软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(2):413−426 [doi: 10.13328/j.cnki.jos.004793]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

整数漏洞研究:安全模型、检测方法和实例*

孙 浩[1,2], 曾庆凯[1,2]

[1](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

[2](南京大学 计算机科学与技术系,江苏 南京 210023)

通讯作者: 曾庆凯, E-mail: zqk@nju.edu.cn

摘 要: C/C++语言中整型的有限表示范围、不同符号或长度间的类型转换导致了整数漏洞的发生,包括整数上溢、整数下溢、符号错误和截断错误.攻击者常常间接利用整数漏洞实施诸如恶意代码执行、拒绝服务等攻击行为.综述了整数漏洞的研究进展,从缺陷发生后行为的角度提出了新的整数漏洞安全模型,总结了判定整数漏洞的充分条件.从漏洞判定规则对充分条件覆盖的角度对现有检测方法进行比较和分析.通过实例分析,讨论了整数漏洞在现实中的特征分布.最后指出了整数漏洞研究中存在的挑战和有待进一步研究的问题.

关键词: 整数漏洞;缺陷发生后行为;安全模型;故意使用;实例研究

中图法分类号: TP311

中文引用格式: 孙浩,曾庆凯.整数漏洞研究:安全模型、检测方法和实例.软件学报,2015,26(2):413−426. http://www.jos.org.cn/1000-9825/4793.htm

英文引用格式: Sun H, Zeng QK. Research on integer-based vulnerabilities: Security model, detecting methods and real-world cases. Ruan Jian Xue Bao/Journal of Software, 2015,26(2):413−426 (in Chinese). http://www.jos.org.cn/1000-9825/4793.htm

Research on Integer-Based Vulnerabilities: Security Model, Detecting Methods and Real-World Cases

SUN Hao[1,2], ZENG Qing-Kai[1,2]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

Abstract: In C/C++ language, limited rages represented by integer types and castings between different signs or widths cause integer-based weakness, including integer overflow, integer underflow, signedness error and truncation error. Attackers usually exploit them indirectly to commit damaging acts such as arbitrary code execution and denial of service. This paper presents a survey on integer-based vulnerabilities. A novel security model is proposed in view of behaviors resulting from the weakness occurrence, and the sufficient conditions in determining integer-based vulnerabilities are also presented. A thorough comparison among detecting methods is further conducted in consideration of covering sufficient conditions. Through an empirical study on real-world integer bug cases, the characteristics and distributions are discussed. Finally, the challenges and research directions of integer-based vulnerabilities are explored.

Key words: integer-based vulnerability; behaviors after weakness; security model; intentional uses; empirical study

整型(integer type)是 C/C++语言中常用的基本数据类型,广泛应用于整数数值的表示、算术运算、内存地址、数组下标、循环计数以及标志位[1].为了满足不同的计算需求,C/C++语言提供多种字节长度的整数类型,根据具体的体系结构、编译环境和语言标准来解释整数的符号、宽度及运算规则.

计算机中采用有限字节来表示整数,意味着整型变量的取值是有严格范围的.对于由 N 比特表示的无符号

* 基金项目: 国家自然科学基金(61170070, 61431008, 61321491); 国家科技支撑计划(2012BAK26B01)
收稿时间: 2014-05-04; 修改时间: 2014-07-16; 定稿时间: 2014-11-18; jos 在线出版时间: 2014-12-12
CNKI 网络优先出版: 2014-12-12 14:08, http://www.cnki.net/kcms/detail/11.2560.TP.20141212.1408.005.html

类型,其表示范围是 0~2^N–1;由 N 比特表示的有符号整型,其表示范围则是–2^{N-1}~2^{N-1}–1.因此,算术运算如加法、减法、乘法和左移操作可能使运算结果超出相应类型的表示范围的上界或者下界,使得计算结果是期望值与相应类型极值的取模,这类计算结果的失真称为整数溢出.例如,32 位乘法 0x40000001*4 的运算结果并不是 0x100000004,而是 4.超出上界称为整数上溢(integer overflow,简称 OF),超出下界称为整数下溢(integer underflow,简称 UF).

程序中的类型转换操作会涉及不同符号和宽度的转换.符号上的转换会引起无符号类型的最高位与有符号类型的符号位之间的转换,可能引起数值上的误解.一个有符号类型的负数数值会被解释成一个很大的无符号类型的正数数值,反之亦然.例如,有符号的–1 会被解读为最大的无符号正数 2^{32}–1.这类数值上的误解在内存空间分配或者安全条件判断上会造成严重的后果,称为符号错误(signedness error,简称 Sign Err). C/C++语言设计不同宽度的整数类型,包括 char,short,int,long 和 long long.高宽度的整数类型转换成低宽度的整数类型,可能导致高宽度的整数类型在高位上的缺失,使转换前后的数值大小不等,引起截断错误(truncation error,简称 Trunc Err).例如,32 位 int 类型存放的 65540(即 0x10004)如果用 16 位 short 类型存放,数值则会转换为 4.

如果算术运算引起的计算结果失真,或者类型转换造成的数值被误解,使程序语义偏离了程序员的期望,则称该整数操作发生了整数缺陷(integer-based weakness),它们易被攻击者间接利用,实施诸如恶意代码执行、拒绝服务(denial of service)等攻击行为.国际权威漏洞披露组织 CVE(common vulnerabilities and exposures)[2]在 2007 年发布的年度报告中指出,整数漏洞(integer-based vulnerability)**已成为威胁软件安全的第二大类漏洞[4],仅次于缓冲区溢出漏洞.目前,很多研究者致力于全面、有效、精确地检测程序中潜在的整数漏洞,代表性的工作如 RICH[5],IntPatch[6],IntScope[7],BRICK[8],RA[9]和 Kint[10]等.本文统计了 CVE 公布的历年整数漏洞数量,如图 1 所示.其中,横坐标表示年份,纵坐标表示漏洞数量.

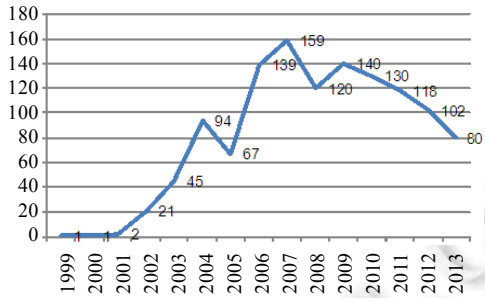

Fig.1 Number of integer bugs disclosed by CVE from 1999 to 2013

图 1 CVE 公布的 1999 年~2013 年的整数漏洞的数量

为了更好地理解整数漏洞问题,全面认识当前的研究现状,进一步完善检测方法,本文综述了整数漏洞的研究进展.第 1 节从缺陷发生后行为(behavior after weakness)的角度提出新的整数漏洞的安全模型(security model).第 2 节分类讨论现有的检测方法.第 3 节通过对现实整数漏洞的实例研究,分析其特征及分布.第 4 节提出整数漏洞问题中存在的挑战和有待研究的问题.

可以看出:整数漏洞自 2007 年开始得到一定的缓解,漏洞数量大致呈现逐年下降的趋势.可能的原因是 CVE 等发布的漏洞报告引起了人们对整数操作代码安全的重视以及整数漏洞检测研究等工作的作用.然而漏洞数量仍然居高不下,究其原因包括:

1) 整数操作在程序中普遍存在,但只有其中极少一部分可能引发程序安全问题,程序员难免会遗漏对

** 按照维基百科中的定义,软件漏洞是指可被攻击者利用以降低软件系统安全性的软件缺陷(“In computer security, a vulnerability is a weakness which allows an attacker to reduce a system's information assurance.”[3]).本文中,如果整数操作引起的计算结果失真和数值被误解,干扰了程序员对语义的理解,则称该整数操作发生了整数缺陷,将攻击者能够利用的整数缺陷称为整数漏洞.

