基于动静态程序分析的整形漏洞检测工具

陈 平[1],韩 浩[1],沈晓斌[2],殷新春[2],茅 兵[1],谢 立[1]
(1.南京大学软件新技术国家重点实验室,南京大学计算机科学与技术系,江苏南京 210093;
2.扬州大学信息工程学院,江苏扬州 225009)

摘 要: 近几年,针对整形漏洞的攻击数目急剧上升.整形漏洞由于隐蔽性高,成为危害巨大的软件漏洞之一.本文提出了一种自动检测整形漏洞的防御工具,它结合了静态和动态程序分析技术.在静态分析阶段,该工具反编译二进制程序,并创建可疑的指令集.在动态分析阶段,该工具动态地扫描可疑集中的指令,结合可触发漏洞的输入,判断指令是否是整形漏洞.我们的工具有两个优点:首先,它提供了精确并且充足的类型信息.其次,通过基于反编译器的静态分析,工具减少了动态运行时需要检测的指令数目.实验结果表明,我们的工具可以有效地检测到实际程序中的整形漏洞,并且在我们检测的软件中,没有发现漏报,误报率也很低.

关键词: 计算机安全;软件安全;软件漏洞;整形漏洞
中图分类号: TP309 **文献标识码:** A **文章编号:** 0372-2112 (2010) 08-1741-07

Detecting Integer Bugs Based on Static and Dynamic Program Analysis

CHEN Ping[1], HAN Hao[1], SHEN Xiao-bing[2], YIN Xin-chun[2], MAO Bing[1], XIE Li[1]
(1. *State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu 210093, China*; 2. *College of Information Engineering, Yangzhou University, Yangzhou, Jiangsu 225009, China*)

Abstract: In recent years, Integer bugs have been rising sharply and become a potential threat as it is often hidden behind other bugs. In this paper, we propose a tool which can automatically detect Integer bugs. We implement the tool based on static and dynamic program analysis. In the static phase, the tool decompiles a binary and creates the suspect instruction set. In the dynamic phase, it monitors the instructions in the suspect set and generates the test cases to further detect which instructions are real Integer bugs. Our tool has two advantages. First, it provides more accurate and sufficient type information. Second, static analysis reduces the instructions which are monitored at runtime. Experimental results shows that our tool can efficiently detect the Integer bugs in several real-world programs. In addition, our tool has no false negatives and low false positives.

Key words: computer security; software security; software vulnerability; integer bugs

1 引言

整形漏洞是目前仅次于缓冲区溢出的第二大软件漏洞.它可以分为三类[1]:整形溢出,符号错误以及赋值截断.近几年,整形漏洞上升得很快.从CVE[2]近几年的统计数据可以看出,整形漏洞呈急剧上升的趋势.由1999年的1例上升到2007年的150例.

整形漏洞通常与其他程序漏洞(如缓冲区溢出,格式字符串)结合,构造攻击.由于整形漏洞隐蔽性很强,传统的检测工具报告的漏洞信息一般仅仅是整形漏洞的"表象",而不是漏洞的根源.如检测缓冲区溢出的工具SafeBird[3],以及控制流攻击检测方法[22].

在过去的几年里,一些检测整形漏洞的方法被提出来,取得了一定的成果.但整形漏洞仍然没有消失.其主要原因有三点:(1)有相当多的方法,如BLIP[28],RICH[1],SafeInt[17],IntSafe[18]等需要程序源代码,但对绝大多数商业软件来说,源代码对普通用户一般是不可得到的.(2)类型信息提取不完整.针对二进制代码的整形漏洞检测工具(如BRICK[21],IntScope[5],SmartFuzz[6])在提取类型信息的时候,没有考虑控制流依赖关系,丢失了一部分的类型信息,从而影响到整形漏洞的检测.(3)检测局限性,一些工具只能检测整形溢出,如IntScope[5],UQBTng[4].而有的工具缺少专门针对整形漏洞的动态检测工具,会产生漏报.SmartFuzz[6]产生可能

收稿日期:2009-08-21;修回日期:2009-12-13
基金项目:国家自然科学基金(No.60773171,No.90818022),国家863高技术研究发展计划(No.2007AA01Z488),国家973重点基础研究计划(No.2009CB32075),江苏省自然科学基金(No.BK2007176)

触发整形漏洞的测试用例,并通过检测工具 Memcheck[13]报告引起内存错误的测试用例. 然而引起非控制流攻击的整形漏洞[15]不能被 Memcheck 检测到,该漏洞的测试用例不会被报告,SmartFuzz 产生漏报.所以提供高实用性和高有效性的整形漏洞检测方法,仍然是研究人员不遗余力的目标.

