数语义，适当扩充了污点变量，完善了对污点数据影响控制流的监控，提高了准确率；

3)针对同一污点变量，精简污点数据的符号值，降低了符号执行的复杂性，大大提高了效率；

4)针对给定的程序行为，能够获得具有上层语义的输入变量约束条件，有利于分析理解安全漏洞的本质原因，并提取出易于使用的安全漏洞特征。

目前在实验层面上尚不支持结构、指针等复杂的数据结构变量类型，计划在下一步工作中完善。

参 考 文 献

[1] Newsome J, Song D. Dynamic. Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software[C]. //Proceedings of 12th Annual Network and Distributed System Security Symposium(NDSS'05), San Diego, California, USA, 2005.

[2] King J. Symbolic Execution and Program Testing[J]. Communications of the ACM, 1976, 19(7): 385-394.

[3] LIN Zhiqiang, ZHANG Xiangyu, XU Dongyan. Automatic Reverse Engineering of Data Structures from Binary Execution[C]. // Proceedings of 17th Annual Network & Distributed System Security Symposium(NDSS'10), San Diego, California, USA, 2010.

[4] Zhu D, Jung J, Song D, et al. TaintEraser: Protecting Sensitive Data Leaks Using Ap-

plication-Level Taint Tracking[J]. Operating Systems Review, 2011, 45(1): 142-154.

[5] SONG Chengyu, ZHUGE Jianwei, HAN Xinhui, et al. Preventing Drive-by Download via Inter-Module Communication Monitoring[C]. //Proceedings of 5th ACM Symposium on Information, Computer and Communications Security (AsiaCCS 2010), Beijing, China, 2010.

[6] Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation[C]. //Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation(PLDI'07), San Diego, California, USA, 2007.

[7] Kang M, McCamant S, Poosankam P, et al. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. // Proceedings of 18th Annual Network & Distributed System Security Symposium(NDSS'11), San Diego, California, USA, 2011.

[8] WANG Tielei, WEI Tao, GU Guofei, et al. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection[C]. // Proceedings of the 31st IEEE Symposium on Security & Privacy (Oakland'10), Oakland, California, USA, 2010.

[9] WANG Zhou, TANG Zhushou, ZHOU Kan, et al. DsVD: An Effective Low-Overhead Dynamic Software Vulnerability Discoverer[C]. // Proceedings of The 10th International Symposium on Autonomous Decentralized Systems (ISADS 2011), Kobe, Japan, 2011.

[10] Edward J, Avgerinos T, Brumley D. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask) [C]. // Proceedings of the 31st IEEE Symposium on Security & Privacy (Oakland'10), Oakland, California, USA, 2010.