部分整数操作的数值范围的安全检查(如观察 4 指出,Not Realize 是引入整数漏洞的主要原因).

2) 程序员对复杂整数语义(integer semantics)理解不充分[10],导致对已知整数漏洞的修复仍不安全(如第 1.3.3 节讨论).

3) 程序员会故意使用整数溢出(intentional uses of integer overflow)[11]以达到编程目的,如随机数生成(random number generation)和哈希值计算(hash value computation)等.产生的异常数值仍然在程序员的期望内,属于良性(benign)的溢出.程序员对故意使用溢出的容忍,为精确识别整数漏洞提出挑战(如第 1.3.4 节讨论).

1 整数漏洞安全模型

1.1 整数缺陷

本文将整数缺陷表示成一个由整数操作 *op*、操作数类型符号信息 *sign*、操作数类型宽度信息 *width* 和操作数数值范围信息 *range* 组成的四元组:

$$INT\_WEAK:\langle op,sign,width,range\rangle|op\in INT\_OP,range\in TRIGGER\_COND$$
$$sign\in\{\langle source\_sign,dest\_sign\rangle|source\_sign,dest\_sign\in SIGN\}$$
$$width\in\{\langle source\_width,dest\_width\rangle|source\_width,dest\_width\in WIDTH\}$$

其中,*INT_OP* 是程序中可能引发整数缺陷的操作***,包括加法 ADD、减法 SUB、乘法 MUL、左移 SHL、符号类型转换 *SIGN_CAST* 和宽度类型转换 *DOWN_CAST*.*sign* 描述了 *INT_OP* 的源操作数和目的操作数的类型符号信息对,*SIGN* 包括有符号类型 *signed* 和无符号类型 *unsigned* 两类.*width* 描述了 *INT_OP* 的源操作数和目的操作数的类型宽度信息对,*WIDTH* 用 *int*_8,*int*_16,*int*_32 和 *int*_64 来表示.*TRIGGER_COND* 指源操作数在当前(*op*,*sign*,*width*)下触发整数缺陷的数值条件(可触发整数缺陷的数值条件,是判定整数漏洞的基本.细节可参阅文献[5]的表 1 和文献[12]的表 2).以 32 位无符号加法 $x=o_1+o_2$ 为例,该操作发生整数溢出的触发条件是:操作数 o_1 和 o_2 相加后的理论结果大于 32 位无符号整型的表示上界,即 $2^{32}-1$.

定义 1(整数缺陷的判定). 程序中的整数操作(*op*,*sign*,*width*)导致整数缺陷,当且仅当源操作数的取值范围满足缺陷触发条件,即 *range*∈*TRIGGER_COND*.

1.2 基于缺陷发生后行为的整数漏洞安全模型

沿用 RICH[5]的分类,整数缺陷可分为整数上溢、整数下溢、符号错误和截断错误(可参阅文献[5]图 1 的现实漏洞实例).加法、减法、乘法和左移操作导致的整数溢出和类型转换操作导致的截断错误会使计算结果与期望值存在偏差,类型转换操作导致的符号错误会混淆负数和极大正数之间的数值解读.我们称这些受整数缺陷影响的数值为异常整数数值(malformed integer value).整数缺陷的直接危害是计算结果失真、数据丢失和数值被误解,并不会直接对内存作越界修改,所以通常不能直接利用整数缺陷实施攻击.然而,攻击者可以利用程序中使用异常整数数值的敏感操作(security-related operation)间接地实施恶意攻击,危害程序的安全性[5,12].

传统的整数漏洞安全模型[5,8,9,11]与整数缺陷模型类似,仅从操作类别、符号信息、类型信息和操作数取值范围的角度来判断整数操作是否引发漏洞.文献[6,7,12]扩展了传统的安全模型,将既能被用户输入控制又能被程序敏感操作使用的整数操作作为整数缺陷的判定对象.然而,程序中的异常整数数值还可能被程序员添加的安全检查语句过滤[6,10,13],而当前的安全模型并没有考虑这一要素.为此,本文从整数缺陷发生后行为的角度出发,提出更全面的整数漏洞安全模型,以完善对整数漏洞的描述.

本文定义了基于缺陷发生后行为的整数漏洞安全模型,即,一个由整数缺陷 *INT_WEAK*、数值是否可信 *TRUST*、安全检查 *SANITY_CHK* 和使用模式 *USE_PATTERN* 组成的四元组:

$$INT\_VULN:\langle INT\_WEAK,TRUST,SANITY\_CHK,USE\_PATTERN\rangle|TRUST,SANITY\_CHK\in Boolean$$

*** 这里描述的 *INT_OP* 是经过了隐式类型转换(implicit casting)之后的形式,即所有算术运算的操作数的符号和宽度都是相同的,类型转换操作仅在符号或宽度上不同(符号类型转换操作的宽度相同,宽度类型转换操作的符号相同).

$USE\_PATTERN \in \{NONE, MEM, ARRAY, WHILE, IF, PTR\}$

其中,*TRUST* 标识源操作数是否由程序外部通过用户输入得到,因为对于用户不能控制的整数缺陷,攻击者是无法利用的[6,12].*SANITY_CHK* 和 *USE_PATTERN* 是整数缺陷发生后的具体行为,*SANITY_CHK* 表示程序员针对可能的异常整数添加的数值过滤,如果能捕获到异常整数数值则为 true,否则为 false.*USE_PATTERN* 指异常整数数值在程序中的使用模式,包括 NONE(表示异常整数数值没有被使用,或者仅被用于与程序安全性无关的使用点[11])、MEM(与程序内存操作相关的敏感操作)、ARRAY(用于数组访问操作)、WHILE(用于循环判断语句)、IF(用于程序安全条件判断语句)和 PTR(用于指针偏移计算操作).

基于缺陷发生后行为的整数漏洞安全模型不再简单地将整数缺陷判定为整数漏洞,既考虑该缺陷是否可能会被攻击者利用,又考虑了异常整数数值是否会被程序员的安全检查代码捕获、是否会被程序安全相关的敏感操作使用.因此,整数漏洞的判定过程可定义如下:

定义 2(整数漏洞的判定). 程序中的整数操作(*op*,*sign*,*width*)导致潜在的整数漏洞,当且仅当:

$$range \in TRIGGER\_COND \wedge TRUST = \text{false} \wedge SANITY\_CHK = \text{false} \wedge USE\_PATTERN != NONE.$$

由此可见,一个整数操作满足以下 4 个充分条件即可判定为整数漏洞:

- T1:源操作数必须来自不可信数据,即能被攻击者的输入控制.
- T2:源操作数的数值范围 *range* 满足缺陷触发条件 *TRIGGER_COND*,即引发整数缺陷,产生的目的操作数为异常整数数值.
- T3:异常整数数值在随后的使用中,能绕过程序员实现的不严密的安全检查语句 *SANITY_CHK*,或者随后的使用中没有安全检查语句.
- T4:异常整数数值必须用于可影响程序安全性的敏感操作,才能被攻击者利用,以实施恶意攻击.

1.2.1 使用模式 *USE_PATTERN*

定义 3(不严重的整数缺陷 *uncritical INT_WEAK*). 程序中的整数操作(*op*,*sign*,*width*)导致不严重的整数缺陷,当且仅当 $range \in TRIGGER\_COND \wedge USE\_PATTERN = \boldsymbol{NONE}$.

不严重的整数缺陷产生的异常整数数值不会流入与程序安全性相关的敏感操作中(不满足充分条件 T4),因此不会被攻击者利用完成恶意攻击,在现实的整数漏洞检测中需要排除对这类缺陷的报告.例如,SPEC 2000[14] 164.gzip 中,deflate.c:540 处的无符号减法产生的异常数值在随后并没有被使用;186.crafty 中,iterate.c:438 处的有符号乘法产生的异常数值只是用于调试信息的打印.