本文提出了一种自动检测整形漏洞的二进制工具.该工具由静态和动态分析两部分组成.首先,利用反编译器将 x86 二进制程序转换成中间语言.接着,通过扩展反编译器的类型分析系统,提取完整的类型信息,并且构造可疑指令集.最后,结合动态检测工具,在可疑集中确定与漏洞相关的指令.实验结果表明,我们的工具可以有效地检测整形漏洞,误报率和漏报率都很低,并且发现 slocate-2.7 中的新漏洞.

2 整形漏洞攻击的表现形式

我们研究了 CVE 公布的 350 例整形漏洞[12],发现整形漏洞的根本原因是由于操作数的类型和其值不匹配,并且每种整形漏洞都有其特点.整形漏洞攻击通常结合特定的函数或语句产生攻击.在检测整形漏洞时,这些函数和语句也可以帮助我们从二进制程序中提取类型信息.

(1)内存分配函数.内存分配函数(如 malloc,realloc)是攻击者的首选.内存分配函数的 size 参数是无符号型整数.利用内存分配函数,有两种方法可以实现攻击.①通过整形溢出,size 参数将会得到比预期值小/大的值,从而被分配的内存要么不够,要么被耗尽.②通过符号错误,绕过比较操作,并将负值传给内存分配函数的 size 参数.

(2)内存拷贝函数.内存拷贝函数(如 memcpy,memset)包含无符号的 length 参数,这个参数决定了从源操作数拷贝数据到目标操作数的大小.如 length 参数被溢出,会引起缓冲区溢出.

(3)数组下标.数组下标被视为无符号型整数,它通常用做基地址访问内存的偏移.当攻击者将数组的下标通过整形漏洞窜改后,可以访问任意的地址空间.

(4)判断语句.判断语句,特别是有符号的比较操作通常被攻击者利用负值绕过,接着这个负值被当做一个很大的无符号数使用.这个值被用来分配或者访问内存,完成一次攻击.

3 检测方法描述

我们检测整形漏洞之前,对整形漏洞进行形式化描述,然后判断代码中是否存在所描述的整形漏洞.首先,我们分析整形漏洞涉及到程序中哪些方面的属性;第二使用什么方法判断二进制程序与这些属性的一致性.针对第一个问题,我们分析整形漏洞在程序中的表现,建立相应的漏洞模型;针对第二个问题,我们结合静态和动态方法解决.整个检测过程分三个步骤完成.(1)建立整形漏洞模型,描述漏洞相关的属性.(2)根据漏洞模型,静态扫描代码,分析可疑指令集.(3)根据漏洞模型,动态运行代码,检测可疑集中的指令是否为整形漏洞.

3.1 整形漏洞形式化描述

整形漏洞可能出现在与整形变量相关的运算操作,赋值操作等地方.但不是所有这两类指令都是整形漏洞.还需要判断整形变量的数值是否在其类型所能表示的范围内,以及该数值是否与外部输入相关.与源代码不同的是,二进制程序中没有类型的定义信息,只能从二进制的一些特定操作指令(如 SAL,SHL 等)得到这些信息.此外,还可以从第 2 节提到的函数或语句中提取类型信息.类型信息对整形漏洞的检测非常重要.

3.1.1 类型表示

我们将类型信息分为两部分,包括宽度信息和符号信息.宽度信息指变量在内存中所占的大小,用 n 表示.例如 $n=16$ 表示 16 bits 的变量.符号信息是指有无符号位,包括有符号数(signed)和无符号数(unsigned).值得指出的是,在符号错误中,某个变量存在类型冲突,即这个变量既被当成有符号数使用,又被当成无符号数使用.我们将符号冲突类型表示为 *bot*.