按照敏感操作的类别,本文将异常整数数值的使用模式分为以下 5 类:

1) MEM 使用模式:通过影响与程序安全性相关的库函数如 *malloc*(·),*memcpy*(·),*strncpy*(·),*memset*(·)等,影响内存空间的分配、内存或字符串的复制,完成诸如缓冲区溢出、恶意代码执行、拒绝服务等攻击,是最为常见的异常数值使用模式,如实例:

   CVE-2008-1722[15],CVE-2011-1659[16],CVE-2011-0188[17].

2) ARRAY 使用模式:程序中整数的一个重要用途是数组的下标.*ARRAY* 使用模式借助实际值与期望值的偏差,完成数组越界访问攻击,如实例 CVE-2012-4405[18].

3) WHILE 使用模式:常常借助异常整数数值来影响循环判定条件,进而影响循环次数,造成无限循环(infinite loop)的攻击,如实例 CVE-2010-4164[19].

4) IF 使用模式:整数可以用于判断程序中的运行状态,条件判断语句根据整数数值来决定控制流的走向.攻击者利用异常数值与期望值的偏差或数值的误解,使程序执行错误的程序行为,如实例:

   CVE-2009-1438[20].

5) PTR 利用模式:借助异常数值与期望值的偏差,来影响指针偏移运算中的偏移值,从而完成指针越界访问攻击,如实例 CVE-2010-2500[21].

1.2.2 安全检查 *SANITY_CHK*

防止整数漏洞的一种有效方法是阻止异常整数数值流入程序的敏感操作点,因此,经验丰富的程序员会预

期整数缺陷的发生,选择对重要的整数操作(可能的整数缺陷发生点)添加数值范围的安全检查[6,10,13],以捕获异常数据.

定义 4(被拒绝的整数缺陷 *denied INT_WEAK*). 程序中的整数操作(*op*,*sign*,*width*)导致被拒绝的整数缺陷,当且仅当 $range \in TRIGGER\_COND \wedge SANITY\_CHK$=**true**.

图 2(a)描述了一个被拒绝的整数缺陷的代码实例,是针对 CUPS 的 CVE-2008-1722[15]漏洞的修复.在原始的漏洞代码中,攻击者可以恶意构造一个拥有很大 *width* 和 *height* 的 PNG 图片文件,引发第 10 行的整数乘法操作溢出,然后导致第 16 行分配的内存空间比预期少,造成缓冲区溢出.第 11 行~第 15 行的安全检查语句是开发者针对该漏洞添加的修复代码.显然,异常整数数值 *bufsize* 会被第 11 行的安全检查语句捕获,不会到达第 16 行的敏感操作,该整数溢出为被拒绝的整数缺陷.

```
/* Read a PNG image file */
1   int_cupsImageReadPNG(Cups_image_t *img,FILE *fp,...){
2     png_uint_32 width, Height;   /* width and height of image */
3     png_structp   pp;   /* PNG read pointer */
4     png_init_io(pp,fp);
5     /* Get the image dimensions */
6     png_get_IHDR(pp,info,&width,&height,...);
7     img→xsize=width;
8     img→ysize=height;
9     size_t bufsize;   /* Size of buffer */
10    bufsize=img→xsize*img→ysize;
11+   if ((bufsize/img→ysize)!=img→xsize){
12+      fprintf(stderr,"PNG image dimensions Too large!\n");
13+      fclose(fp);
14+      return 1;
15+   }
16    in=malloc(bufsize);
17    …
18 }
```

(a) 开发者针对 CUPS 的 CVE-2008-1722 整数漏洞的补丁代码

```
1   long int ap_proxy_send_fb(BUFF *f,…){
2     …
3     long remaining=0;
4     …
5     /* get the chunk size from the stream */
6     chunk_start=ap_getline(buf,buf_size,f,0);
7     if ((chunk_start≤0)||…) {
8       n=−1;
9     }
10    else {
11      remaining=ap_get_chunk_size(buf);
12      …
13    }
14    /* read the chunk */
15    if (remaining>0) {
16      n=ap_bread(f,buf,MIN((int)buf_size,
17                           (int)remaining));
18      …
19    }
20    …
21 }
```

(b) Apache 中由体系结构变更引发的截断错误

Fig.2 Real-World integer bug cases

图 2 整数漏洞实例

被拒绝的整数缺陷因不满足整数漏洞判定条件中的 T3,不会流入与程序安全性相关的敏感操作中,进而不会被攻击者用来完成恶意攻击.因此,在整数漏洞检测中,需要排除对这类缺陷的报告.

1.3 影响因素

体系结构、编译器和程序员是影响整数漏洞的重要因素,这些因素不易通过整数漏洞安全模型表示,常常被人们忽视,但却给整数漏洞的检测带来极大的挑战.

1.3.1 体系结构

体系结构决定了整数的表示宽度.在不同的体系结构下,用于表示整数的位数不同,会影响整数缺陷模型中的宽度信息 *width* 和触发缺陷的取值范围 *TRIGGER_COND*,进而影响整数缺陷的判定.相同的程序代码移植到不同的体系结构,可能会发生整数漏洞,而人们往往会忽略体系结构带来的影响.

图 2(b)描述了 Apache-1.3.42 中由体系结构变更导致的截断错误实例 CVE-2010-0010[22].在 32 位体系结构下,*int* 类型和 *long* 类型都是由 32 比特表示,然而在 64 位体系结构下,*long* 类型由 64 比特表示.该软件在 32 位体系结构下运行正常,但在 64 位体系结构下存在截断错误.具体来说,*long* 类型的 *remaining* 变量在第 11 行通过 *socket* 从外界读取数值,随后在第 17 行强制转换成 *int* 类型后被 *ap_bread*(·)调用,攻击者可以通过构造恶意长度的 *buf* 来实现第 17 行的截断错误.

1.3.2 编译器的过度优化(over optimization)

发生后验证(post-condition test)[11]是比较常用的安全检查.在整数操作之后添加对目的操作数的数值范围的检查,以判断是否发生了整数缺陷.例如,为保证无符号加法 $c=a+b$ 不受整数溢出的影响,程序员在该加法之后添加 if ($c<a$)的检查代码.然而在编译器的优化过程中,$c<a$ 会被表示为 $a+b<a$,对于无符号的整数运算规则,该条件永远为 false.在高优化级别下,此类安全检查代码会被编译器视为死代码(dead code)而自动删除掉[13],从而使程序员实现的安全检查机制失效,即,被拒绝的整数缺陷转变成潜在的整数漏洞.

1.3.3 程序员对整数操作语义的理解不准确

由于对复杂的语言标准和整数操作语义的理解不准确,程序员在对已知整数漏洞的修复中出现修复不完备或引入新漏洞的问题,造成检查代码失效,如 Wang 等人[10]的讨论,其主要原因是使用了不正确的限制边界(incorrect bounds)、不正确的检查格式(malformed checks)、符号的误用(incorrect sign)以及引入 undefined behavior[13,23].

1.3.4 整数溢出的故意使用

程序员会故意使用整数溢出来实现特定的功能,例如随机数生成、哈希值计算、消息的加密等[11].这些整数溢出是程序员所容忍的,其运算结果也在程序员的期望之内,属于良性的整数溢出,因此在整数漏洞检测中需要排除对故意使用的报告.

2 检测方法分析

由基于缺陷发生后行为的整数漏洞安全模型可知,T1~T4 是判定整数漏洞的充分条件,研究者通过识别满足其中一个或多个充分条件的整数操作来判定潜在的整数漏洞.一个整数操作满足的充分条件越多,则其成为整数漏洞的可能性越高.本文从漏洞判定规则对 4 个充分条件的满足情况出发,将当前的检测方法大致分为以下 3 类:

1) 整数缺陷的识别 *weakness_detect*:通过判定源操作数的值范围 *range* 是否满足整数缺陷的触发条件 *TRIGGER_COND* 来判定整数漏洞,即,将所有的整数缺陷都判定为整数漏洞.漏洞判定规则可描述为 Vul-R1:T2→*INT_VULN*.在实现上,分为由代码替换(如 SafeInt[24],IntSafe[25])或代码插装(如 RICH[5],IOC[11])实现的动态判定以及静态的约束求解(如 RA[9]).
2) 危险路径上整数缺陷的识别 *path_weakness_detect*:提取程序中由不可信输入点到敏感操作点之间的所有执行路径,随后判定执行路径上的整数操作是否会发生整数缺陷,包括 Kint[10],IntPatch[6],IntScope[7],SmartFuzz[26],DRIVER[12]等.漏洞判定规则可描述为 Vul-R2:T1∧T2∧T4→*INT_VULN*.
3) 危险路径上敏感操作点使用异常整数数值的识别 *sink_malform_use*:提取整数缺陷发生点到敏感操作点的执行路径,排除安全检查语句对异常整数数值的过滤,将敏感操作能使用异常整数数值作为整数漏洞的判定依据,如 SIFT[27].漏洞判定规则可描述为 Vul-R3:T1∧T2∧T3∧T4→*INT_VULN*.

2.1 整数缺陷的识别*weakness_detect*

由定义 1 可知,源操作数的数值范围 *range* 是否满足触发条件 *TRIGGER_COND* 是判定整数缺陷的充分条件.数值范围 *range* 最直观地可以在程序运行时动态得到,也可以在静态阶段得到粗略估算,因此,*weakness_detect* 方法又可分为动态和静态两类.

2.1.1 动态 *weakness_detect*

源操作数的具体数值在程序运行时是确定的,因此,对整数操作添加触发条件的判定是检测整数缺陷的有效方法.触发条件判定语句的添加可以分为代码替换(code transformation)和代码插装(code instrumentation).

代码替换方法一般在源代码上将整数操作直接替换为安全的整数操作库函数,包括 SafeInt[24],IntSafe[25],IntegerLib[28,29]和 Ranged Integer[30].例如,有符号加法操作 $x_{si}+y_{si}$ 会被替换为 $addsi(x_{si},y_{si})$,函数 *addsi* 实现了对操作数的数值范围的判断.这样,代码的可读性虽然受到影响,但其安全性得到加强.然而,代码替换方法的不足之处是安全库函数本身可能引入新的整数漏洞,如 IOC[11]从 IntegerLib 中检测出 20 个整数溢出错误.

代码插装方法一般借助编译器如 GCC 和 LLVM,在中间表示层 IR(intermediate representation)插装对数值范围的判断,不会修改源代码,例如 ARCHERR[31],RICH[4],AIR[32],IOC[10],BRICK[7]和 RICF[33].表 1 是对基于代码插装的动态 *weakness_detect* 方法的介绍,表中第 1 列表示所属的参考文献编号;第 2 列表示该文献的发表年份;第 3 列表示原型工具;第 4 列表示该方法适用的分析对象类别,包括源代码和二进制代码;第 5 列表示该方法能处理的整数漏洞类别;第 6 列表示该方法产生的动态运行开销;第 7 列描述该方法的特点.其中,ARCHERR 考虑了体系结构对整数漏洞的影响(如第 1.3.1 节讨论),为整数算术运算插装对应体系结构的判定代码.RICH 将类型安全语言中的子类型理论(sub-type theory)应用到 C 语言中,为整数操作语义定义形式化的安全规则,根据类型规则约束插装动态验证代码,插装后的代码仅有 5%的运行开销.BRICK 是第一个检测二进制整数漏洞的分析工具,但会产生 50 倍的运行开销.IOC 从软件工程的角度出发,深入探讨了整数溢出的特征(是否为程序员故意使用,是否属于 undefined behaviors).

Table 1 Dynamic *weakness_detect* methods using code instrumentation

表 1 基于代码插装的动态 *weakness_detect* 方法

文献编号	发表年份	工具名称	分析对象	漏洞类别	开销	特点
文献[31]	2004	ARCHERR	源代码	OF+UF	2.5X	考虑体系结构的影响
文献[5]	2007	RICH	源代码	4 类	5%	定义形式化的安全规则
文献[8]	2009	BRICK	二进制	4 类	50X	第 1 个检测二进制文件中整数漏洞
文献[33]	2010	RICF	二进制	OF+UF	N/A	用有限状态自动机表示验证代码
文献[32]	2010	AIR	源代码	OF+UF	6%	提供灵活的处理机制
文献[11]	2012	IOC	源代码	OF+UF	30%	从软件工作角度分析整数溢出的特征

2.1.2 静态 *weakness_detect*

整型变量在程序中的取值范围可以在静态阶段得到粗略估算.具体方法是:根据条件判断语句约束变量在不同分支下的数值范围,在遍历抽象语法树的过程中不断更新深层分支下的约束信息,从而得到变量的大致取值范围.利用这些粗略的取值范围信息来判定是否满足触发条件 *TRIGGER_COND*,包括文献[34,35].然而,静态得到的数值范围比较粗糙,会产生很多的误报.

RA[1]提出了新的值范围分析算法,以更精确、快速地求解变量的值范围,工作包括:1) 按拓扑顺序分析强连通分支(strong connected component),以加快约束求解的速度;2) 提出分阶段的方法来提取变量之间关系操作(comparison between variables)的约束;3) 设计新的中间表示形式,以加快值范围的 sparse analysis.实验结果表明:RA 能够在 10s 内处理百万行的代码规模,能够静态判定 25%的整数操作是安全的,即,取值范围不满足触发条件的整数操作.

2.2 危险路径上整数缺陷的识别*path_weakness_detect*

定义 5(危险路径 *critical path*). 由不可信输入到敏感操作的执行路径称为危险路径.

危险路径是整数漏洞多发的程序片段,攻击者可以通过不可信输入来控制程序中的数值范围,进而在敏感操作点控制异常数值来危害程序的安全性.危险路径上的整数操作满足条件 T1 和 T4,因此,提取程序的危险路径是识别整数漏洞的有效手段.

path_weakness_detect 方法通常借助污点分析(taint analysis)、类型规约(type qualifier)等方法来提取程序中的危险路径,随后,利用代码插装或静态约束求解的 *weakness_detect* 方法来判断危险路径上的整数操作是否发生整数缺陷,将报告的缺陷判定为潜在的整数漏洞.

表 2 是 *path_weakness_detect* 方法的介绍,表中第 1 列~第 5 列与表 1 前 5 列的含义相同,第 6 列表示危险路径的提取方法,第 7 列表示危险路径中敏感操作的类别,第 8 列表示整数缺陷的判定方法.

IntPatch[6],Kint[10],IntScope[7]和 IntHunter[36]主要检测流入 MEM 敏感操作的整数溢出漏洞,即 IO2BO[6]漏洞.IntPatch 采用类型规约的方法跟踪用户输入数据流向内存分配操作点,然后为路径上所有的整数操作插装验证代码.IntPatch 能够有效地抽取危险路径上的整数操作,实验结果表明:IntPatch 能够排除 90%的无关整数操作,且运行开销极低,约 1%.Kint 处理的对象是大规模系统软件,如 Linux 内核,需要程序员为关键函数和变量提供

注释以进行全局的数值范围求解.Kint 针对 Linu 内核发现了 100 多个新的整数溢出漏洞.不足之处是:Kint 因别名分析、循环展开等问题产生误报,同时也会因约束求解器的有限求解能力而产生漏报.IntScope 和 IntHunter 的分析对象则是二进制代码,根据静态提取的数值约束信息,采用符号执行的方法判定缺陷的发生.