定义 1 使用 T 表示整形类型,T_w 表示宽度类型,T_s 表示长度类型.

定义 2 假设 $t\in T$,二元组$(\min_t,\max_t)$记录了 t 所能表示的数值的范围.$\min_t$ 表示 t 所能表示的最小值,$\max_t$ 表示最大值.

3.1.2 对整形运算以及赋值操作的抽象表示

定义 3 假设 operation(addr)表示地址为 addr 的算术运算.其中,result 表示运算结果的类型.而 opcode 表示该运算的操作名称,loperand 和 roperand 表示该运算的左右操作数.这里是对二元操作的表示方法,一元和三元运算的表示与二元运算的表示相似.

operation(addr) = {(opcode, result, loperand, roperand)}.

定义 4 假设 assignment(addr)表示地址为 addr 的赋值操作.其中,destination 表示目的操作数的类型,source_value 表示源操作数的数值.

assignment(addr) = {(destination, source_value)}.

3.1.3 对整形漏洞建模

针对 3.1.2 节中提到的运算和赋值操作,我们对其进行约束限制,以检测其是否构成整形漏洞.这种约束限制就是我们对整形漏洞的建模.我们构造如下的限制条件.

规则 1

operation(addr)→

$$\frac{min_{result} \leqslant result_value \leqslant max_{result}}{result_value = loperand \quad opcode \quad roperand}$$

规则 2

assignment(addr)→

$$\frac{min_{destination} \leqslant source_value \leqslant max_{destination}}{}$$

规则 3

$$operand's\ type\ is\ bot \rightarrow \frac{operand_value \geqslant 0}{}$$

上述的三条规则分别针对三种整形漏洞的特点检测.其中,第一条规则检测整形溢出.对一些特定的算术运算进行约束,判断算术运算得到的结果是否超过目标操作数所能表示的最大/最小值.第二条规则用来检测赋值截断.对赋值运算进行约束,判断源操作数的值是否在目标操作数类型可表示的数值范围内.第三条规则检测符号错误.用来对存在类型冲突的操作数进行约束,判断其值是否是负数.因为负数被有符号和无符号数解释成不同的数值.

3.2　静态分析

给定二进制程序,我们采用静态分析的方法从二进制程序中提取类型信息,并构造可疑集.我们在反编译器的基础上,对其类型分析进行扩展.

扩展的类型分析如下(1)利用反编译器将二进制程序转化为中间语言.(2)在该中间语言上,按照反编译的顺序,利用一些具体的函数和语句来提取信息,包括:算术/逻辑运算,判断语句,数组下标,内存分配函数以及内存拷贝函数.(3)获得类型信息后,在基本块内部按照数据流图中的数据依赖关系,传播该类型信息,并更新那些没有确定的操作数类型.(4)利用控制流图,将类型信息传播到其他基本块,并保存操作数的类型信息.

在获得较精确的类型信息以后,我们静态扫描二进制程序,将包含完整类型信息的算术运算和赋值操作作为可疑指令纳入可疑集.

3.3　动态分析

静态分析得到较大的可疑集,我们需要从中进一步确定那些真正的整形漏洞.利用二进制动态插装工具,我们对程序进行动态分析,其过程分为两部分.(1)染色分析.由于整形漏洞通常由外部输入触发,因此那些包含与外部输入相关的操作数,并且在可疑集中的指令是我们检测的对象.我们利用一些常用的外部输入函数(读文件,读网络数据包,用户输入函数)对外部输入的数据染色,并在程序运行时传播该染色标记.从可疑集中,我们筛选出那些包含与外部输入相关操作数的指令.(2)动态检测.基于类型分析,动态的运行程序,并对上述筛选出的指令进行检测.检测的方法利用3.1.3提出的约束规则.如果程序违背了三条规则之一,则检测到整形漏洞.