Table 2 *path_weakness_detect* methods

表 2 *path_weakness_detect* 方法

文献编号	发表年份	工具名称	分析对象	漏洞类别	危险路径的提取方法	Use pattern	整数缺陷的判定方法
文献[6]	2010	IntPatch	源代码	OF+UF	类型规约	MEM	代码插装
文献[10]	2012	Kint	源代码	OF+UF	污点分析	MEN	静态求解
文献[7]	2009	IntScope	二进制	OF+UF	污点分析	MEM	静态求解
文献[37]	2010	IntHunter	二进制	OF+UF	污点分析	MEM	静态求解
文献[38]	2008	–	源代码	4 类	污点分析	5 类	静态求解
文献[26]	2013	DRIVER	源代码	4 类	污点分析	5 类	代码插装
文献[39]	2010	IntFinder	二进制	4 类	污点分析	5 类	代码插装
文献[25]	2009	SmartFuzz	二进制	4 类	污点分析	5 类	静态求解

Pozza 等人[37]和 DRIVER[12]采用污点分析的方法尝试提取程序中所有危险路径,随后判定危险路径的整数操作是否会触发整数缺陷.不同点在于:Pozza 等人调用 GCC 编译器内置的值范围分析 VRP(value range propagation)来判定整数缺陷,分析精度不足,存在较高的误报率;而 DRIVER 采用代码插装的方式来动态判定整数缺陷,精度相对较高.

IntFinder[38]和 SmartFuzz[26]分析的对象是二进制代码,只考虑不可信数据到整数操作的执行路径.其中,IntFinder 是对 BRICK 的改进,通过扩展反汇编的类型系统,得到相对完整的类型信息,从中抽取可疑指令集作为动态保护的对象,运行开销由原来的 50 倍降低到 5 倍左右;SmartFuzz 从程序入口为起点收集数值范围的约束信息,生成测试用例来触发整数缺陷,以验证程序的安全性,有较高的测试用例覆盖率.

2.3 危险路径上敏感操作使用异常数值的识别*sink_malform_use*

sink_malform_use 方法全面分析危险路径上整数缺陷的发生后行为,确保该缺陷产生的异常数值流入敏感操作点.如果异常数值被安全检查语句捕获,则不会将该整数缺陷判定为整数漏洞.该类方法的漏洞判定规则满足所有 4 个充分条件 T1~T4,理论上是最精确的检测方法.

目前,仅有 SIFT[27]属于 *sink_malform_use* 方法的范畴.与上述方法不同的是,SIFT 不是一个漏洞检测工具,而是一个健全的(sound)输入过滤器(input filter),即,如果一个输入能通过其过滤模块,则该输入在程序的正常使用中不会引起整数溢出错误.与 IntPatch 和 Kint 类似,SIFT 考虑的敏感操作仅仅是内存空间的分配和复制操作(MEM 使用模式),即,分析对象是 IO2BO 漏洞.

SIFT 从敏感操作点出发,进行过程间的、需求驱动的、后向的静态分析(inter-procedural,demand-driven, backward static analysis)提取所有从程序入口到敏感操作的路径上与输入域(input fields)相关的约束信息集合,该约束信息集合包含程序针对各个输入域在到达敏感操作点之前进行的所有计算.SIFT 再结合对输入文件的结构分析,构造输入过滤器.对于给定的输入,SIFT 根据约束集合中对相应输入域的条件限制来评估该输入是否能够触发溢出.SIFT 具有以下优点:

1) 健全性(soundness):传统的过滤器生成方法以一个可触发漏洞的危险路径为起点,收集这条路径的约束信息并借助符号执行生成过滤器.传统的方法需要可触发漏洞的危险路径为输入,未被分析的路径也可能引发漏洞.而 SIFT 从程序中所有的敏感操作点出发,后向地提取约束信息,能够保证覆盖所有的可能流入敏感操作点的路径.
2) 高效(efficiency):传统方法将待分析路径上所有的约束都进行收集求解,而 SIFT 只收集路径上与输入域相关的约束,这样能够极大地提高求解效率.
3) 精度(accuracy):Kint,IntScope 和 SmartFuzz 收集从程序入口到溢出操作之间的数值约束信息,并没有覆盖溢出操作之后可能存在的安全检查语句,因而会将被拒绝的整数缺陷判定为整数漏洞,产生误

报.而 SIFT 从敏感操作点出发后向收集约束,包含安全检查语句.以图 2(a)中代码片段为例,SIFT 从第 16 行的 *malloc* 函数出发后向收集约束,自然会覆盖第 11 行的安全检查,从而排除对该被拒绝的整数缺陷的误报,即,一个输入虽然能触发第 10 行的整数溢出,但 SIFT 并不会过滤该输入,因为第 11 行的安全检查能确保溢出值不会被第 16 行的敏感操作使用.

2.4 讨 论

表 3 是对这 3 类检测方法的小结,其中,第 1 列表示检测方法的名称,第 2 列表示漏洞判定规则所能满足的充分条件,第 3 列比较 3 类方法的检测精度,第 4 列介绍误报情况,第 5 列给出评价.从充分条件的满足情况可以看出,3 类方法是递进扩展的关系.

Table 3 Summary on three types of detecting methods

表 3 3 种检测方法的总结

方法	满足的充分条件	精度	误报	评价
weakness_detect	T2	低	*uncritical INT_WEAK*, *denied INT_WEAK*	无漏报.动态的方法得到很好的应用.静态的方法分析精度不高,但可被用于排除静态安全的操作或者生成测试用例.
path_weakness_detect	T1+T2+T4	中	*denied INT_WEAK*	挑战:危险路径的提取
sink_malform_use	T1+T2+T3+T4	高	无误报	挑战:危险路径的提取和安全检查代码是否严密的判定.扩展考虑更多的使用模式.

缺陷识别 *weakness_detect* 方法是检测整数漏洞的基本工作,以充分条件 T2 作为判定整数漏洞的规则.这类方法的优点是不会产生漏报,但由于没有考虑危险路径和缺陷发生后的安全检查这两个条件,因此难免会产生大量的误报(误将 *uncritical INT_WEAK* 和 *denied INT_WEAK* 判定为整数漏洞),给出的缺陷报告需要人工的进一步确认.其中,动态 *weakness_detect* 方法已经得到了很好的应用,如整数操作的安全库函数被越来越多的开发者使用、代码插装工具 IOC 已内置于最新版本的 LLVM 编译器中;静态 *weakness_detect* 方法虽然分析精度不高,但被 RA[9]应用于排除静态不满足缺陷触发条件的整数操作中,或者结合符号执行技术以生成可以触发整数缺陷的测试用例,如 Kint,IntScope 和 SmartFuzz.

危险路径上缺陷的识别 *path_weakness_detect* 方法是对 *weakness_detect* 方法的扩展,将整数漏洞的判定由整数缺陷的发生扩展到危险路径上的整数缺陷的识别.该方法初步考虑了整数缺陷发生后的行为,检测精度得到一定的提升.与 *weakness_detect* 方法相比,因为考虑了危险路径信息,因此可以排除对 *uncritical INT_WEAK* 的误报;但与 *weakness_detect* 一样,仍然无法排除对 *denied INT_WEAK* 的误报,给出的缺陷报告仍需要人工确认. *path_weakness_detect* 方法包括危险路径的抽取和整数缺陷的识别两部分:缺陷识别部分可沿用 *weakness_detect* 方法的成熟技术(如表 2 最后一列所描述);危险路径的提取是 *path_weakness_detect* 方法的重点,然而会受到指针分析、循环展开和结构体的域敏感等问题的影响[6,10,26],难免会产生漏报.如 DRIVER[12]的实验结果描述:对于 95 个已知整数漏洞,DRIVER 最多只能准确提取其中 86 个危险路径.