4　设计与实现

我们的工具建立在反编译器 Boomerang-0.3[8]和动态插装工具 PIN-2.2[7]上.它由三个部分组成:(1)类型分析组件;(2)染色分析组件;(3)动态检测组件.系统结构如图 1 所示.给定二进制程序,我们的工具利用反编译器 Boomerang[8]将程序转换成 SSA 形式的中间语言.接着我们扩展反编译器的类型分析组件,提取类型信息,并构建可疑集.第三,运行程序,在二进制动态插装工具 PIN 的基础上,构建染色分析组件找出包含外部输入数据的指令集,将这个指令集与可疑集相交,对所得集合中的指令利用动态检测组件进行检测.

4.1　类型分析组件

4.1.1　类型信息的提取

类型分析组件在检测整形漏洞中起到了非常重要的作用.类型分析组件基于反编译器 Boomerang,Boomerang 将二进制程序转换成 SSA 形式的中间语言,并提供控制流图和数据流图.基于反编译器现有的类型分析功能,我们的工具增加了额外的类型分析技术,以获得更多更准确的类型信息.列举如图 1 所示.

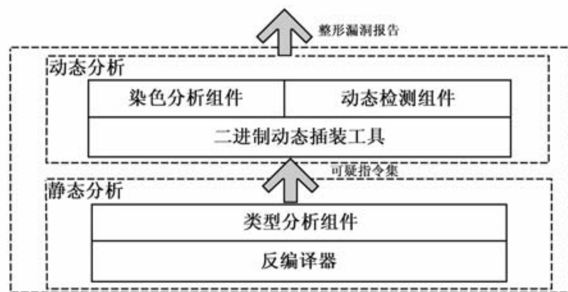

图1　检测工具结构图

(1)Boomerang 现有的类型构建仅仅考虑了特殊的运算操作如 IMUL,SAL 确定操作数的符号类型.我们的工具扩展了该方法,利用第 2 节提到的特定函数或语句提取更多的类型信息.

(2)Boomerang 记录少量的类型信息,它不保存单个操作数类型信息.为了方便类型分析,我们为语句中的内存操作数保存类型信息.此外,Boomerang 不考虑由后向前的类型传播(与反编译的方向相反),然而在检测整形漏洞时,这种后向的传播方式非常重要.例如,符号错误中,我们需要判断同一操作数现有的类型信息和已有的类型信息是否存在冲突.这就需要我们由后向前分析类型.另一个例子,在整形溢出中,如果操作数的类型信息不能在发生溢出的那条语句判断出来,当该类型信息在随后的某条语句判断出后,需要由后

向前将该类型传播到溢出语句.

(3)Boomerang 现有的类型分析通过数据流分析来提取类型信息.换句话说,它只能分析出在一个基本块内部的类型.因此如果不做控制流分析,会造成类型信息的丢失,产生漏报.我们需要利用控制流分析,在基本块之间传播操作数的类型信息,从而构建更完整的类型信息.

类型分析组件提取类型信息的方法如下,首先当反编译器将程序装载时,我们修改某些库函数的定义,例如,修改 memcpy 第三个参数的类型,从“size_t”到“unsigned int”.第二步,反编译器执行其原有的类型分析功能.第三步,利用第 2 节讨论的某些函数或语句,提取类型信息并在基本块内部传播,记录内存操作数的类型信息.第四步,我们利用控制流图,将内存操作数的类型信息传播到其他的基本块.注意到我们仅仅传播那些完整的类型信息(包含符号和长度信息).当发现类型冲突时,我们设置该操作数类型为“bot”,但不再继续传播该类型.当没有可更新的类型信息时,停止类型传播.

4.1.2 可疑集的构建

类型分析组件在提取类型信息完成后,构建可疑集.针对整形溢出,将那些目标操作数有完整类型信息的算术运算指令放入可疑集.针对符号错误,将那些包含类型冲突操作数的指令放入可疑集.针对赋值截断,将那些源操作数与目标操作数大小不一致的赋值指令放入可疑集.

4.2 染色分析组件

静态分析得到的可疑集相对较大,并且包括一些良性的指令(程序员或编译器优化加进去的).为了进一步的缩小检测指令的数量,减少程序的误报率,我们采用动态的染色技术选择那些包含外部输入数据的指令.接受外部输入的函数包括 read,fread,recv 等.我们选择这些函数作为外部输入源,并且将与这些函数相关的内存染色,根据不同的指令传播这些标记,我们仅检测那些包含染色数据的可疑指令.染色分析组件基于二进制动态插装工具 PIN[7]实现.

4.3 动态检测组件

动态检测组件也是在二进制动态插装工具 PIN[7]上实现的.它结合了静态分析得出的可疑集,以及动态染色组件,选择可疑集中包含染色数据的指令进行检测.为此,我们设计了检测策略分别针对不同类型的整形漏洞.

(1)整形溢出.针对整形溢出,我们按照 William Stallings[16]的检测方法,通过 EFLAGS 寄存器来检测整形溢出.然而对于一些特殊的运算,我们需要重新计算.例如,8/16 位的加法运算,GCC 编译器在编译时,将 8/16 位的操作数提升至 32 位,并存入 32 位的寄存器.这会将内存中的一些“脏数据”存入 32 位寄存器的高字节,当执行该加法指令时,CF 和 SF 符号位会出错.这种情况下,在检测时我们需要重新计算该加法指令.

(2)符号错误.针对符号错误,我们检测那些包含类型冲突操作数的指令,判断该操作数的值是否为负数.如果是,则报告符号错误.

(3)赋值截断.针对赋值截断,我们检测那些源和目的操作数宽度类型不一致的赋值语句,并且判断源操作数的值是否超过了目标操作数类型所能表示的范围.

5 实验结果

为了评估我们的工具对整形漏洞检测的有效性和对应用程序的性能影响,我们进行了一系列实验.测试的平台是 Pentium Dual E2180 和 2G 内存,以及 Linux Kernel 2.6.15.所有的测试程序都由 gcc-3.4.0 编译以及 glibc-2.3.3 链接.

5.1 可疑指令集

我们的工具静态构建可疑指令集,这是减少动态运行时需要检测的指令数目的第一步.表 1 显示的是针对不同程序,静态分析得到的可疑指令数目.我们的工具平均每 100K 程序静态报告 6 个可疑指令.可以看到,静态分析大大的减少需要检测的指令数量.

5.2 类型信息提取的精确性

我们的工具提供了相对精确的类型信息.我们将可疑指令包含的类型信息与源代码中相应的类型信息相比较,发现精确度达到 90% 以上.实验结果如表 1 第 7 列所示.注意到,类型提取中存在一些误报,因为反编译器很难区分如下的类型(1)指针变量和整数变量(2)指针和数组.第一种情况会使得指针运算被作为潜在的整形溢出漏洞.第二种情况,会造成指针偏移和数组偏移的类型分析不准确,因为指针偏移值应该被当成有符号数,而数组偏移值应该被当成无符号数.

表 1 可疑指令数目及其类型信息精确性

程序名称	大小	整形溢出	符号错误	赋值截断	总计	类型信息精确度
slocate-2.7	46.8K	6/6	1/1	1/1	8/8	100%
zgv-5.8	284.7K	23/24	19/19	16/16	58/59	98.3%
python-2.5.2	3.4M	95/96	21/21	61/67	177/184	96.2%
ngircd-0.8.1	329.1K	15/17	13/13	18/19	46/49	93.9%
openssh-2.2.1	150.8K	14/15	1/1	9/9	24/25	96%
mpg123-1.7.1	1.05M	7/8	4/4	1/1	12/13	92.3%
rdesktop-1.5.0	562.2K	2/2	1/1	0/0	3/3	100%

注:x/y,y 表示可疑指令数目,x 表示类型信息正确的指令数目

5.3 有效性测试

我们利用相同的软件对工具的有效性做了测试.

实验结果如表 2 所示,工具有很低的误报和漏报率.表 2 显示我们的工具能检测到所有的漏洞.我们的工具共报告了 18 个漏洞,实际的漏洞为 17 个.误报率为 6%.