危险路径上敏感操作使用的异常数值的识别 *sink_malform_use* 方法是对 *path_weakness_detect* 方法的进一步扩展,将整数漏洞的判定由危险路径上的整数缺陷的发生扩展到敏感操作点使用异常数值,满足 T1~T4 的充分条件,更全面地考虑了缺陷发生后的行为,理论上是检测精度最高的方法.与 *weakness_detect* 和 *path_weakness_detect* 相比,该方法不会产生误报;但与 *path_weakness_detect* 方法类似,*sink_malform_use* 方法同样会面临危险路径提取的挑战.此外,判定安全检查代码是否严密是 *sink_malform_use* 方法的关键,因为程序员可能提供不完备的安全检查代码(如第 1.3.3 节讨论).如果对安全代码严密程度的判定出错,则会产生漏报.在现有的检测工作中,仅 SIFT 属于该类方法,但其过滤对象仅是 IO2BO,因此,*sink_malform_use* 方法的实现不够全面,需要扩展到更多的使用模式和漏洞形式中.

3 整数漏洞实例研究

针对现实漏洞的实例研究不仅能补充理论研究,还能为漏洞检测方法提供有价值的观察结论.Dietz 等人[11]从是否为故意使用和是否为 undefined behaviors[13,23]的角度对 SPEC 2000[14]中的整数溢出进行分析;Wang 等人[10]针对 Linux 内核中的整数溢出漏洞进行分析;Coker 等人[29]针对 5 个常用软件讨论整数的使用形式.这些实例研究要么只分析单一漏洞形式[10,11],要么分析的不是现实漏洞[29].

本节选取 CVE[2]于 2008 年~2013 年公布的 688 个整数漏洞作为研究对象,分析其在漏洞类别上的分布趋势;并针对其中 315 个开源软件的整数漏洞,从引入原因、使用模式和修复方式上进行深入讨论,以更全面地认识现实整数漏洞的特征.

3.1 漏洞类别的分布趋势

本文选取了 CVE 于 2008 年~2013 年公布的 688 个整数漏洞作为研究对象,称为 *full dataset*.其中,285 个(约 41.4%)漏洞存在于商业软件中,另外 403 个(约 58.6%)漏洞存在于开源软件中.图 3 描述了 4 类整数漏洞的百分比堆积柱状图,其中,横坐标分为商业软件和开源软件两部分,分别记录不同年份和总数;纵坐标表示每类整数漏洞的数量占当年总数的比例分布,柱形上的数字为相应漏洞的具体数量.

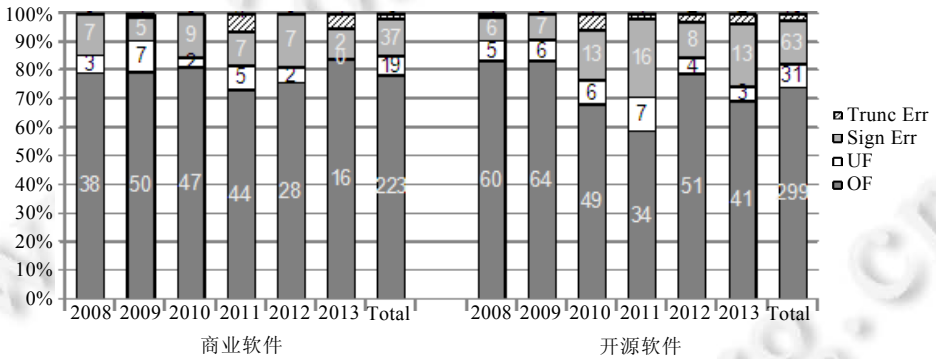

Fig.3 Distributions of integer-based vulnerability categories in commodity and open-source software respectively

图 3 商业软件和开源软件中 4 类整数漏洞的逐年分布

观察 1:总体来说,整数溢出(OF 和 UF)是整数漏洞的主要部分,商业软件中约 84.9%的整数漏洞是整数溢出,开源软件中该比例约为 81.9%;其次是符号错误(比例分别为 13.0%和 15.6%);截断错误数目最少(比例分别仅为 2.1%和 2.5%).从逐年的数量趋势来看,商业软件和开源软件的分布趋势相似,溢出漏洞都占有绝大部分的比例,符号错误紧随其后,截断错误的数目依旧很少(保持在 4 个以下,部分年份无截断错误).

观察 2:在统计的 157 个不同的软件商(vendor)中,出现整数漏洞数目较多的包括 Apple(69),Microsoft(62),Linux(51),Adobe(48),Google(38),Mozilla(22),Wireshark(20)和 GNU(14).大约有 74%的软件商仅出现 1~2 个整数漏洞.

成熟软件出现大量整数漏洞的可能原因是:

1) 成熟软件通常具有代码量大、程序结构复杂的特点,程序员难免会遗漏对其中一些整数操作的安全性分析;此外,复杂的程序结构,也增加了检测方法的分析难度.
2) 软件商如 Apple,Microsoft,Google 等所提供的是最为常用的软件,针对这些软件的攻击具有很高的商业价值,因此是攻击者的重点攻击对象.

3.2 开源软件中的整数漏洞

Full dataset 中共有 403 个整数漏洞存在于开源软件中,但由于部分漏洞没有公布明确的漏洞位置及上下文信息,我们选择其中漏洞信息明确的 315 个实例作为本节的研究对象,称为 open dataset.

3.2.1 漏洞引入原因分布

从程序员对整数操作使用的角度,我们将整数漏洞引入原因分为以下 5 类:

- Not Realize 表示程序员没有预期相应变量的数值在运行时会达到能触发漏洞的范围(即 *range* 能满足 *TRIGGER_COND*),并没有意识到该整数操作会发生缺陷.
- Incorrect Check 表示程序员预期该整数操作可能会触发缺陷,因此选择添加安全检查代码.然而,由于对整数语义的理解不准确,检查代码不严密或者检查代码本身就有缺陷(如第 1.3.3 节讨论).
- Sign Misuse 表示整数类型的符号声明出错,声明的符号与后续所有使用点的符号不同.
- Width Misuse 表示整数类型的宽度声明出错,声明的宽度与后续使用点不符合.
- Arch Change 表示因程序员没有意识到体系结构的变化,良性的代码被触发为整数漏洞.

具体的分布情况见表 4.

Table 4 Distributions of integer-based vulnerabilities in bug introduction

表 4 整数漏洞引入原因分布

Type	Not realize	Incorrect check	Sign misuse	Width misuse	Arch change
OF	212	20	0	0	1
UF	22	2	0	0	0
Sign Err	25	1	23	0	0
Trunc Err	3	0	0	4	2

观察 3:Not Realize 是整数漏洞引入的最主要原因.程序的实际执行违背了程序员的预期,程序员没有预期到这些整数操作的可能取值范围,导致整数漏洞的发生.Incorrect Check 引发的整数漏洞也占有很大的比重.

3.2.2 使用模式分布

如第 1.2.1 节讨论,使用模式是指攻击者利用异常数值完成攻击行为的方式.表 5 描述了现实漏洞在不同使用模式下的分布情况.

Table 5 Distributions of integer-based vulnerabilities in use patterns

表 5 整数漏洞的使用模式分布

Type	Use pattern				
	MEM	ARRAY	WHILE	IF	PTR
OF	179	6	14	34	1
UF	13	2	5	3	0
Sign Err	26	7	6	9	0
Trunc Err	6	1	0	2	0

观察 4:内存相关的库函数(MEM 使用模式)是整数漏洞的主要使用模式,231 个(约 73.3%)整数漏洞是通过这种方式被攻击者利用来危害程序安全性的.