表 2　整形漏洞检测

CVE #	程序名称	漏洞名称	类型	能否检测	报告漏洞	实际漏洞
2003-0326	slocate-2.7	parse_decode_path bug[9]	整形溢出	√	1	1
2004-1095	zgv-5.8	Multiple integer overflow[10]	整形溢出	√	11	11
2008-1721	python-2.5.2	zlib extension module bug[11]	符号错误	√	1	1
2005-0199	ngircd-0.8.1	List_MakeMask bug[14]	整形溢出	√	1	1
2001-0144	openssh-2.2.1	Detect_attack bug[15]	赋值截断	√	1	1
2009-1301	mpg123-1.7.1	the store_id3_text bug[20]	符号错误	√	2	1
2008-1801	rdesktop-1.5.0	Iso_recv_msg function() bug[19]	整形溢出	√	1	1

5.4　性能测试

我们的工具包含动态检测组件,我们对该组件进行性能测试.对于每个程序,我们分别测试程序单独运行和在工具的动态检测组件监控下运行的性能开销.表 3 显示了动态检测组件的性能测试结果.第 1 和 2 列给出了程序及其漏洞 CVE 编号.第 3 至第 5 列分别给出了程序正常运行,在没有插装的 PIN 下运行,以及在我们工具的动态检测组件监控下运行时间.第 6 和 7 列指出没有插装的 PIN 和我们的工具(没有染色分析)给程序运行带来的时间开销.我们发现动态检测组件的性能开销约 5.3 倍,相比较而言,没有插装功能的 PIN 的时间开销为 4.6 倍.我们认为动态检测组件适合在运行时对程序进行监控.我们同样也测试了染色分析组件的性能开销,其性能开销约为 50 倍.由于染色分析组件可以减少检测整形漏洞的误报率,但是会引入较大的性能开销,我们提供了接口以提供用户选择是否采用染色分析组件.

表 3　动态检测组件的性能开销

程序	测试(CVE #)	正常运行	PIN	工具	PIN 的性能开销	工具性能开销
slocate-2.7	2003-0326	0.08s	0.296s	0.390s	37.0X	48.7X
zgv-5.8	2004-1095	0.101s	0.348s	0.447s	3.4X	4.4X
python-2.5.2	2008-1721	0.585s	2.033s	2.592s	3.5X	4.4X
ngircd-0.8.1	2005-0199	1.156s	3.945s	4.377s	3.4X	3.8X
mpg123-1.7.1	2009-1301	0.260s	3.083s	3.458s	11.9X	13.3X
Average		0.422s	1.941s	2.253s	4.6X	5.3X

5.5　发现的新漏洞

在实验中,我们的工具找到了在 slocate-2.7 中新的符号错误漏洞.这个漏洞存在 main.c 文件中函数 decode_db 中.在该函数中,有符号变量"tot_size" 被当成 main.c 文件第 1224 行 realloc 函数的 size 参数,但是这个变量的值在被 main.c 文件第 1222 行的移位操作以后,可能得到一个负值.当数据库的大小达到"GB"时,将会触发整形漏洞.对这个漏洞,我们还没有在 slocate 邮件列表和 CVE 中得到验证和修复.

6　相关工作

6.1　静态检测工具

该方法通过静态的分析源程序来检测整形漏洞.LCLint[23]采用类型系统扩展以及局部的数据流分析对算术运算操作进行检测.Dipanwita Sarkar et al[24].基于 PREfast AST[25]提出了整形漏洞检测算法.静态分析可以用来检测整形溢出.然而由于缺少运行时操作数的值,这种方法存在较高的误报率.

6.2　安全语言

安全语言可以在编译时静态的找到潜在的漏洞,或者动态的比较运行时的值与类型信息,并给出警告.一些工作目标是将 C 语言转换为类型安全的语言,例如 CCured[26]和 Cyclone[27].它们提供了针对整形漏洞的很好的保护,但是需要人工的将 C 源程序转变成类型安全的语言.

6.3　C/C++ 编译器扩展

一些工具作为 C/C++ 编译器的补丁为编译器(如 GCC)提供安全函数.BLIP[28](Big Loop Integer Protection)对每一个 for/while 循环检测"blip".BLIP 是通过为循环设置阈值进行检测的,它会产生高的误报和漏报率.RICH[1]利用其被证明的子类型系统检测整形漏洞.它也是 GCC 编译器的一个扩展.

7　讨论

7.1　反编译器的局限性

与其他反编译器相同,Boomerang 存在局限性,会使反编译的结果不精确.(1)不能处理间接跳转.(2)不能精确地将指针与整形变量区别开来.(3)不能精确的将数组和指针区别开来.正因为这些局限性,我们的类型分析才不完美,在构建可疑指令集的时候会出现误报和漏报.