MEM 使用模式涉及内存操作,是程序安全性最敏感的地方,很容易引发缓冲区溢出的攻击,因此,MEM 使用模式是攻击者的重点.大量 IO2BO 漏洞的出现,也引发专门针对 IO2BO 检测方法的出现,如 IntPatch,IntScope, Kint 和 SIFT 等.

3.2.3 修复方式分布

由于 open dataset 中部分整数漏洞没有公布相应的补丁信息,因此,我们选取其中补丁信息明确的 291 个漏洞实例作为本节的研究对象.我们将整数漏洞的修复方式分为以下 7 类:

- Precondition 表示程序员在发生漏洞的整数操作之前添加操作数是否满足触发漏洞的值范围的检测[11],这是预防整数漏洞发生的一种最有效的修复方式.例如,在无符号加法操作 $x=o_1+o_2$ 之前添加 if ($o_1>INT\_MAX-o_2$) *error*(·)的数值范围检测语句.
- Narrow 是指程序员在发生漏洞的整数操作之前添加对操作数的值范围界定.例如,对无符号加法操作 $x=o_1+100$ 之前添加 if ($o_1<200$).
- Postcondition 是指程序员添加安全检查语句以捕获异常整数数值(如图 2(a)所示).

- Width Extension 是指程序员提升操作数的类型宽度,其数值范围也得到相应的扩展,进而能够有效地表示异常整数数值.该修复方式可以修复整数溢出和截断错误.
- Sign Change 专门修复由 Sign Misuse 导致的符号错误,将误用的类型符号修正.
- Format Change 通过变更操作符类型以避免漏洞的发生.例如,将 $a+b>c$ 变更为 $a>c-b$ 以避免 $a+b$ 产生的溢出漏洞.
- Remove 是指程序员舍弃发生漏洞的代码片段.

具体的分布情况见表 6.

Table 6 Distributions of integer-based vulnerabilities in fix patterns

表 6 整数漏洞的修复方式分布

Type	Pre-Condition	Narrow	Post-Condition	Width extension	Sign change	Format change	Remove
OF	113	24	32	26	0	14	6
UF	13	1	3	0	0	3	0
Sign Err	13	1	4	0	26	0	3
Trunc Err	3	0	0	4	0	0	2

观察 5:Precondition 和 Narrow 是从限制操作数取值的角度预防整数漏洞的发生,使其不满足缺陷触发条件(使漏洞判定条件 T2 失效),约有 57.73%的漏洞实例采用此类修复方式.Postcondition 阻止异常数值流入敏感操作,使漏洞判定条件 T3 失效,约有 13.40%的漏洞实例采用此类修复方式.

4 挑战与趋势

整数漏洞是一个普遍存在且不易察觉的程序安全问题,即使是经验丰富的程序员编写的成熟软件也难以避免整数漏洞(观察 2).针对整数漏洞的检测是一个有意义的研究热点,随着大量检测方法的出现(第 2 节介绍),整数漏洞问题得到一定程度的缓解,但一些关键问题研究仍存在不足之处.具体来说,在整数漏洞研究中仍然存在以下具有挑战性的问题:

(1) 研究防治和避免整数漏洞的编程安全支持技术,以辅助程序员编写更安全的程序代码

Not Realize 是整数漏洞引入的最主要原因,一定比重的整数漏洞是由 Incorrect Check 引发(观察 3),并且程序员在修复整数漏洞时容易出错(如第 1.3.3 节讨论),因此,研究设计完善的针对整数漏洞的安全支持技术以辅助程序员编写更安全的程序代码,是从根本上防止和避免整数漏洞的途径之一.例如:

- 设计全面而准确的整数操作安全规则,指导程序员对关键的数据操作设计严格的取值限制;
- 提供安全的整数漏洞修复模式和详细的修复实例,指导程序员进行漏洞修复工作;
- 提供完善的体系结构变迁日志,明确受体系结构影响的整数操作(如第 1.3.1 节讨论),当变迁发生时实施代码变更;
- 明确编译器的优化配置,及时排除高优化级别对修复的影响(如第 1.3.2 节讨论).

(2) 预测和区分整数溢出的故意使用

如第 1.3.4 节所述,程序员会故意使用整数溢出来实现特殊功能.预测和区分这些故意使用,可以有效地降低检测方法的误报率.然而,根据整数运算操作来推理其语义信息和程序员的意图十分困难,而且当前没有涵盖故意使用溢出的整数漏洞安全模型.因此,如何预测和区分故意使用整数溢出,是当前检测方法亟需解决的问题,将会是下一步的研究热点.

(3) *sink_malform_use* 方法的全面实现(full implementation)

目前,*weakness_detect* 方法已得到很好的应用,DRIVER 和 SmartFuzz 已完成了 *path_weakness_detect* 方法的全面实现(见表 2),考虑了所有 5 类使用模式和所有 4 类漏洞形式.因此,*sink_malform_use* 方法的全面实现将会是下一步的研究重点.难点在于危险路径的抽取和安全检查语句的严密性判定:首先,与 *path_weakness_detect* 方法所面临的困难一样,路径抽取过程会受到指针分析、循环展开和结构体的域敏感等问题的影响,导致危险路径的抽取不完备;其次,如何判定安全检查语句 *SANITY_CHK* 是否能捕获异常整数数值,是排除 *denied INT_*

WEAK 误报的重点(见定义 4),也是 *sink_malform_use* 方法的重点(充分条件 T3).如果采用动态信息流跟踪(dynamic information tracking)的方法来判断异常整数数值的后续行为,则需要额外开辟内存空间来标记和更新异常数值的状态,严重影响程序的运行速度;如果采用静态约束求解的方法来判断缺陷触发条件和安全检查条件是否能够同时满足,则存在分析精度低和误报率高的问题.

(4) 二进制代码中的整数漏洞检测

在本文的实例研究中,约 41.4%的整数漏洞存在于商业软件中(如第 3.1 节讨论),并且 Apple,Microsoft,Adobe 等提供的软件中存在大量的整数漏洞(观察 2).因此,越来越多的安全研究人员将目光投入到商业软件上,试图在未知源代码的情况下检测整数漏洞.然而与源代码分析方法相比,基于二进制代码的整数漏洞检测方法(如 BRICK,IntFinder,IntScope 和 SmartFuzz)所面临的挑战是如何精确地得到整型变量的符号和宽度信息.二进制代码恢复(binary recovery)技术[39,40]致力于重构二进制代码的类型和结构信息,将会推进商业软件中整数漏洞的检测进展.

致谢 感谢审稿人的仔细评阅和中肯意见.

References:

[1] Ahmad D. The rising threat of vulnerabilities due to integer errors. IEEE Security & Privacy, 2003,1(4):77−82. [doi: 10.1109/MSECP.2003.1219077]

[2] Common vulnerabilities and exposures (CVE). http://cve.mitre.org/

[3] Definition of ‘vulnerabilty’ in computer science. 2015. http://en.wikipedia.org/wiki/Vulnerability_%28computing%29

[4] Christey S, Martin RA. Vulnerability type distributions in CVE. 2007. http://cve.mitre.org/docs/vunl-trends/vuln-trends.pdf

[5] Brumley D, Chiueh T, Johnson R, Lin H, Song D. RICH: Automatically protecting against integer-based vulnerabilities. In: Proc. of the 14th Annual Network and Distributed System Security Symp (NDSS). San Diego: Internet Society, 2007.

[6] Zhang C, Wang T, Wei TL, Chen Y, Zou W. IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In: Proc. of the 15th European Conf. on Research in Computer Security (ESORICS). Berlin, Heidelberg: Springer-Verlag, 2010. 71−86. [doi: 10.1007/978-3-642-15497-3_5].