7.2　动态检测的局限性

7.2.1　语义错误

我们的工具不能检测由程序语义错误引起的整形漏洞.例如,NetBSD 的整形漏洞,因为引用计数强制转

化成 0,使得引用对象即使在使用的时候被释放.

7.2.2 逻辑操作

在当前的实现中,我们忽略了除 SHL 和 SAL 外,与逻辑操作相关的整形漏洞.我们有两点考虑:(1)在我们研究的 350 例整形漏洞中,很少有漏洞直接利用逻辑操作.(2)很难区分良性的逻辑操作和恶性的逻辑操作.(3)逻辑操作在程序中出现的频率很大.如果我们检测逻辑操作,将会带来很大的性能开销.

7.3 测试用例的局限性

为了从可疑集中找出真正的整形漏洞,我们需要构建可以触发整形漏洞的输入.在我们当前的实现中,使用了安全网站如 CVE 发布的可以触发漏洞的输入.然而这种方法只能检测到已知的漏洞.为了使我们的工具能够检测到未知的整形漏洞.我们需要构建与可疑指令相关的输入.为了达到这一目标,有以下两种方法:(1)使用符号执行构造输入来触发可疑指令;(2)使用验证工具,如动态测试工具[29]构建输入与可疑指令之间的对应关系.

8 结论

本文提出了自动检测二进制程序中整形漏洞的工具.给定一个二进制程序,该工具反编译程序,并做类型分析.在此基础上构建可疑指令集.然后,结合染色分析技术,工具动态的检测可疑集中数据被染色的指令.与其他方法相比,我们的工具提供了更加准确和完整的类型信息.并且静态分析大大降低了动态运行时需要检测指令的数量.在我们检测的软件中,我们工具没有漏报,误报率也很低.

致谢 感谢普度大学博士研究生林志强的帮助;与本课题组博士研究生王逸和辛知等的讨论也使本文受益匪浅.

参考文献:

[1] David Brumley, et al. RICH: Automatically protecting against integer-based vulnerabilities[A]. In Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'07)[C]. Reston, VA: Internet Society, 2007.351 – 363.

[2] Vulnerability Type Distributions in CEV[DB/OL]. http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf, May, 2007.

[3] 林志强,王逸等.SafeBird:一种动态和透明的运行时缓冲区溢出防御工具集[J].电子学报,2007,35(5):882 – 889.
Lin Zhiqiang, Wang Yi, et al. SafeBird: a dynamic and transparent toolkit for run-time buffer overflow preventions[J]. Acta Electronica Sinica, 2007, 35(5): 882 – 889. (in Chinese)

[4] Rafal Wojtczuk. UQBTng: a tool capable of automatically finding integer overflows in Win32 binaries[A]. 22nd Chaos Communication Congress[C]. Bielefeld: Verlag Art d'Ameublement, 2005.16 – 21.

[5] T Wang, T Wei, Z Lin, W Zou. IntScope: automatically detecting integer overflow vulnerability in x86 binary using symbolic execution[A]. Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09)[C]. San Diego, CA: Internet Society, 2009.208 – 221.

[6] David Molnar, Xue Cong Li, David Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs[A]. Proceedings of the USENIX'09 Annual Technical Conference[C]. San Jose, CA, USA: USENIX Association, 2009.67 – 82.

[7] C.-K. Luk, et al. Pin: building customized program analysis tools with dynamic instrumentation[A]. In PLDI'05[C]. Chicago, IL, USA: ACM, 2005.190 – 200.

[8] Michael James Van Emmerik. Static single assignment for decompilation[D]. Master Thesis of The University of Queensland, 2007.

[9] Integer overflow in parse decode path() of slocate[DB/OL]. http://www.cve.mitre.org/cgi-bin/cvename.cgi? name = CVE-2003-0326, 2003.

[10] Integer overflow in zgv-5.8.[DB/OL]. http://www.cve.mitre.org/cgi-bin/cvename.cgi? name = CVE – 2004 – 1095, 2004.

[11] Signedness Error in python-2.5.2[DB/OL]. http://www.cve.mitre.org/cgi-bin/cvename.cgi? name = CVE-2008 – 1721, 2008.

[12] CVE version: 20061101, CVE[DB/OL]. http://cve.mitre.org/cgi-bin/cvekey.cgi? keyword = integer.

[13] J Seward, N Nethercote. Using valgrind to detect undefined memory errors with bit precision[A]. In Proceedings of the USENIX05 Annual Technical Conference[C]. Anaheim, California, USA: USENIX Association, 2005.17 – 30.

[14] Integer underflow in ngIRCd before 0.8.2[DB/OL]. http://cve.mitre.org/cgi-bin/cvename.cgi? name = CVE-2005 – 0199.2005

[15] SSH CRC – 32 compensation attack detector vulnerability, CVE[DB/OL]. http://www.cve.mitre.org/cgi-bin/cvename.cgi? name = CVE-2001 – 0144, 2001.

[16] Computer Organization and Architecture designing for performance[M]. Willian Stallings 1996. By Prentice Hall, Inc.

[17] D. LeBlanc, Integer handling with the C++ SafeInt class [DB/OL]. http://msdn.microsoft.com/library/default.asp? url = /library/enus/dncode/html/secure01142004.asp, Jan. 2004.

[18] M. Howard et al. Safe integer arithmetic in C[DB/OL]. http://blogs.msdn.com/ michael howard/archive/2006/02/02/523392.aspx, Feb, 2006.

[19] Integer underflow in the iso_ recv_ msg function (iso.c) in

rdesktop 1.5.0[DB/OL]. http://cve. mitre. org/cgi-bin/cvename.cgi? name = CVE-2008-1801,2008.

[20] Integer signedness error in the store_id3_text function in the ID3v2 code in mpg123 before 1.7.2[DB/OL]. http://cve. mitre. org/cgi-bin/cvename. cgi? name = CVE-2009-1301, 2009.

[21] Ping Chen, et al. BRICK: a binary tool for run-time detecting and locating integer-based vulnerability[A]. Availability, Reliability and Security, International Conference[C]. Los Alamitos, CA, USA: IEEE Computer Society, 2009.208 – 215.

[22] 夏耐,郭明松,等.基于简化控制流监控的程序入侵检测[J].电子学报,2007,35(2):358 – 361.
Xia Nai, Guo Mingsong, et al. Program intrusion detection based on simplified control flow monitoring[J]. Acta Electronica Sinica, 2007, 35(2): 358 – 361. (in Chinese)

[23] David Evans, et al. LCLint: a tool for using specification to check code[A]. In Proceedings of the ACM SIGSOFT 94 Symposium on the Foundations of Software Engineering[C]. New York, USA: 1994.87 – 96.

[24] Dipanwita Sarkar et. al. Flow-insensitive static analysis for detecting integer anomalies in programs[A]. Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering[C]. Innsbruck, Austria: ACTA Press, 2007.334 – 340.

[25] Jim Larus et. al. Righting software[J]. IEEE Software, 2004, 21(3): 92 – 100.

[26] G. C. Necula, et al. CCured: type safe retrofitting of legacy code[A]. Proceedings of the Symposium on Principles of Programming Languages[C]. NY, USA: ACM Press, 2002.128 – 139.

[27] T. Jim, G. Morrisett, et al. Cyclone: a safe dialect of c[A]. USENIX Annual Technical Conference[C]. Berkley, CA, USA: USENIX Association, 2002.275 – 288.

[28] O. Horovitz. Big loop integer protection Phrack Inc. [DB/OL]. http://www. phrack. org/issues. html? issue = 60&id = 9JHJarticle. Dec, 2002.

[29] Z. Lin, et al. Convicting exploitable software vulnerabilities: an efficient input provenance based approach[A]. Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08)[C]. Anchorage, Alaska, USA: IEEE Computer Society, 2008.247 – 256.

作者简介:

陈　平　男,1985 年生于江苏兴化.南京大学计算机科学与技术系博士研究生.研究方向为系统安全.
E-mail: chenping@sns.nju.edu.cn

韩　浩　男,1985 年生于江苏南通.南京大学计算机科学与技术系硕士研究生,研究方向为系统安全.