[7] Wang TL, Wei T, Lin ZQ, Zou W. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In: Proc. of the 16th Annual Network and Distributed System Security Symp. (NDSS). San Diego: Internet Society, 2009. 1−14.

[8] Chen P, Wang Y, Xin Z, Mao B, Xie L. BRICK: A binary tool for run-time detecting and locating integer-based vulnerability. In: Proc. of the 4th Int’l Conf. on Availability, Reliability and Security. 2009. 208−215. [doi: 10.1109/ARES.2009.77]

[9] Rodrigues RE, Campos VHS, Pereira FMQ. A fast and low-overhead technique to secure programs against integer overflows. In: Porc. of the IEEE/ACM Int’l Symp. on Code Generation and Optimization (CGO). 2013. 1−11. [doi: 10.1109/CGO.2013.6494996]

[10] Wang X, Chen H, Jia Z, Zeldovich N, Kaashoek MF. Improving integer security for systems with KINT. In: Proc. of the 10th USENIX Conf. on Operating Systems Design and Implementation (OSDI). Berkeley: USENIX Association, 2012. 163−177.

[11] Dietz W, Li P, Regehr J, Adve V. Understanding integer overflow in C/C++. In: Proc. of the 2012 Int’l Conf. on Software Engineering (ICSE). Piscataway: IEEE Press, 2012. 760−770. [doi: 10.1109/ICSE.2012.6227142]

[12] Sun H, Li HP, Zeng QK. Statically detect and run-time check integer-based vulnerabilities with information flow. Ruan Jian Xue Bao/Journal of Software, 2013,24(12):2767−2781 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4385.htm [doi: 10.3724/SP.J.1001.2013.04385]

[13] Wang X, Zeldovich N, Kaashoek MF, Solar-Lezama A. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In: Proc. of the 24th ACM Symp. on Operating Systems Principles (SOSP). New York: ACM Press, 2013. 260−275. [doi: 10.1145/2517349.2522728]

[14] SPEC CPU benchmark suites. 2005. http://www.spec.org/cpu2000/

[15] CUPS integer overflow unerability. 2008. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1722

[16] Glibc crafted pattern argument integer overflow vulnerability. 2011. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1659

[17] Ruby BigDecimal class truncation vulnerability. 2011. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0188

[18] Ghostscript integer underflowvulnerability. 2012. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4405

[19] Linux kernel X25 integer underflow vulnerability. 2010. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4164

[20] Libmodplug crafted MED file integer overflow vulnerability. 2009. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1438

[21] FreeType font file integer overflow vulnerability. 2010. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2500

[22] Apache 64-bit platform integer overflow vulnerability. 2010. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0010

[23] Wang X, Chen HG, Cheung A, Jia ZH, Zeldovich N, Kaashoek MF. Undefined behaviors: What happened to my code? In: Proc. of the Asia-Pacific Workshop on System (APSys). New York: ACM Press, 2012. Article No.9. [doi: 10.1145/2349896.2349905]

[24] SafeInt class. 2004. http://safeint.codeplex.com/

[25] IntSafe. 2006. http://blogs.msdn.com/b/michael_howard/archive/2006/02/02/523392.aspx

[26] Molnar D, Li XC, Wagner DA. Dynamic test generation to find integer bugs in x86 binary linux programs. In: Proc. of the 18th USENIX Security Symp. San Diego: USENIX Association, 2009. 67−82.

[27] Long F, Douskos S, Kim D, Rinard M. Sound input filter generation for integer overflow errors. In: Porc. of the 41st ACM Symp. on Principles of Programming Languages (POPL). New York: ACM Press, 2014. 439−452. [doi: 10.1145/2578855.2535888]

[28] Seacord R. The CERT C Secure Coding Format. Boston: Addison-Wesley Professional, 2008. 139−201.

[29] Coker Z, Hafiz M. Program transforms to fix C integers. In: Proc. of the 2013 Int'l Conf. on Software Engineering (ICSE). Piscataway: IEEE Press, 2013. 792−801.

[30] Ranged integer. 2006. http://www.sei.cmu.edu/library/abstracts/reports/07tn027.cfm?RL=library&WT.ac=RLlibrary

[31] Chinchani R, Iyer A, Jayaraman B, Upadhyaya S. ARCHERR: Runtime environment driven program safety. In: Proc. of the 9th European Symp. on Research in Computer Security (ESORICS). Berlin, Heidelberg: Springer-Verlag, 2004. 385−406. [doi: 10.1007/978-3-540-30108-0_24]

[32] Dannenberg R, Dormann W, Keaton D, Plum T, Seacord RC, Svoboda D, Volkovitsky A, Wilson T. AIR: As-if infinitely ranged integer model. Technical Report, CMU/SEI-2009-TN-023, 2009.

[33] Wang Y, Ruan D, Tang Z, Xu JP, Wen M. RICF: Dynamic analysis of integer arithmetic overflow vulnerability via finite state machine. Journal of Computational Information Systems, 2010,6(6):1933−1941.

[34] Sarkar D, Jagannathan M, Thiagarajan J, Venkatapathy R. Flow-Insensitive static analysis for detecting integer anomalies in programs. In: Proc. of the 25th Conf. on IASTED Int'l Multi-Conf.: Software Engineering. ACTA Press, 2007. 334−340.

[35] Chen S, Kalbarczyk Z, Xu J, Iyer RK. A data-driven finite state machine model for analyzing security vulnerabilities. In: Proc. of the 2003 Int'l Conf. on Dependable Systems and Networks (DSN). IEEE Computer Society, 2003. 605−614. [doi: 10.1109/DSN.2003.1209970]

[36] Lu XC, Li G, Lu K, Zhang Y. High-Trusted-Software-Oriented automatic testing for integer overflow bugs. Ruan Jian Xue Bao/Journal of Software, 2010,21(2):179−193 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3785.htm [doi: 10.3724/SP.J.1001.2010.03785]

[37] Pozza D, Sisto R. A lightweight security analyzer inside GCC. In: Proc. of the 3rd Int'l Conf. on Availability, Reliability and Security (ARES). IEEE Computer Society, 2008. 851−858. [http://dx.doi.org/10.1109/ARES.2008.26]

[38] Chen P, Han H, Wang Y, Shen XB, Yin XC, Mao B, Xie L. Intfinder: Automatically detecting integer bugs in x86 binary program. In: Proc. of the Int'l Conf. on Information and Communications Security. Berlin, Heidelberg: Springer-Verlag, 2009. 336−345. [doi: 10.1007/978-3-642-11145-7_26]

[39] Lin ZQ, Zhang XY, Xu DY. Automatic reverse engineering of data structures from binary execution. In: Proc. of the 17th Annual Network and Distributed System Security Symp. (NDSS). San Diego: Internet Society, 2010.

[40] Slowinska A, Stancescu T, Bos H. Howard: A dynamic excavator for reverse engineering data structures. In: Proc. of the 18th Annual Network and Distributed System Security Symp. (NDSS). San Diego: Internet Society, 2011.

附中文参考文献:

[12] 孙浩,李会朋,曾庆凯.基于信息流的整数漏洞插装和验证.软件学报,2013,24(12):2767−2781. http://www.jos.org.cn/1000-9825/4385.htm [doi: 10.3724/SP.J.1001.2013.04385]

[36] 卢锡城,李根,卢凯,张英.面向高可信软件的整数溢出错误的自动化测试.软件学报,2010,21(2):179−193. http://www.jos.org.cn/1000-9825/3785.htm [doi: 10.3724/SP.J.1001.2010.03785]

孙浩(1987－),男,山东枣庄人,博士生,主要研究领域为信息安全,程序分析.

曾庆凯(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为信息安全,分布计算.